TechMetrix research

www.techmetrix.com

Axis: the New Incarnation of Apache SOAP

A technical White Paper by TechMetrix Research
Audience: Project Managers, Technical architects, developers

Abstract

Apache Axis is a new implementation of the SOAP specification developed by the The Apache Software Foundation, which is the successor to Apache SOAP. This new implementation aims to offer enhanced performances, and greater modularity and extensibility than its predecessor. What is more, Axis is now based on the JAXM (Java API for XML Messaging) and JAX-RPC (Java API for XML RPC) specifications currently being drawn up by Sun. With complete support for the WSDL 1.1 standard, Axis also displays greater compatibility with the latest versions of the SOAP specification (partial support for SOAP 1.2).

Apache Axis is both an environment for hosting Web services and a comprehensive development toolkit for creating services and client access to external services. This toolkit features advanced automatic code generation functionality based on the parsing of WSDL descriptions of Web Services and automatic two-way mapping between Java classes and WSDL descriptions.

Contents

1 Background

IBM has played a significant role in emancipating Web Service technology on the Java platform, and it became something of a pioneer when it created one of the very first implementations of SOAP for Java, the SOAP4J framework. Initially available as a technology preview on IBM's alphaWorks site, SOAP4J quickly aroused a lot of interest within the Java developers' community.

IBM, banking on this community's ability to ensure the success and durability of the framework, decided to donate SOAP4J to the open source world in the second half of 2000. In fact, the future of IBM's framework was entrusted to The Apache Software Foundation (renowned for having been behind some of the most prestigious open-source projects, such as the eponymous Web Server and the Tomcat Java application server). IBM has not left the limelight, however, and is continuing to contribute actively to the development of the project.

IBM's commitment as a major player in the Open Source movement:
As well as SOAP4J, IBM also instigated two other complementary frameworks in the field of Web Services: WSDL4J (Java framework for the manipulation of WSDL descriptions of Web Services) and UDDI4J (Java framework for interaction with UDDI directories, co-developed with HP). As a key player in the free software world, IBM is also behind a number of other projects, such as the XML parser Xerces (formerly XML4J), the XSL transformer Xalan (formerly LotusXSL) and more recently, the Eclipse platform (modular, extensible Java development environment).

Renamed Apache SOAP, IBM's framework underwent a number of upgrades and adjustments and enjoyed widespread success – so much so that IBM and other major vendors such as Sun quickly adopted Apache SOAP to integrate it into their J2EE application servers as a support structure for Web services.

Those interested in reading more about the SOAP protocol, the Apache SOAP implementation and its integration with application servers, can consult previous issues of our monthly newsletter TrendMarkers:

- SOAP (Simple Object Access Protocol)
- Features of Apache SOAP 2.2
- iPlanet Application Server and Web Services

Apache SOAP version 2.2, the most popular version due to its relative stability and maturity, came out in May 2001. In the space of a year, the framework underwent few adjustments, and it was only in May 2002 that version 2.3 appeared. The very latest version of Apache SOAP (2.3.1) was released extremely recently, in June 2002. This late update of the Apache SOAP seems a little odd, given that the framework has apparently reached the end of its lifetime.

In fact, for over a year, all efforts have been concentrated on the development of a successor to Apache SOAP, based on an entirely new architecture. Apache SOAP's

replacement, named Axis, has been designed to overcome its predecessor's drawbacks in terms of extensibility (due to an architecture that had become too complex, too rigid, and too closely linked to the SOAP specification) and performance (hampered by the use of the Document Object Model (DOM) for processing XML streams). Rather than continue developing Apache SOAP in increasingly difficult conditions, the Apache Foundation opted to wipe the slate clean, and set out to conquer Web Services with a whole new set of foundations. And hence the Axis project was born. More modular, with better performances, and easily extended and adapted to different specifications, Axis also offers enhanced support for Web Service specifications such as SOAP and WSDL.

In parallel, Sun and the other players participating in the development of the Java platform have been working to draw up a new family of specifications and APIs dedicated to the infrastructure of XML exchanges. This new family, the Java XML Pack, includes the following specifications:

- JAXM (Java API for XML Messaging): A set of APIs for exchanging messages formatted as XML documents. JAXM includes low-level functions for construction, routing and delivery of XML documents via SOAP 1.1 with Attachments (NB: in the JAXM latest version, 1.1, the implementation of the SOAP exchange protocol has been transferred to a new independent API, called SAAJ – SOAP with Attachments API for Java).

- JAX-RPC (Java API for XML-based RPC): A set of APIs and conventions for implementing XML-based RPC exchanges. The implementation model for RPC services used is copied from the RMI model, and uses server-side proxy classes (skeletons) and client-side proxy classes (stubs) for sending and receiving calls. JAX-RPC includes an extensible model for mapping data types along with serialization/deserialization mechanisms between Java objects and XML-Schema data types.

  Also, JAX-RPC defines two-way mapping functions between Java (interface classes, methods) and XML/WSDL descriptions of RPC service interfaces. As one of the aims of the specification is to remain independent from any particular standard, JAX-RPC is designed to adapt to different protocols (including SOAP) and different transport protocols (including HTTP and SMTP). SOAP support within JAX-RPC, provided through the JAXM functionalities, is now based on SAAJ.

In compliance with the objectives of the JAXM and JAX-RPC specifications, Axis is designed to be extensible and independent with regard to particular transport or exchange protocols. So, although it supports SOAP over HTTP, Axis will in the future support other protocols such as the forthcoming XML Protocol (based on SOAP 1.2), and other transport such as SMTP. For this reason, the name "Axis" (Apache eXtensible Interaction System) was chosen over "Apache SOAP 3.0", which was felt to be too restrictive due to the mention of the word SOAP.

Since last summer, several pre-versions of Axis have followed one after the other. After three Alpha versions and two Beta versions, a third Beta version is under preparation. We should also point out that a C++ version of Axis (on Windows and Linux platforms for the moment) is also currently being prepared. The release of the first public "final" version of Axis (which will feature the version number 3.0, to mark its succession from Apache SOAP

2.x), initially planned for the summer, has apparently been held up. However, Axis already has sufficient assets and functionality for us to be able to devote this article to the Beta 2 version which has been available since the end of April.

2 Presentation of Apache Axis Beta 2

Apache Axis is an open-source implementation on the Java platform of the SOAP version 1.1 and SOAP Messages with Attachments specifications and includes certain features of the new version of the standard, SOAP 1.2. Axis is developed by the Apache Software Foundation.

Apache Axis is both an environment for hosting Web Services and a complete toolkit for creating services and for client access to external services. This toolkit features advanced automatic code generation functionality based on parsing of WSDL descriptions of Web Services, and automatic two-way mapping between Java classes and WSDL descriptions.

2.1 New features and improvements with regard to Apache SOAP

Compared to Apache SOAP, Axis enjoys a great deal of improvements and new features:

- **Fully revised architecture**: Axis is built on entirely new foundations, based on concepts defined by the JAX-RPC specification. The architecture of Axis is modular and extensible: extensions can be easily added (for specific processing of messages, or to include new administration or management functions). Similarly, at its core, Axis contains an architecture of interchangeable transport protocols ("pluggable transport architecture"), independent of any transport protocol in particular. It is therefore possible to add on support for other transport protocols such as SMTP, FTP and so on.

- **Enhanced performance**: one of Axis' objectives is to offer improved performances (in terms of response times, memory usage and scalability) in comparison with Apache SOAP. In particular, where Apache SOAP used the DOM (Document Object Model) to parse XML streams, Axis is now based on SAX (Simple API for XML Parsing). This enables memory usage to be significantly reduced, and cuts the time required to process XML streams.

- **WSDL 1.1 support and Java/WSDL mapping functionality**: one of the important new features of Axis is its WSDL support. Axis is able to automatically generate WSDL descriptions of deployed services, on the fly. Moreover, Axis implements the Java/WSDL mapping functionality of the JAX-RPC API, which makes it possible to automatically generate a Java client from the WSDL description of a service, or to generate a WSDL description for a Java interface or class to be exposed as a Web service.

- **Enhanced datatype support**: support for datatypes from the XML-Schema (2001) standard in Apache Axis is now much more comprehensive. In particular, Apache Axis now supports numbered types, as well as untyped values.

- **New service deployment model**: like Apache SOAP, Axis uses XML deployment descriptors (WSDD – Web Service Deployment Descriptor) for deploying services. However, the new WSDD XML format is incompatible with the old format used by Apache SOAP. Axis also supports a new "instant, on-the-fly" deployment mode for Web services: it is now possible to expose a Java class as a Web service very simply, by moving a Java source file to a particular directory and changing the ".java" extension to ".jws" (Java Web Service).

- **Support for sessions and security**: Axis features user session support mechanisms (via SOAP headers, independent from the transport protocol used). Axis also integrates access authentication and authorization mechanisms which can be interfaced with the security mechanisms defined by the J2EE Servlet API.

- **Greater SOAP support**: Axis fully supports the SOAP 1.1 specification, and even offers partial compatibility with SOAP 1.2. Another new feature that was not offered by Apache SOAP is the support for the concept of actors (now called roles in SOAP 1.2) and the "mustUnderstand" attribute in SOAP header blocks. Conversely, Axis does not yet support the concept of SOAP intermediaries.

- **Improved interoperability with the other SOAP implementations on the market**. In particular, Axis is now able to process SOAP messages with untyped parameters. This brings greater interoperability with other SOAP implementations such as Microsoft's MS-SOAP, which use untyped parameters in SOAP message exchanges.

2.2 Architecture

The AxisEngine is designed with a modular architecture composed of different subsystems. This architecture comprises a hierarchy made up of two main layers. The first – fundamental – layer is consists of the *Message Flow subsystem* which defines the message circulation and processing logic within the engine. This first general layer is backed up by various specialized subsystems that implement functions specific to the transport, encoding and message model protocols for RPC exchanges, along with administration functions.

