Web Services & Axis2

Architecture & Tutorial

Ing. Buda Claudio
claudio.buda@unibo.it
2nd Engineering Faculty
University of Bologna

June 2007

Axis from SOAP

Apache Axis is an implementation of the SOAP ("Simple Object Access Protocol") submission to W3C.

Extract from the draft W3C specification:

"SOAP is a lightweight protocol for exchanging structured information in a decentralized, distributed environment. It is an XML based protocol that consists of three parts: an envelope that defines a framework for describing what is in a message and how to process it, a set of encoding rules for expressing instances of application-defined datatypes, and a convention for representing remote procedure calls and responses."

Axis2

A new architecture for Axis was introduced during the August 2004 Summit in Colombo, Sri Lanka. The new architecture on which Axis2 is based on is more flexible, efficient and configurable in comparison to Axis1.x architecture. Some well established concepts from Axis 1.x, like handlers etc., have been preserved in the new architecture.

The Apache Axis2 project is a Java-based implementation of both the client and server sides of the Web services equation. Designed to take advantage of the lessons learned from Apache Axis 1.0, Apache Axis2 provides a complete object model and a modular architecture that makes it easy to add functionality and support for new Web services-related specifications and recommendations.

Axis2 abilities

Axis2 enables you to easily perform the following tasks:

- Send SOAP messages
- Receive and process SOAP messages
- Create a Web service out of a plain Java class
- Create implementation classes for both the server and client using WSDL
- Easily retrieve the WSDL for a service
- Send and receive SOAP messages with attachments
- Create or utilize a REST-based Web service
- Create or utilize services that take advantage of the WS-Security, WS-ReliableMessaging, WS-Addressing, WS-Coordination, and WS-Atomic Transaction recommendations
- Use Axis2's modular structure to easily add support for new recommendations as they emerge