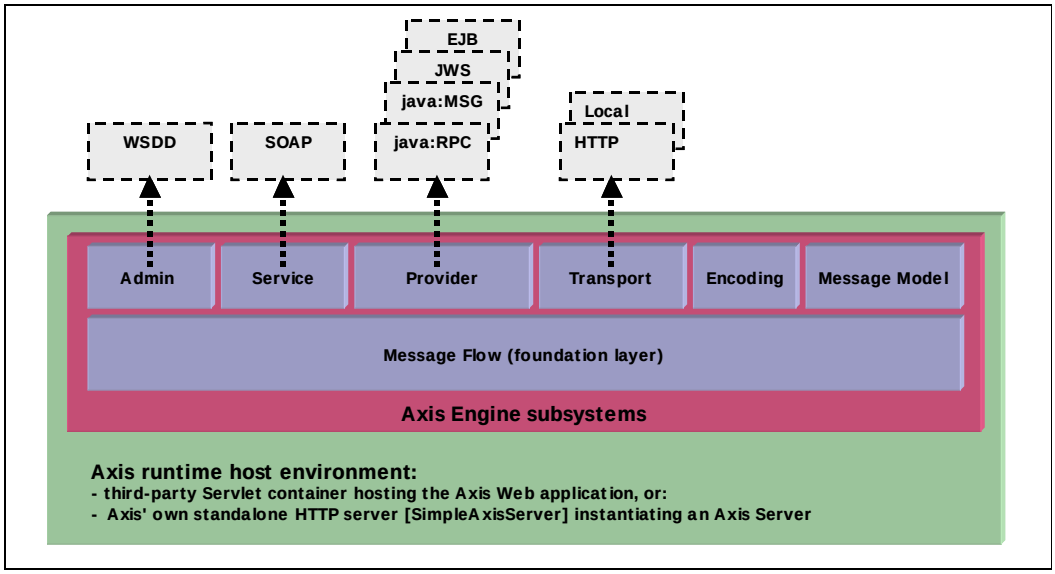
Whether Axis is used as a server (for hosting Web services) or client (for accessing Web services), the architecture stays the same: the Axis server and client use exactly the same engine. What differentiates an Axis client from an Axis server is the fact that an Axis server requires a particular execution environment.

The Axis server may be hosted in two types of environment:

- Within a Web container such as that in Apache Tomcat or another J2EE application server. The Axis distribution comprises a Web application which must be deployed on the application server. This is the most appropriate solution, and complies most closely with the relevant recommendations in the JAX-RPC specification.

- Alternatively, Axis includes a specific lightweight HTTP server ("Axis Simple Server") for hosting the Axis server. However, this is not really a viable option, except during development, as this lightweight server with its basic functionality is not suitable for production purposes.

The diagram below gives an overview of the various components of the architecture. The dotted lines show the different implementations available for a given subsystem.



Overview of subsystems of AxisEngine

The AxisEngine is responsible for orchestrating the different subsystems:

- The Administration subsystem, which handles administration and configuration of the server and various message handlers, which we shall cover in more detail in the next section.

- The Service subsystem which implements the RPC exchange protocols. For the moment, Axis only supports the SOAP protocol. However, in line with JAX-RPC, Axis is designed to be able to support several exchange protocols, such as the forthcoming XML Protocol.

- The Provider subsystem, which is used to link between the Axis server and the external component methods which are to be exposed as Web Services. Currently, Axis enables simple Java classes or stateless session beans to be exposed as Web Services. By the time the first stable version of Axis is released, support for other component models should be available, including support for component types previously supported by Apache SOAP: scripting language

objects (via the Bean Scripting Framework), and COM objects from Microsoft Windows.

- The Transport subsystem, which receives and delivers the messages from the Service subsystem. As in the JAX-RPC specification, Axis' internal architecture is totally independent of the underlying transport used for exchanges: it is therefore possible to use any type of transport, such as HTTP, SMTP, FTP, etc. However, at present, only one transport implementation is available for Axis: the HTTP protocol.

There is also another type of transport called Local, which is essentially for test purposes and operates in a local loop on one machine, enabling messages to transit between an Axis client and an Axis server deployed on the same machine. It is likely that the SMTP protocol will be supported in the first official release of Axis, as is the case for Apache SOAP.

- The Encoding subsystem. This subsystem manages encoding/decoding and serialization/deserialization between XML data types and Java types. Mapping between Java types (classes or primitives) and XML datatypes relies on the type mapping mechanisms defined by JAX-RPC. The encoding subsystem stores matches between Java types and XML types for services deployed within a Type Mapping Registry.

- The Message Model subsystem defines the structure of the different elements of a SOAP message (envelope, header, body) and also provides functionality for building and parsing the different elements of these messages. Parsing of SOAP messages is carried out according to the event-based SAX model which optimizes processing speed and memory usage.

2.2.1 Operation of engine for message exchange and handling

In order to understand the internal operation of Axis, we must look at the concepts used by the engine's key subsystem, the Message Flow Subsystem.

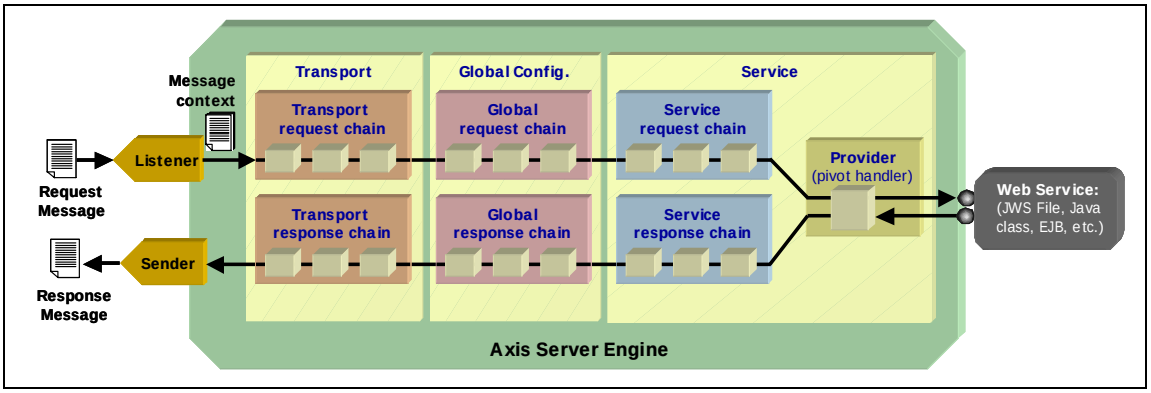
Axis is above all a message exchange and handling system, and the framework is based on three key concepts: message, flow, and handling.

A message is essentially a set of information represented by an XML data structure (in other words, a message is an XML stream). An RPC exchange is defined by the circulation or flow of messages between a client and a server: a request message, and a response message. The request and response messages circulate not only *between* the client and the server, but also within the handling and sending/receiving subsystems of the client and server.

In the Axis architecture, this circulation is essentially what underpins the operation of the server and the message handling mechanisms: the internal architecture of the Axis engine is modular and extensible, as it is based on a configurable message-handling

pipeline. The handling pipeline comprises a modulable number of stages which each carry out a specific process on the messages that pass through them.

Although this general description gives an overview of the message-handling architecture of Axis, in practice there are also more complex concepts and mechanisms, as illustrated in the diagram below:



Message flow and handling within the Axis server engine

The request or response messages are encapsulated in a more general structure called the Message Context: a message context may encompass both a request and a response message, and it also contains a "properties bag". The properties contained in this "bag" include information on the transport used (HTTP, SMTP, etc.), and on the messages themselves (in particular, the values of headers specific to SOAP messages, especially for authentication and session management mechanisms).

The processing operations on messages are encapsulated in Message Handlers. These message handlers are then grouped into Handler Chains made up of one or more handlers.

The message context objects pass successively through the message handlers of several handler chains. In turn, each message handler invoked carries out a specific operation on the message context object.

This model is remarkably similar to the filtering model of the Servlet 2.3 API (Servlet Filters), and is likewise based on a design pattern called the responsibility chain (Responsibility Chain Design Pattern): each element in the chain is responsible for a particular process, and the processes and responsibilities are clearly shared between each of the elements in the chain.

Axis' message handling system contains two separate message flow circuits: a request, or message receiving circuit, and a response, or message sending circuit. Both circuits have three specific Handler Chains:

- A **Transport Chain**, containing handlers responsible for carrying out formatting operations on the messages received or sent. As messages circulate in both

directions (request and response direction), the transport chain is split into two. Depending on the direction of circulation, each message received or sent either passes through the Request Transport Chain or the Response Transport Chain. Each transport protocol configured in Axis (e.g. HTTP, SMTP, etc.) has its corresponding transport chain. Several Transport Chains can therefore be created, each handling messages transiting by the relevant transport protocol.

- A **Global Chain** containing handlers responsible for carrying out generic operations on messages being sent or received. Once again, this global handling chain is split up into a Global Request Chain, and a Global Response Chain. All the messages that transit within the Axis server pass via the Global Chain, independent of the type of transport used in the service.

- Lastly, it is possible to configure a specific handler chain called a **Service Chain** for each Axis Web service. This chain is also split into two further chains for sending and receiving messages: Service Request Chain and Service Response Chain. Messages going to or coming from a particular service therefore pass via the specific handler chain for this service.

Each message handler chain can comprise any number of message handlers. The handler chains are configured in the configuration file of the AxisEngine. What is more, it is very easy to develop one's own message handlers for specific processes, and to deploy them in one or other of the AxisEngine handler chains.

After passing through all the handler chains, the request messages reaches the Provider. This handler dispatches the request messages to the actual service component (Java class, EJB, etc.), and receives and sends the response messages via the response chain. In Axis terminology, these provider handlers are also given the particular appellation of Pivot Handlers.

This server-side message flow schema is mirrored on the client side, if Axis is used as a client for accessing Web services. Since the architecture is the same, the Axis client also has three configurable handler chains (service, transport and global) which the request and response messages travel through. The only differences concern the order of handler chains (in reverse order to the server) and the Pivot Handler: on the client side, there is no provider handler, as the service is not on the client but on a remote server. Client-side request messages pass in turn through the Service, Global and Transport Chains, to end up at a particular handler called the Sender. On the client side, the sender plays the role of Pivot Handler: it sends requests messages to the server, receives response messages and sends them via the client's response chains.

2.3 Support for different types of SOAP exchanges

The SOAP specification defines two main types of exchange for implementing Web services: RPC-based exchanges and message-based exchanges. Apache Axis supports both these types of exchanges, and allows message- or RPC-based services to be created, as well as (client) access to both these types of service.

RPC-based exchanges constitute the highest-level implementation of services, and also the most familiar and easy to use. These services operate according to the synchronous request/response mode, and use a predefined XML message format for input and output: the SOAP envelope, made up of a SOAP header, a message body, and optional additional non-standardized message elements. In these RPC exchanges, the definition of the name, type and encoding of In and Out parameters in the SOAP message are standardized. Access to these services is simplified (in terms of programming), as the Axis framework automatically handles serialization and deserialization of parameters and return values.

Message-based exchanges, meanwhile, constitute a lower-level exchange model (on which the RPC exchange model is based), and offer complete flexibility in terms of the structure and content of exchanged messages, but at the expense of programming simplicity. Unlike the RPC-based exchange mode, the message-based mode does not define any particular rule for exchanging messages. Although requests from a message-based service must be presented as a SOAP envelope, the structure and content of the SOAP message body are free. Also, message-based services do not necessarily operate in request-response mode: the service may send no response, or may send a formatted response message inside a SOAP envelope, or may send a response message with a particular structure. Message-based exchanges therefore bring complete freedom but entail more complex programming: extraction of request message information (and if necessary within response messages) is entirely handled by the service (and if necessary, the client).

2.4 SOAP types supported and Java equivalents

Apache Axis supports all the datatypes defined in the SOAP specification: simple types (scalar types) and complex types (data structures). As regards XML encoding of data, Axis supports SOAP 1.1 encoding, SOAP 1.2 encoding, and XML-Schema encoding (1999, 2000, and 2001 versions).

The table below shows the equivalences between the Java types supported by Axis in SOAP exchanges, and the corresponding simple types from the XML Schema specification:

Primitive or Java class	XML-Schema datatype
boolean, java.lang.Boolean	xsd:boolean
byte, java.lang.Byte	xsd:byte
short, java.lang.Short	xsd:short
int, java.lang.Integer	xsd:int
long, java.lang.Long	xsd:long
float, java.lang.Float	xsd:float
double, java.lang.Double	xsd:double
java.lang.String	xsd:string
java.math.BigInteger	xsd:integer
java.math.BigDecimal	xsd:decimal
java.util.Calendar	xsd:dateTime

`java.util.Date`	`xsd:date`
`javax.xml.rpc.namespace.Qname`	`xsd:QName`
Binary stream coded using Base64 coding in a byte array `(byte[])`	`xsd:base64Binary`
Binary stream coded as a character string with hexadecimal representation `(java.lang.String)`	`xsd:hexBinary`
Any Java type whose values can be represented as a character string	`xsd:anyType`

Axis also enables complex types to be used in SOAP exchanges:

Java class	**XML-Schema or SOAP data type**
`java.util.Map`	Represented by a complex type (xsd :complexType). Each value-key pair in the map is represented by the "generic" type xsd:anyType.
`org.w3c.dom.Element`	Represented as an XML element
`java.util.Vector`	Represented by a complex type (xsd :complexType). The elements of the vector are represented by the "generic" type xsd :anyType.
Array of simple or complex type elements	Represented by a SOAP array type (soapenc :arrayType). Array elements are represented by the corresponding simple or complex type.
JavaBeans	Represented by a complex type (xsd :complexType)
Enumerations of numeric or alphanumeric values mapped to character strings or numeric types	Represented by type xsd:enumeration. Support for all simple types except Boolean, xsd :dateTime, xsd :QName, xsd :base64Binary and xsd :hexBinary

On the Axis server, mappings between Java and XML types are stored in a *Type Mapping Registry*. Each type mapping contains various information:

- An identifier (URI) describing the encoding style used for XML encoding of values of this type.
- A *Qualified Name* or *QName*: this is the unique identifier for this datatype. A QName has two parts: a namespace (URI) and type name.
- The names of Java classes to be used for serialization and deserialization of Java objects in XML.

2.5 Support for Out and In-Out parameters

The SOAP and WSDL specifications have greatly evolved, and now describe two new types of parameters for RPC exchanges: Out parameters and In-Out parameters. These new parameter types remedy one of the main limitations of the first version of the SOAP protocol, in which RPC exchanges were limited to a single return value.

The new parameter types provide greater flexibility when designing Web services based on the RPC model:

- The Out parameters can be used for implementing services sending several return values, without requiring the return values to be wrapped in a single complex type. The Web service fixes the value of the Out parameters that will be sent back to the caller.
- The In-Out parameters enable you to simulate a model for parameter passing by reference: these are in parameters whose value can be modified by the Web service, and which are returned as return parameters.

Given that the Java language itself does not natively support the concept of Out or In-Out parameters, implementation of these mechanisms poses a problem, which can be resolved by using Holder classes. The mechanism of these holder classes is defined by the JAX-RPC specification (this mechanism is similar to that used by the CORBA Java implementation).

JAX-RPC defines Holder classes for each of the primitive Java types in the `javax.xml.rpc.holders` package.

It is possible to define your own holder classes to contain an instance of a particular class. This involves creating a final (non-derivable) class which implements the interface `javax.xml.rpc.holders.Holder` (this is a marker interface, that does not define any method signature). This Holder class must:

- obey the naming convention for Holder classes: the class name must take the format: "*NameOfClassContained*Holder".
- contain a public instance variable of the required class
- define a default constructor
- define a constructor that takes as a parameter an object from the required class and assigns it to the corresponding instance variable.

Example:

Taking the "Article" class, we will define the corresponding holder class "ArticleHolder" as follows:

```
public final class ArticleHolder implements javax.xml.rpc.holders.Holder
{ // public instance variable of class Article
  public Article ref;

  public ArticleHolder()
  { // default, no-argument constructor
  }
  public ArticleHolder(Article value)
  { // parameterized constructor to assign a value to the member reference
    this.ref = value;
  }
}
```

2.6 Security support

The successive versions of the SOAP specification (1.1 and 1.2) have omitted any definition of rules or mechanisms for managing security issues (authentication and control of access to services, as well as confidentiality, integrity and non-repudiation of messages). Work is currently underway to remedy these shortcomings, and to provide extensions to the SOAP specification to define a certain number of security mechanisms. Some of these mechanisms are based on the use of information from SOAP message headers, and there are also plans to offer message signature mechanisms (SOAP Security Extensions: Digital Signature). Pending the standardization of security mechanisms for SOAP, the specification leaves the way open for implementation of specific mechanisms (based on the use of headers in messages, or using the security mechanisms of the underlying transport protocol used for SOAP exchanges).

Meanwhile, the JAX-RPC specification expressly requires support for the "Basic" HTTP authentication model for implementations of SOAP over the HTTP transport protocol.

In this context, Axis meets the requirements of the JAX-RPC specification and offers two security models that can be used via SOAP exchanges over HTTP:

- **Simple integrated security model (*Simple Security Provider*)**

  This is the default security model in Axis. The list of users and passwords for authentication must be stored in the `WEB-INF` sub-directory of the Axis Web application deployed on the host Servlet container, in a flat file called `users.lst`. Each line of this file contains a username-password pair, separated by a space.

  Another flat file called `perms.lst`, in the same directory, can be used to define access restrictions for the services published on the server. The syntax for defining access restrictions is simple: each line in the file contains the following information, separated by a space: a username, followed by one or more names of services which this user is authorized to access.

  In this model, the security credentials of the user must be retrieved from information in the HTTP header. A specific message handler carrying out the operation (`org.apache.axis.handlers.http.HTTPAuthHandler`) must therefore be deployed on the Axis server in the transport request chain for the HTTP protocol. As the default configuration of the Axis server includes deployment of this handler, no modification will generally be required on the configuration.

- **Security model interfaced with Servlet API (*Servlet Security Provider*)**

  This security model is based on the standard security models of the J2EE Servlet API, and uses the authentication and access authorization mechanisms of the host server on which the Axis server is deployed. The user referential employed to authenticate users with their password therefore depends on the Servlet container used.

As regards definition of access authorizations, this model is implicitly based on the access restriction definition mechanism of the Servlet API (definition of access restrictions in WEB-INF/web.xml deployment descriptor of Web applications). Therefore, in the deployment descriptor of the Axis Web application, it will be necessary to specify the restrictions for access to the URLs of deployed services.

Authentication is generally activated for each service deployed, by configuring deployment of the relevant message handlers in the corresponding service request chain:

- For each service requiring user authentication, the authentication handler must be deployed: `org.apache.axis.handlers.SimpleAuthenticationHandler`. By default the authentication handler uses the simple security provider (`org.apache.axis.security.simple.SimpleSecurityProvider`). If you wish to use the Servlet Security provider (`org.apache.axis.security.servlet.ServletSecurityProvider`), you will need to attribute the name of this security provider as a value of the "`securityProvider`" parameter for the authentication handler in the service deployment descriptor.

- For each service requiring access authorization checking in addition to authentication, the corresponding handler must also be deployed: `org.apache.axis.handlers.SimpleAuthorizationHandler` .

The security management functionality provided by Axis is promising, although it is somewhat rudimentary at present, and unfortunately is very poorly documented.

2.7 Session management

Axis defines an abstract representation of a session (`org.apache.axis.session.Session` class), which enables use of an extendable set of underlying implementations.

For the time being, Axis offers an implementation of this session model which employs a hashtable for storing session information (`org.apache.axis.session.SimpleSession` class). With this implementation it is possible to store any type of object in session (including non-serializable objects).

As regards transmission of the session ID in SOAP exchanges, Axis proposes a smart mechanism, independent of the underlying transport used, which is based on use of a SOAP header block for transmission of the session ID between the client and the server. (In a future version, Axis may also offer another mechanism, especially for HTTP transport, based on the use of cookies.)

A specific message handler is used for managing sessions and sending the session ID in messages: `org.apache.axis.handlers.SimpleSessionHandler`. This handler must be deployed both on the client and server side.

Evidently, the session management mechanism only works if Axis is used both as the Web Services platform and as the client for accessing these services. There are not yet any solutions to ensure interoperability in terms of session management between Axis and other Web Service implementations.

Furthermore, the documentation available on session use is still too short, and no examples are provided (oddly, the Axis FAQ does provide an example of use, though, `test.session.TestSimpleSession`, which remains impossible to find in the distribution!).

2.8 Automatic code generation

Axis features powerful code generation functionality based on the Java mapping mechanisms of JAX-RPC. These improved functions can be used via two Java utilities:

- The Java2WSDL enables you to automatically generate, from a Java interface or class, a WSDL description which represents the "translation" of this Java interface into an XML RPC interface.
- The WSDL2Java tool makes it possible to generate the following, from the WSDL description of a service:
  - a JAX-RPC framework of interfaces and client proxy classes to considerably simplify development of a client for this service
  - A JAX-RPC framework of server-side interfaces and classes for creation and deployment of a service that implements the functions defined in the WSDL description

2.9 Administration

The Administration section of Axis has been completely overhauled since Apache SOAP, in which administration and deployment were based on a Web interface.

Within the AxisEngine, the administration service (for configuration of the server and deployment of services) is now a Web service itself (message-based SOAP service), deployed as "AdminService."

However, there is still a Web administration interface (managed by the AdminServlet), although it is extremely rudimentary; it can be accessed at the following address:
http://**hostname:port**/axis/

The functions offered by this Web interface are very limited:

- the *Administer Axis* option is used to start or stop the Axis server
- the *Visit the Axis Servlet* option is used to list the services deployed on the Axis server and the names of the methods exposed by each of these services.

The majority of the administration tasks for Axis will therefore be carried out using the administration Web service, whose SOAP access point is found at the following address:
http://***hostname:port***/axis/services/AdminService

The functions offered by Axis' administration service may be divided into two categories:

- configuration of AxisEngine: configuration of global server properties, and message handler chains (global, transport and service chains), transport protocols, and mappings of types used
- deployment of services within Axis

To carry out the various administration tasks, Axis features a command-line Java SOAP client, which effectively replaces a Web interface: `org.apache.axis.client.AdminClient`.

As Axis' administration service is a Web service, the AdminClient dialogs with this service via SOAP messages. The administration commands to be performed are defined in XML command files which must be passed as parameters to the administration client.

These XML command files conform to an XML grammar specific to Axis, called "WSDD" ("Web Service Deployment Descriptor"). So, to deploy a service in Axis, you write an XML deployment file for the service ("deploy.wsdd") which is passed as a parameter to the administration client:

```
java org.apache.axis.client.AdminClient deploy.wsdd
```

The administration service examines and executes the commands contained in the deployment descriptor and adjusts the configuration of the server as necessary. The server configuration information is stored in a file called `server-config.wsdd` , which is located in the `WEB-INF` sub-directory of the Axis Web application.

This configuration file contains several sections:

- one section for the global configuration parameters of the server: `<globalConfiguration>`
- one section for each Web service deployed: `<service>`

In some cases, it is necessary to modify this configuration file manually, particularly if configuring remote access to the administration service.

To authorize remote administration of Axis, which is not enabled by default, you must directly modify the value of the "enableRemoteAdmin" parameter in the server's configuration file, in the <service> section corresponding to the "AdminService". It is also possible to define a password for accessing the administration service ("adminPassword" parameter) in the global configuration section of the server:

```
<globalConfiguration>
 <parameter name="adminPassword" value="admin"/>
  …
</globalConfiguration>
  …
<service name="AdminService" provider="java:MSG">
  …
  <parameter name="enableRemoteAdmin" value="false"/>
</service>
```

Once remote administration has been enabled, it is possible to launch administration commands from another machine using the administration client (having installed the Axis Java libraries on the remote machine beforehand). In this case, we inform the administration client of the URL of the administration service on the remote machine, using option `-l`:

```
java org.apache.axis.client.AdminClient deploy.wsdd
-lhttp://hostname:port/axis/services/AdminService
```

The format of WSDD deployment descriptors, and use of the administration client will be covered in detail in the sections on service development and deployment with Axis.

2.10 Monitoring RPC exchanges between client and server

Axis provides a graphic utility for TCP traffic monitoring (TCP Mon) which enables you to view the request and response messages exchanged between a Web service and a client.

This tool can in particular be very useful for debugging client applications.

3 Installing Axis

Axis is distributed as an archive in ZIP format, which contains the following items:

- Axis HTML documentation (`docs/` directory) and framework APIs (Javadoc)
- The various Java libraries making up the framework (`lib/` directory)
- A Web application (`webapps/axis/` directory) for hosting an Axis server in a Web container
- A series of examples and tutorials (`samples/` directory)

In addition to its own library (`axis.jar`), Axis also uses a number of additional Java libraries:

- Log4J(`log4j-core.jar`): Java open-source logging API developed as part of the Jakarta project by the Apache foundation.

- Commons Logging (`commons-logging.jar`): Also developed as part of the Jakarta project, this API aims to standardize usage of diverse logging APIs (such as Log4J or the logging API from JDK 1.4), in the same way that JAXP offers a standardized interface for using XML parsers

- WSDL4J (`wsdl4j.jar`): Reference implementation developed by IBM of the working specification ("Java Specification Request) Java APIs for WSDL (JSR-110). JSR-110 defines a standard set of APIs for building and manipulating models of descriptions of services in WSDL format.

- JAX-RPC (`jaxrpc.jar`): APIs for implementing XML-based RPC exchanges. Axis is based on version 0.8 of JAX-RPC. This API has evolved since then and version 1.0 is now available. In the future, Axis will doubtless integrate the changes that have arisen and become compatible with version 1.0.

- The TechTrader ByteCode Toolkit (`tt-bytecode.jar`): This is a library for manipulating compiled Java code (bytecode). It is used in particular to carry out optimization or profiling on Java code, or for post-processing of code after compilation, so as to add specific functionality seamlessly.

Furthermore, Axis also requires the presence of an XML parser compatible with the XML-Schema standard, and XML namespaces. The documentation recommends using either the Apache Xerces parser (version 1.4 or above, or the version 2.0 or above), or the Crimson parser, also developed by the Apache foundation.

Lastly, remember that to operate correctly, Axis must be hosted in a particular execution environment, which may be either:

- a Servlet container such as that of Apache Tomcat or another J2EE application server

- the lightweight HTTP server supplied with Axis (Axis Simple Server). Remember that this solution is not recommended for use in production, and is not highly practical for development.

For this reason, we will describe how to use Apache Tomcat as the execution environment for Axis.

3.1 Deploying Axis in the Apache Tomcat 4 .x server

To deploy the Axis server within Apache Tomcat 4.0, the procedure is as follows.

Firstly, the Axis Web application (`webapps/axis` directory) must be copied to the Web applications directory in Tomcat (`<Tomcat install. dir>/webapps/`).

Then, you need to copy the `jaxrpc.jar` archive supplied with Axis to the `<Tomcat install. dir.>/common/lib` directory. In fact, this library includes the classes placed in "`java`" and "`javax`" packages, and Tomcat does not authorize loading of any class belonging to one of these packages from the `WEB-INF/lib` directory of a Web application.

It is also necessary to copy the JAR archive of the XML parser to be used (for example, `xerces.jar` for the Xerces-J 1.4.1 parser) to Tomcat's `common/lib` directory.

3.2 Configuring the Development Environment

Access to Axis functions (for administration of the server, and for development and deployment of services) is principally through the Java utilities included in the Axis framework.

Before these functions can be used, you must modify the system `CLASSPATH`, or use a temporary `CLASSPATH` including all the libraries distributed with Axis, and the library of the XML parser (Xerces or other):

- `log4j-core.jar`
- `commons-logging.jar`
- `wsdl4j.jar`
- `jaxrpc.jar`
- `tt-bytecode.jar`
- `xerces.jar` (or library of another suitable XML parser)

Once this operation is complete, Axis is ready for use.

4 Development with Axis

In this section, we shall look at how to use Axis to develop and deploy Web services, and to create clients for existing Web Services.

Compared to Apache SOAP, Axis brings considerable improvements, and offers various development models, both for creating Web services and for developing Web service clients.

We shall study these different models through various examples, and shall then give an evaluation of the functionality provided by Axis.

4.1 JWS: drag & drop development for Web services

One of the major new features that Axis offers, compared to Apache SOAP, is a new simplified system of development for Web services: JWS (Java Web Service). This system enables Web services to be created and exposed with incredible ease: in fact, all that you have to do is rename an existing source file from .java to .jws, and then to copy that file to the WEB-INF sub-directory of the Axis Web application. No compilation is required beforehand, and the service is then ready to use.

We shall examine how this "feat" is possible, and shall develop our first Web service with Axis in a matter of minutes. We shall also look at the hidden disadvantages of this development system and shall explain why this new feature is not as appealing as it may seem at first glance.

4.1.1 Developing a JWS service with Axis

To change from the eternal "Hello World" or other conventional examples, we have chosen to develop a more original (and useful!) service: weight diagnosis!

This service, called BMICalculator, calculates the Body Mass Index of a person from their height (in meters) and their weight (in kilograms), and provides a health diagnosis (ideal weight, overweight, underweight) depending on the BMI calculated. This service features two methods:

- calculation of BMI from height and weight (computeBMI method)
- delivery of text diagnosis from a BMI (getDiagnostic method)

```
public class BMICalculator
{
  // default constructor
  public BMICalculator() {}

  /** computeBMI: calculate the body mass index
    * @param height: height in meters
```

```
    * @param weight: weight in kilograms
    * @return: body mass index (BMI)
    */
  public double computeBMI(double height, double weight)
  {
    return weight / (height * height);
  }

  /** getDiagnostic: get a health diagnostic based on a BMI value
    * @param BMI: BMI value
    * @return: health diagnostic for the BMI value (String)
    */
  public String getDiagnostic(double BMI)
  {
    if     (BMI < 17.0) return "anorexia";
    else if(BMI < 20.0) return "underweight";
    else if(BMI < 25.0) return "ideal weight";
    else if(BMI < 30.0) return "overweight";
    else                return "obesity";
  }
} // end class implementation
```

Once the source file is written and saved as "`BMICalculator.jws`", it is copied (without even needing to be compiled) to the root directory of the Axis Web application (for Tomcat, the directory is: `<Tomcat install.dir>/webapps/axis/`).

When this operation is complete, the Web service is ready for use!

The SOAP access point to our service is, quite simply:
http://localhost:8080/axis/BMICalculator.jws

In addition, Axis is able to automatically supply a WSDL description on request, which is available at the following address (simply add "?WSDL" to the access URL for the service):
http://localhost:8080/axis/BMICalculator.jws?WSDL

4.1.2 Developing an Axis client for access to the Web service

To test our service, we will develop an Axis client, using the tools shipped with Axis for automatic generation of Java/XML interfaces. We will use the WSDL2Java utility, which enables you to generate, from the WSDL description of a service, the different client interfaces and classes required to call this service on the client side.

To generate the client Java interfaces for our service, enter the following command (on a single line):

```
java org.apache.axis.wsdl.WSDL2Java
http://localhost:8080/axis/BMICalculator.jws?WSDL
-p services.bmi.client
```

You need to specify the WSDL description of the service to the WSDL2Java utility (either a WSDL file, or, as is the case here, a URL sending a WSDL service description stream). Option –p enables you to specify the Java package in which the generated classes and interfaces will be placed (if this operation is not specified, the target package is determined from the target namespace in the WSDL description.

When generating the client classes and interfaces, the Java2WSDL utility obeys the recommendations of the JAX-RPC specification and generates a certain number of Java classes and interfaces:

- one class for each complex datatype which may have been defined by the service
- one interface corresponding to the service interface with the different methods exposed by this service. In the sense of the WSDL specification, this is a portType which implements several "operations" (the methods of our service).
- A client proxy (stub) for each binding defined by the service. A binding is the implementation of a service according to a given exchange model (RPC or message-based).
- A service locator class, comparable to a factory class, which will enable a stub to be instantiated for each of the bindings defined by the service. The service locator class is used to instantiate the implementation class corresponding to the "binding" required and to retrieve the portType interface on the corresponding stub.

For our service, we do not use any complex datatypes, and only use a single portType and a single binding (RPC mode call via SOAP over HTTP). WSDL2Java therefore generates the following Java interfaces and classes:

- `BMICalculator:`	The interface of our service (portType) which comprises the different operations (methods exposed by the service) defined by the service.
- `BMICalculatorSoapBindingStub:`	The client proxy class (stub) for RPC binding of the service using the SOAP protocol over HTTP
- `BMICalculatorServiceLocator:`	The service locator class for our service
- `BMICalculatorService:`	The interface for the locator class for our service

Once these classes have been generated, it is extremely simple to develop a client for our Web service.

In the client code, we instantiate the locator class for our service, and retrieve an interface for this class. This interface enables us to retrieve an interface (a "port") to a service stub, and make remote calls to our service via this port. The only remaining task is to process any SOAP errors that may be returned when calling the service, which are wrapped in the Java exception "`Axis Fault`".

```
import services.bmi.client.*;
import org.apache.axis.AxisFault;

public class BMIServiceClient
{
  public static void main(String[] args)
  {
    if(args.length < 2)
    {
      System.out.println("Required params: [height in meters] [weight in kg.]");
      System.exit(0);
    }
```

```
    double height = Double.valueOf(args[0]).doubleValue();
    double weight = Double.valueOf(args[1]).doubleValue();

    try
    {
      // Make a service
      BMICalculatorService service = new BMICalculatorServiceLocator();
      BMICalculator        port    = service.getBMICalculator();
      // Make the actual calls to the two methods
      double BMI        = port.computeBMI(height, weight);
      String diagnostic = port.getDiagnostic(BMI);
      // Print out the results
      System.out.println("BMI        = " + BMI);
      System.out.println("Diagnostic = " + diagnostic);
    }
    catch(AxisFault af)
    {
       System.err.println(af.dumpToString());
    }
    catch(Exception e)
    {
    System.out.println("Exception caught: " + e);
    }
  }
}
```

Once the client is compiled, we can test our service quite easily from the shell (the client which we have developed retrieves the parameters to pass to the *computeBMI()* method or our service via the command line, and then calls the *getDiagnostic()* method by passing to it, as a parameter, the value of the BMI returned by the call to the first method of the service).

```
> java BMIServiceClient 1.78 63
BMI        = 19.883853048857468
Diagnostic = underweight
```

4.1.2.1 What happens in the Axis server during the call to our service?

The server identifies the request to the URL of our service (whose access point is, remember, http://localhost:8080/axis/BMICalculator.jws). It detects the .jws extension of our service which tells it that this is a JWS type Web service. The global request chain of the AxisEngine contains a special "provider" type message handler, able to process JWS Web services: JWSProvider.

The JWSProvider handler locates the service's source file in the root directory of the Web application. When the first call is made to our service, the JWSProvider handler compiles the source file and instantiates an object from the corresponding class (BMICalculator) to which it delegates the method call. The return value of the method is then retrieved, transformed into a response message, and sent back to the client.

When making subsequent calls to our service, the JWSProvider checks whether the .jws source file has been changed since its last compilation, and recompiles it if necessary. It is therefore possible to modify the source of a JWS Web service at any time: the changes are immediately and seamlessly taken into account, without needing to relaunch the Axis server.

4.1.2.2 Every silver lining has its cloud...

Unfortunately, while the JWS development model for Web services offered by Axis displays some undeniable advantages, it also suffers from certain major drawbacks, which strongly limit its usefulness in the context of "serious" developments. The JWS services suffer three main failings:

- Obligation to belong to the Java package by default: the classes defined in a .jws file must necessarily belong to the Java package by default. Using the "package" directive in a JWS source file, in order to place a class in a particular package, is therefore prohibited.
- Complex types (such as Javabeans) may not be used as In parameters or return values for a JWS service. The JWS model does not allow you to specify the type mappings necessary in the configuration of the server. With the JWS model, no server configuration is possible, which leads on to the last of our three points.
- Impossible to specify particular configuration properties for a JWS service. As well as type mapping, this limitation also concerns the possibility of configuring a chain of message handlers specific to a JWS service.

These disadvantages therefore strongly limit the utility of the JWS model for developing Web Services. Nevertheless, potential ways to overcome these shortcomings are being sought, and solutions (such as the use of deployment metadata within source files) will doubtless be found in a future version of Axis. It is, however, highly unlikely that these solutions will appear in the first official version of Axis.

4.2 WSDD flexible development and deployment model

In addition to the new JWS development model, Axis keeps a more traditional, flexible development model, which offers complete freedom of development, configuration and deployment. This development model is sometimes called the "WSDD development model," in reference to the WSDD XML schema (Web Service Deployment Descriptor) for deployment descriptors used to deploy the services developed on the Axis server.

This advanced development model allows us to take full advantage of the power and new features of Axis, while conserving a satisfactory level of productivity. The possibilities and advantages of this development model are manifold:

- Possibility of employing arbitrary datatypes by defining corresponding type mappings.
- Full control of the deployment properties of the service: service name, choice of provider component (Java class, EJB, or other), which also defines the exchange

mode used (message-based or RPC-based), list of operations (methods) to be exposed, etc.
- Possibility of defining handler chains specific to this service.
- Total freedom of use of Axis' advanced functions for designing the service: in particular, this model makes it possible to use the JAX-RPC mechanisms for automatic generation of Java/WSDL interfaces, which can be used to create the skeleton of a service from a Java interface or WSDL description, or to create a WSDL description from an existing Java class.

4.2.1.1 The different types of service providers in Axis

Development of a Web service generally involves exposing the functions of an existing component via a Web service interface, so that this component can be accessed via an RPC-XML protocol (SOAP).

Axis' Provider subsystem features a number of Provider handlers which can be used to expose various types of components as Web services. Depending on the situation, each of these providers will operate either in RPC-based exchange mode, or message-based exchange mode:

- The JWSProvider, which we used in our first Web service example, makes it easy to expose any Java class as a RPC-based Web service. As we have seen, however, this provider is specific to Axis' JWS service development model.
- The Java RPCProvider is used to expose any Java class as a Web service, in RPC-based exchange mode
- The Java MsgProvider is used to expose any Java class as a Web service, in message-based exchange mode
- The EJBProvider is used to expose an EJB as a Web service. At present, Axis documentation does not explain how to use this provider, which for the time being only enables you to expose stateless session beans in RPC-based exchange mode. In the future, this provider should come in various versions, and should enable message-driven EJBs in message-based exchange mode to be exposed, as well as stateful session EJBs in RPC-based exchange mode.

Axis will also feature two other types of providers (which were also present in Apache SOAP), which are not yet completely finalized.

- the BSFprovider will enable components written in server scripting language (JavaScript, VBScript, etc.) to be exposed as RPC-based Web services. This provider takes its name from the library which supports it: Bean Scripting Framework
- the COMProvider will be used to expose Miscrosoft Windows COM components as RPC-based Web services

The flexible development model offered by Axis allows any of these providers to be used (except for the JWSProvider) to expose a component as a Web service.

As an illustration, we shall describe how to develop a Web service according to this model, using the Java RPC provider.

For this example, we shall develop a compound interest calculator service. This service, which is more elaborate than our first one, will enable us to illustrate certain more complex development mechanisms using Axis. In particular, we shall look at:

- using complex types, and server-side / client-side type mapping
- using Axis API functions for dynamic invocation of services, which contrasts with static invocation of services (based on the automatic generation of a JAX-RPC client framework from the WSDL definition of the service) as used in the previous example. As we shall see, dynamic invocation requires much greater programming efforts than static invocation, which is very simple.

4.2.2 Developing an RPC-based Java Web Service from a Java class

The service which we shall develop is a compound interest calculator service, consisting of a method (projectEarnings) which carries out the calculation from the following parameters:

- *principal*: initial capital
- *yearlyRate*: annual interest rate (between 0 and 100%)
- *term*: period (number of years) of investment

From these parameters, our service calculates the following, for each year of the investment term:

- the initial capital at the start of the year
- the interest accrued during the year
- the capital at year end (initial capital plus interest)

All the values calculated for each year in the investment period are sent as an array of complex types as a return value for our service.

Our service will also check the In parameters, and will be able to throw an exception if the value of one of them is invalid (for example, initial capital = negative).

For this interest calculator service, it is essential for the calculation to be accurate. Therefore, we cannot employ Java floating point types (`float` or `double`) for calculations, due to their lack of precision which can cause rounding-off errors. Instead of these types, we shall use objects from the `java.math.BigDecimal` class, which enable decimals to be handled with precision. This numerical decimal type corresponds to the XML type "xsd:decimal" which is one of the simple types defined by the SOAP specification.

4.2.2.1 Developing the service

We shall start by creating the Java class for our interest calculator service, and which we shall expose as a Web service.

InterestCalculator.java

```
package services.intcalc;
import java.math.BigDecimal;

public class InterestCalculator
{
  static private final int  SCALE = 2; // precision for calculations: 2 decimal places
  static private BigDecimal ONE_HUNDRED = null; // BigDecimal constant for 100.0

  static
  {
    try { ONE_HUNDRED = new BigDecimal("100"); }
    catch(NumberFormatException nfe) { /* We're in trouble !! */ }
  }

  /** Performs a compound interest calculation projection:
    *  @param principal: the capital to invest
    *  @param yearlyRate: the annual interest rate in %
    *  @param term: the term for the projection (number of years)
    *  @return YearlyReturns[] results:an array containing the start principal,
    *                                  earnings and end principal for each year
    */
  public YearlyReturns[] projectEarnings(BigDecimal principal, BigDecimal yearlyRate,
                                         int term) throws Exception
  {
    // check input parameters
    if(principal == null || principal.signum() <= 0)
      throw new Exception("invalid null or negative value for 'principal'");
    if(yearlyRate == null || yearlyRate.signum() <= 0 ||
       yearlyRate.compareTo(ONE_HUNDRED) > 0)
      throw new Exception("invalid value for 'yearlyRate': must be in interval [0.0, 100.0]");
    if(term <= 0) throw new Exception("invalid null or negative value for 'term'");

    BigDecimal startPrincipal  = principal.setScale(SCALE, BigDecimal.ROUND_HALF_DOWN);
    BigDecimal yieldRate       = yearlyRate.setScale(SCALE, BigDecimal.ROUND_HALF_DOWN);
    yieldRate = yieldRate.movePointLeft(2); // interval shift from [0.0, 100.0] to [0.0, 1.0]

    YearlyReturns[] results = new YearlyReturns[term];
    for(int year = 0; year < term; year++)
    {
      BigDecimal earnings     = startPrincipal.multiply(yieldRate).setScale(SCALE,
                                BigDecimal.ROUND_HALF_DOWN);
      BigDecimal endPrincipal = startPrincipal.add(earnings);
      results[year] = new YearlyReturns();
      results[year].setStartPrincipal(startPrincipal);
      results[year].setYearlyEarnings(earnings);
      results[year].setEndPrincipal(endPrincipal);
      startPrincipal = endPrincipal;
    }
    return results;
  }
} // end class implementation
```

The set of values calculated for each year of the investment period is encapsulated in a JavaBean called "YearlyReturns" and is used by our service:

YearlyReturns.java

```
package services.intcalc;
import java.math.BigDecimal;

/** YearlyReturns holds the yearly details for a compound interest calculation:
  * start principal, yearly earnings, and end principal
  */
public class YearlyReturns implements java.io.Serializable
{
  public BigDecimal startPrincipal; /** principal at start of year       */
  public BigDecimal yearlyEarnings; /** interest earned through the year */
  public BigDecimal endPrincipal;   /** principal at end of year         */

  public YearlyReturns() { } // default constructor

  public BigDecimal getStartPrincipal()         { return startPrincipal; }
  public BigDecimal getYearlyEarnings()         { return yearlyEarnings; }
  public BigDecimal getEndPrincipal()           { return endPrincipal;   }

  public void setStartPrincipal(BigDecimal val) { startPrincipal = val;  }
  public void setYearlyEarnings(BigDecimal val) { yearlyEarnings = val;  }
  public void setEndPrincipal(BigDecimal val)   { endPrincipal   = val;  }
}
```

Once the implementation classes for our service have been compiled, we need to write a WSDD deployment descriptor (an XML document complying with the WSDD XML Schema used by Axis for deployment), in order to expose our service in the Axis server.

Service deployment descriptor: intcalc_deploy.wsdd

```
<deployment xmlns="http://xml.apache.org/axis/wsdd/"
            xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">
  <service name="InterestCalculatorService" provider="java:RPC">
    <parameter name="className" value="services.intcalc.InterestCalculator" />
    <parameter name="allowedMethods" value="projectEarnings" />
    <parameter name="scope" value="Application" />

    <beanMapping xmlns:ns1="urn:InterestCalculatorServiceTypes"
       qname="ns1:YearlyReturns"
       languageSpecificType="java:services.intcalc.YearlyReturns" />

    <typeMapping xmlns:ns1="urn:InterestCalculatorServiceTypes"
       qname="ns1:ArrayOfYearlyReturns"
       type="java:services.intcalc.YearlyReturns[]"
       serializer="org.apache.axis.encoding.ser.ArraySerializerFactory"
       deserializer="org.apache.axis.encoding.ser.ArrayDeserializerFactory"
       encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
  </service>
</deployment>
```

This document contains the set of deployment properties of the service, as well as the mappings of types used by the service.

The deployment properties of the service are expressed by the <service> element, whose attributes define the principal characteristics of the service:

- the "name" attribute indicates the name of the service, which will be used, in particular, to define the SOAP end-point for this service, which takes the form: `http://<hostname>:<port>/axis/services/<Name of Service>`
- the "provider" attribute defines the type of service provider which will be used to carry out the implementation of this service. Our service uses Axis' Java RPCProvider, which corresponds to the value "java:RPC" specified in this attribute.

The rest of the properties of our service are defined via <parameter> elements which define the name and the value of different properties:

- The "className" parameter is used to specify the full name of the service's Java implementation class.
- The "allowedMethods" parameter is used to define the list of methods which will be exposed by the service. The value of this parameter will therefore be a list of one or more method names, separated by spaces. The special "*" value indicates that you wish to expose all the methods of a service.
- The "scope" parameter is used to accurately define the range of Web service instances which will be created by the Axis server for handling client requests. Three options are possible:
  - "Application": a single instance of the Web service will be created by Axis and used to process all requests to the service
  - "Request": each individual request to the Web Service will entail the creation of a dedicated instance for processing this request. Except in specific cases, this option is not recommended as it naturally causes performances to deteriorate when processing a high number of simultaneous calls
  - "Session": this option is very worthwhile if you wish to benefit from Axis' session management mechanisms to implement stateful services. In this case, the Axis server creates an instance of the service for each client accessing the service. The set of requests from one client will therefore be processed by one particular instance of the service.

After the deployment properties of the service comes the definition of the mappings of complex types used by the service.

Each type mapping is identified by a unique Qualified Name (QName). A QName is made up of a URI (which may be either a URL or URN), and a type name, which can be chosen freely. For "object array" types, the convention is for the type name to comprise a prefix "ArrayOf" followed by the name of the type of object contained in the array. Next, we specify the full name of the corresponding Java type. Then, we specify a URI to define the encoding to be used to translate the datatype into XML. Lastly, we must indicate which Java classes will be used for serialization and deserialization of the datatype to/from its XML representation. In fact, it is not the serialization/deserialization classes themselves that must be specified, but the factory classes that will be used to obtain instances of Java classes for serialization/deserialization of the datatype.

Axis includes a full set of factory classes and serializer/deserializer classes for standard Java classes and primitives, arrays, JavaBeans, and all the Java datatypes equivalent to the datatypes defined in the SOAP specification. As a general rule, it will be unnecessary to develop specific serialization/deserialization classes in Axis.

Our service uses a specific type of object: the "YearlyReturns" JavaBean, which we must declare. We must also declare the "YearlyReturns object array".

For the type mapping of a JavaBean component, we can use a simplified mapping declaration using the `<beanMapping>` element, which avoids having to specify the Java factory classes for serialization/deserialization. For the declaration of the "object array" type mapping, we use the standard type mapping declaration via the `<typeMapping>` element.

4.2.2.2 Deploying the service

The first stage in deployment of the service involves using the Axis administration tool to register our service in the server configuration. We therefore call the "AdminClient" utility, for which we supply the service deployment descriptor as a parameter:

```
java org.apache.axis.client.AdminClient intcalc_deploy.wsdd
```

The service is now declared in the server's configuration file. Unfortunately, the deployment utility does not take charge of the deployment of the service's implementation classes within the Web application, and so this must be done manually.

The second stage of deployment consists in copying the tree structure of the different classes making up our service (`services/intcalc/*.class`) in the class tree of the Axis Web application on the Tomcat server. To carry out this operation, two approaches are possible:

- Copy the class tree for the service into the class tree of the Axis Web application (directory: `<Tomcat install. dir.>/webapps/axis/WEB-INF/classes`).
- Archive the class tree for the service in a JAR file and copy the JAR archive into the library directory of the Axis Web application (directory: `<Tomcat install. dir.>/webapps/axis/WEB-INF/lib`).

Once this is done, the Axis Web application must reboot in order to take the changes into account. Once again, various approaches are possible:

- The harshest solution is to stop and reboot the Tomcat server in order to reinitialize the Axis Web application so that it can detect the newly copied classes. This may be an option during the development phase, but it is much less acceptable in a production environment.

- An alternative to rebooting Tomcat is to configure the Axis Web application as "dynamically reloadable". This involves editing the Tomcat configuration file (`<Tomcat install. dir/conf/server.xml`), and adding a `<Context …>` section for the Axis Web application. As well as the Web application's name (`path` attribute),

its root directory (`docBase` attribute), this section should explicitly indicate that dynamic reloading of the application is requested (via the `reloadable` which should be given the value "true"):

```
<Context path="/axis" docBase="axis" debug="0"
         cookies="true" reloadable="true" >
```

Once this modification has been made, Tomcat will regularly check the `lib/` and `classes/` sub-directories of the Web application to detect modifications, and will reboot the Web application automatically if necessary. This checking mechanism does slow down Tomcat's performance, and it is therefore not recommended during production.

- The last alternative is to use the functions of Tomcat's Manager Web application in order to reboot only the Axis Web application. As access to the Manager application is restricted, this solution requires prior configuration of a user with the special "manager" role, in the Tomcat user directory (`conf/tomcat-users.xml` by default).

- To restart the Axis Web application using the Manager application, send a request to the following URL: http://localhost:8080/manager/reload?path=/axis. This is the best solution, as it is suitable both for development and use in production.

Once deployment is complete, the Web service is ready to accept client requests. To check whether the service has been properly deployed, we can try and retrieve its WSDL description. As for JWS services, Axis can supply on request the WSDL description of services deployed via a special URL, which is the service's URL ("SOAP end point", followed by request chain " ?WSDL"). For our service, the URL is as follows:
http://localhost:8080/axis/services/InterestCalculatorService?WSDL

4.2.2.3 Client development

To develop an Axis client to access our Web service, two approaches are possible:

- **Static invocation of the Web service:**

  This is the approach that we chose when developing our client to access our first JWS Web service. This involves generating a specific JAX-RPC client framework for the service from the service's WSDL description. Generation of the different client classes and interfaces is carried out automatically using the WSDL2Java tool (WSDL to Java).

- **Dynamic invocation of the Web service**:

Dynamic invocation of services involves using the functions of the Axis APIs (mainly via the standard JAX-RPC interfaces) in order to build the objects and specify the properties required for calling a particular Web service, and to retrieve the result of this call. With this approach, no automatic generation function is available, which means that considerably more time must be spent on development.

As we have already looked at using mechanisms for automatic generation of client frameworks for dynamic invocation of services, for this service we will opt to illustrate development of a client that dynamically invokes a Web service:

Client class for accessing the service by dynamic invocation (TestService.java):

```
import org.apache.axis.AxisFault;
import org.apache.axis.client.Call;
import org.apache.axis.client.Service;
import javax.xml.rpc.namespace.QName;

import java.math.BigDecimal;
import services.intcalc.YearlyReturns;

public class TestService
{
  public static void main(String[] args)
  {
    String SOAP_ENDPOINT = "http://localhost:8080/axis/services/InterestCalculatorService";
    String METHOD        = "projectEarnings";
    String ACTION_URI    = "";
    String TYPEMAPPINGS_URN      = "urn:InterestCalculatorServiceTypes";
    QName  YEARLYRESULTS_QN      = new QName(TYPEMAPPINGS_URN, "YearlyReturns");
    QName  YEARLYRESULTSARRAY_QN = new QName(TYPEMAPPINGS_URN, "ArrayOfYearlyReturns");

    BigDecimal principal = null;
    BigDecimal rate      = null;
    Integer    term      = new Integer(0);

    if(args.length < 3)
    {
      System.out.println("Required params: principal, rate, term");
      System.exit(0);
    }
    try
    {
      principal = new BigDecimal(args[0]);
      rate      = new BigDecimal(args[1]);
      term      = Integer.valueOf(args[2]);
    }
    catch(NumberFormatException nfe)
    {
      System.out.println("Invalid argument: " + nfe);
      System.exit(1);
    }
    try
    {
      Service  service = new Service();
      Call     call    = (Call) service.createCall();
```

```
    call.setEncodingStyle(org.apache.axis.Constants.URI_SOAP_ENC);
    call.setTargetEndpointAddress( new java.net.URL(SOAP_ENDPOINT) );
    call.addParameter("in0",
                      org.apache.axis.Constants.XSD_DECIMAL,
                      javax.xml.rpc.ParameterMode.IN);
    call.addParameter("in1",
                      org.apache.axis.Constants.XSD_DECIMAL,
                      javax.xml.rpc.ParameterMode.IN);
    call.addParameter("in2",
                      org.apache.axis.Constants.XSD_INT,
                      javax.xml.rpc.ParameterMode.IN);
    call.setReturnType(YEARLYRESULTSARRAY_QN);
    call.registerTypeMapping(YearlyReturns.class,
                             YEARLYRESULTS_QN,
                             org.apache.axis.encoding.ser.BeanSerializerFactory.class,
                             org.apache.axis.encoding.ser.BeanDeserializerFactory.class,
                             false);
    call.registerTypeMapping(YearlyReturns[].class,
                             YEARLYRESULTSARRAY_QN,
                             org.apache.axis.encoding.ser.ArraySerializerFactory.class,
                             org.apache.axis.encoding.ser.ArrayDeserializerFactory.class,
                             false);
    call.setUseSOAPAction(true);
    call.setSOAPActionURI(ACTION_URI);
    call.setOperationStyle("rpc");
    call.setOperationName(new QName(ACTION_URI, METHOD));

    Object[] inputParams = new Object[] { principal, rate, term };

    YearlyReturns[] results = (YearlyReturns[]) call.invoke(inputParams);

    dumpResults(principal, rate, term, results);
  }
  catch(AxisFault af)
  {
     System.err.println(af.dumpToString());
  }
  catch(Exception e)
  {
    System.err.println("Exception caught: " + e);
  }
 }
 public static void dumpResults(BigDecimal principal, BigDecimal rate,
                                Integer term, YearlyReturns[] results)
 {
  System.out.println("Results: (year, start principal, yields, end principal)");
  System.out.println("-----------------------------------------------------------");
  for(int i = 0; i < results.length; i++)
  {
    System.out.print("* Year #" + (i+1) + ": ");
    System.out.print(results[i].startPrincipal + ", ");
    System.out.print(results[i].yearlyEarnings + ", ");
    System.out.println(results[i].endPrincipal);
  }
 }
}
```

In the client code, we essentially use two JAX-RPC interfaces: the Service interface and the *Call* interface which encapsulates the call to a service.

To build the call to the Web service, we must define a certain number of properties for the Call object:
the SOAP end-point for the service, the encoding style used, the name of the method to be invoked, and so on.

We must also specify the name and type of each In parameter for the service method, as well as the type of value returned by the service.

If the service manipulates complex types (as is the case here), then we must also declare the required type mappings, as we did on the server side via the deployment descriptor.
Lastly, we invoke the service by passing it the values of the call parameters, and retrieve the return value sent back by the service. Invocation of the service may generate exceptions (application or technical errors) which will if necessary be returned via a SOAP envelope and wrapped in the Java exception *AxisFault*.

As we can see, using Axis APIs for dynamic service invocation creates much more complex code than for static invocation.

However, it is possible to simplify this code to an extent, and in particular to deport registration of the client code type mappings to the Axis client configuration.

The Axis client and Axis server both possess the same architecture, so we can also define a configuration file for the Axis client which may contain instructions for deployment of handlers, transports, and definitions of type mappings.

By default, there is no configuration file for the Axis client. However, it is possible to use the Axis administration tool (in a slightly different form from that which we used for server-side deployment), in order to modify the default configuration of the client. The client-side configuration changes will then be saved in a "`client-config.wsdd`" file which will be automatically created in the current directory from which the administration tool was invoked. If you then wish the client programs to use this configuration, you will need to launch them from the same directory.

In the client code, it is therefore possible to dispense with the two method calls for registering the type mappings shown below, and to deport configuration of these mappings to a client deployment file:

```
call.registerTypeMapping(YearlyReturns.class,
                              YEARLYRESULTS_QN,
                              org.apache.axis.encoding.ser.BeanSerializerFactory.class,
                              org.apache.axis.encoding.ser.BeanDeserializerFactory.class,
                              false);
     call.registerTypeMapping(YearlyReturns[].class,
                              YEARLYRESULTSARRAY_QN,
                              org.apache.axis.encoding.ser.ArraySerializerFactory.class,
                              org.apache.axis.encoding.ser.ArrayDeserializerFactory.class,
                              false);
```

Client-side deployment file (`intcalc_client_deploy.wsdd`) for registration of type mappings:

The client deployment file for registration of type mappings will have the following syntax:

```
<deployment xmlns="http://xml.apache.org/axis/wsdd/"
            xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">
  <beanMapping xmlns:ns1="urn:InterestCalculatorServiceTypes"
       qname="ns1:YearlyReturns"
       languageSpecificType="java:services.intcalc.YearlyReturns" />
  <typeMapping xmlns:ns1="urn:InterestCalculatorServiceTypes"
       qname="ns1:ArrayOfYearlyReturns"
       type="java:services.intcalc.YearlyReturns[]"
       serializer="org.apache.axis.encoding.ser.ArraySerializerFactory"
       deserializer="org.apache.axis.encoding.ser.ArrayDeserializerFactory"
       encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
</deployment>
```

Here we simply extracted the type mappings defined in the service deployment descriptor and recopied them to the client deployment descriptor.

Deploying the type mappings in the Axis client configuration:

For client-side administration, we use the "`org.apache.axis.utils.Admin`" class instead of the "`org.apache.axis.client.AdminClient`" used previously on the server side.
This "`Admin`" utility is more general than "AdminClient" and may be used both on the client and server sides; the utility takes an additional argument as a parameter indicating the configuration (server or client) on which modifications must be carried out. The value of the first parameter passed to "Admin" is therefore either "server," or "client."

```
java org.apache.axis.utils.Admin client intcalc_client_deploy.wsdd
```

4.2.2.4 Testing the server using the Axis client:

Once the client has been compiled we can test our service access client. Here is the result of the interest calculation for an initial capital of €10,000 invested at 5.5% over a period of 5 years:

```
> java TestService 10000 5.5 5
Projection Results: (Year, start principal, yields, end principal)
---------------------------------------------------------------
* Year #1: 10000.00, 550.00, 10550.00
* Year #2: 10550.00, 580.25, 11130.25
* Year #3: 11130.25, 612.16, 11742.41
* Year #4: 11742.41, 645.83, 12388.24
* Year #5: 12388.24, 681.35, 13069.59
```

4.2.2.5 Monitoring RPC traffic using the TCPMon utility

The Apache SOAP framework included a graphic Java utility for monitoring TCP traffic, called "TCPTunnelGUI" which enabled the RPC messages exchanged by the client and server to be viewed.

Fortunately, although it has changed name (it's now called "TCPMon"), this utility is also included in the Axis distribution (`org.apache.axis.utils.tcpmon` class).

The operating principle of this utility is simple: it listens at a TCP port on the local machine, and transfers any connection to this port via a target TCP port on a given machine (local or remote). Responses from the TCP port of the target machine are also relayed in the opposite direction to the source TCP port. The utility saves the request/response sequences which transit through it and displays them on screen. It therefore enables you to view the RPC exchanges between a Web service and client.

As an example, below we describe how to use TCPMon to view the messages transiting between the Axis client and server during the call to our interest calculator service.

First of all, we must edit the client source code in order to replace the TCP port used by Axis, in the service's URL, with the TCP port at which TCPMon is to listen. Our Axis server hosted on Tomcat uses port 8080 and we will use port 8081 as the listening port for TCPMon.

In the `TestService.java` file, we replace port number "8080," below, by "8081", and recompile the file.

```
    String SOAP_ENDPOINT = "http://localhost:8080/axis/services/InterestCalculatorService";
```

Then, we launch the TCPMon utility, passing it the following arguments:

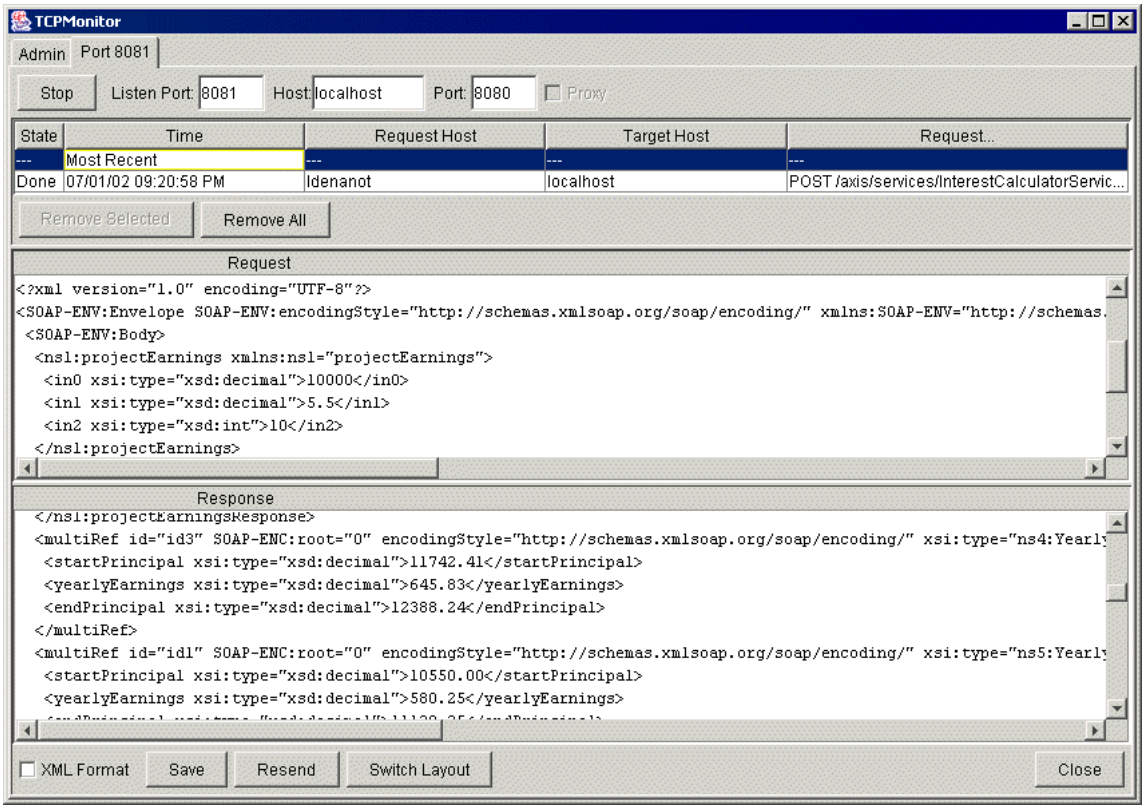
- TCP listening port: 8081
- Target machine name : as the Axis server and client are on the same machine, we specify "localhost" as the name of the target machine.
- TCP target port: 8080

```
> java org.apache.axis.utils.tcpmon 8081 localhost 8080
```

Then, we launch our client to call the interest calculator Web service:

```
> java TestService 10000 5.5 10
```

While the client calls the service and receives the response, the SOAP request and response messages exchanged between the client and server appear in the text "Request" and "Response" zones in the TCPMon window:



4.2.2.6 Undeploying a service deployed on the Axis server

To remove (undeploy) a service deployed on the Axis server, you must, as during deployment, create a WSDD file. This undeployment descriptor simply indicates the name of the service to be removed in the element `<undeployment>` :

Service undeployment descriptor (intcalc_undeploy.wsdd file):

```
<undeployment xmlns="http://xml.apache.org/axis/wsdd/">
  <service name="InterestCalculatorService"/>
</undeployment>
```

To undeploy the service, we launch the Axis administration client, passing it the service undeployment descriptor as an argument:

```
java org.apache.axis.client.AdminClient intcalc_undeploy.wsdd
```

The administration utility removes the corresponding entry for the service from the server configuration file (server-config.wsdd), but of course does not remove the service classes copied into the tree of the Axis Web application.

4.2.3 Developing a Web service from a Java interface

Axis' automatic code generation tools, based on the concepts of JAX-RPC, make development a much easier task, not only on the client side, but also on the server side.

In fact, thanks to Axis' Java2WSDL and WSDL2Java tools, not only can you create a client framework for accessing a service, but you can also create a server framework for creating and deploying a Web service via the JAX-RPC model.

In this development model, we work from a Java interface which represents the interface of the Web service to be created, with its different methods. From this interface, we will perform the following operations:

- Automatic generation of a WSDL description equivalent to this Java interface, using the Java2WSDL tool.
- Automatic generation of JAX-RPC client and server frameworks from the WSDL description generated.
- Use of the server framework generated to create and deploy the service on Axis.
- Use of the client framework generated to develop the client for accessing the service.

To illustrate this approach, we shall develop a new service for generating random numbers, called "RandomNumberGeneratorService".

This service will contain a method called "getRandomNumber". This method will take an integer as a parameter and will return a random number between 1 and the integer passed as a parameter.

Creation of Java interface for our service (`services.rng.RandomNumberGenerator`) :

We must create the Java interface containing the signature for the method that will be exposed by our service:

```
package services.rng;

public interface RandomNumberGenerator
{
  public long getRandomNumber(long upperBound);
}
```

Generation of corresponding WSDL description for this interface:

To generate a WSDL description equivalent to our Java interface, we will use the Java2WSDL tool, passing it the following arguments:

- The name of the WSDL file to be output (RandomNumberGenerator.wsdl)
- The URL which will serve as the access point for our service: "http://localhost:8080/axis/services/RandomNumberGeneratorService "

- The full name of the Java interface of our service ("`services.rng.RandomNumberGenerator`")

The command to run is as follows (to be entered on a single line):

```
java org.apache.axis.wsdl.Java2WSDL
 -o RandomNumberGenerator.wsdl
 -l http://localhost:8080/axis/services/RandomNumberGeneratorService
 services.rng.RandomNumberGenerator
```

Generation of interfaces and classes for implementation of service and client:

Once the WSDL file has been generated, we shall use the WSDL2Java tool to generate the different Java classes and interfaces for the service and client, from this WSDL file. Also, WSDL2Java will automatically generate the deployment and undeployment descriptors for the service.

As with Java2WSDL, we will also have to pass a certain number of arguments to WSDL2Java:

- the "`–o .`" option tells the tool where to generate the class tree (in the current directory ".")
- the "`–s .`" option tells the tool to generate the server-side classes for the service in addition to the client-side classes
- the "`–d Application`" option indicates the "scope" to be used in the deployment descriptor for instantiation of the service (here, choosing the "Application" value will entail the creation of a single instance of the service to handle all requests to this service).
- the "`-p services.rng.impl`" option tells the tool that the set of classes generated should be placed in the Java package "`services.rng.impl `" (it is best to use a different package from that of the original interface, as the Java2WSDL recreates an interface with the same name)
- lastly, you must indicate the name of the WSDL interface describing our service: "`RandomNumberGenerator.wsdl`"

Once again, the command to execute must be entered on a single line:

```
java org.apache.axis.wsdl.WSDL2Java
  -o . –s
  -d Application
  -p services.rng.impl
  RandomNumberGenerator.wsdl
```

The Java2WSDL tool then generates various Java interfaces and classes in compliance with the JAX-RPC specification. All the classes generated belong to the Java package specified (`services.intcalc.impl`):

- The different classes and interfaces corresponding to the service and the complex types used:

- `RandomNumberGenerator`: The interface for our service, which comprises the different operations (methods exposed by the service) defined by the service.
- `RandomNumberGeneratorService`: The interface for the service locator class of our service.
- `RandomNumberGeneratorServiceLocator`: The service locator for our service

- The client proxy class (stub) for calling the service:

- `RandomNumberGeneratorSoapBindingStub`

- The implementation class for our service on the server side:

- `RandomNumberGeneratorServiceSoapBindingImpl`

In addition to these classes and interfaces, WSDL2Java also generates deployment (`deploy.wsdd`) and undeployment (`undeploy.wsdd`) descriptors for our service in Axis.

Adding the processing logic for our service to the server implementation class:

Clearly, the implementation class for our service generated by WSDL2Java (`services.intcalc.impl.InterestCalculatorServiceSoapBindingImpl`)does not contain any business code, so we must therefore insert the processing logic for our service (modifications shown in **boldtype**):

```
package services.rng.impl;

public class RandomNumberGeneratorServiceSoapBindingImpl
implements services.rng.impl.RandomNumberGenerator
{
  public long getRandomNumber(long in0) throws java.rmi.RemoteException
  {
    // return -3;
    return Math.round( Math.random() * (in0 - 1) + 1);
  }
}
```

Deploying the service on Axis

To deploy the service on Axis, the administration client must be executed, by passing to it as a parameter the deployment descriptor (`deploy.wsdd`) created by the WSDL2Java tool:

```
java org.apache.axis.client.AdminClient deploy.wsdd
```

Before developing the client to test the service, we must also remember to compile the classes for our service, copy them to the class tree of the Axis Web application, and

restart the Tomcat server if necessary in order for the modifications to be taken into account.

Developing the client (TestService.java):

The development of the client class for our service is very simple here, as we use the class framework generated by Axis:

```
import services.rng.impl.*;
import org.apache.axis.AxisFault;

public class TestService
{
  public static void main(String[] args)
  {
    if(args.length < 1)
    {
      System.out.println("Required params: upper bound (long integer)");
      System.exit(0);
    }
    long upperBound = Long.valueOf(args[0]).longValue();

    try
    {
      RandomNumberGeneratorService service =
         new RandomNumberGeneratorServiceLocator();
      RandomNumberGenerator port    = service.getRandomNumberGeneratorService();

      long result = port.getRandomNumber(upperBound);
      System.out.println("Result = " + result);
    }
    catch(AxisFault af)
    {
       System.err.println(af.dumpToString());
    }
    catch(java.lang.Exception e)
    {
    System.out.println("Exception caught: " + e);
    }
  }
}
```

Lastly, to test our client:

```
> java TestService 6
Result = 3
```

4.3 Evaluation

Through these few examples, we can see that Axis offers some powerful features, especially the automatic code generation, which greatly simplifies developments.

However, the JAX-RPC development model remains somewhat complex to use on the server side, and in terms of productivity, it is probably more worthwhile developing services as simple Java classes than using the functions of JAX-RPC for server-side code generation from a Java interface.

Conversely, on the client side, we certainly gain by using the client proxy class generation functions, which considerably simplify development of clients for accessing Web services. This static service invocation model therefore seems greatly superior (in terms of enhanced productivity and simplicity of use) to the classic dynamic invocation model of JAX-RPC.

Meanwhile, despite its attraction, the JWS Web service deployment model currently appears to be more of a gadget than a veritable development approach at present.

5 Conclusion

Compared to Apache SOAP, Axis can truly be seen as a revolution, which may bring the development of Web Services into a new era.

Firstly, its modular architecture makes a new development approach possible, based on the use of reusable components to extend the functionality of the server.

With an entirely overhauled development model, Axis brings a simplicity to development that was unknown with Apache SOAP, and offers greatly enhanced productivity, particularly due to the client and server framework generation tools for creating and calling Web Services.

However, despite these technological advances brought by Axis, there is still much work to be done, and the truth is that we are still some way away from seeing a "finalized, stabilized" version of the framework.

Many of Axis' features are still in the development phase, and the available documentation is still too sparse. Consequently, if one wishes to find answers to questions not covered in the documentation, the only options are to rely on the Axis development mailing-lists (addresses available at http://xml.apache.org/axis/) or to closely examine the source code.

With regard to standards compliance, considerable improvements have been made, particularly in terms of WSDL support, and the care taken to conform to the emerging Java standards for Web services (JAX-RPC and JAXM). Real progress has also been made in terms of SOAP support (full support for SOAP 1.1 and partial support for SOAP 1.2), and clear improvements also seem to have been made with regard to interoperability with other SOAP implementations.

Lastly, when it comes to the actual Web services standards, there is still much progress to be made. In particular, the security problems are not always resolved satisfactorily, and the standards (which are still very young), are not always stabilized. We are still awaiting the final version of the SOAP 1.2 specification from the W3C, and the definition of its future replacement, XP (XML Protocol). Likewise, the new Java standards for Web services (within the Java XML Pack), which have only recently appeared, are still in the early stages of their development.

Axis does, however, seem to have a promising future ahead of it, and is likely to be at least as successful as its predecessor Apache SOAP, if not more so. Remember that Apache SOAP was adopted by major vendors such as IBM and iPlanet (Sun), and it therefore seems likely that Axis will in the future become the Web Services infrastructure used by many J2EE application servers.

A white paper by

TechMetrix Research

Author
Laurent DENANOT

USA

TechMetrix Research
76 Bedford Street, suite 33
Lexington, MA 02420

Tel.: +1.781.890.3900
Fax: +1.781.240.0502

EUROPEAN HEADQUARTERS

TechMetrix Research/SQLI
55/57 Rue Saint Roch
75001 PARIS
FRANCE

Tel.: + 33 1 44 55 40 00
Fax: + 33 1 44 55 40 01

SWITZERLAND

TechMetrix Research/SQLI
Chemin de la Rueyre
116-118
CH-1020 RENENS

Tel.: + 41 (0) 21 637 72 30
Fax : + 41 (0) 21 637 72 31

http://www.techmetrix.com
info@techmetrix.com

TechMetrix Research is a technically oriented analyst firm focused on e-business application development needs. TechMetrix Research has developed a unique evaluation approach to provide accurate information on software. Based in Lexington-MA (USA) and Paris (France), the firm publishes comparison reports and product reviews, which are real helpers when it comes to making decisions, or simply keeping pace with the fast moving e-business market.
As its parent company (SQLI) provides information system development and implementation services to major companies, TechMetrix Research also benefits from the feedback and experience acquired during large-scale, long-term development projects.
SQLI is a European global system integrator of 700 employees offering full service and continuing coaching to enable companies to move profitably toward an all-Internet solution.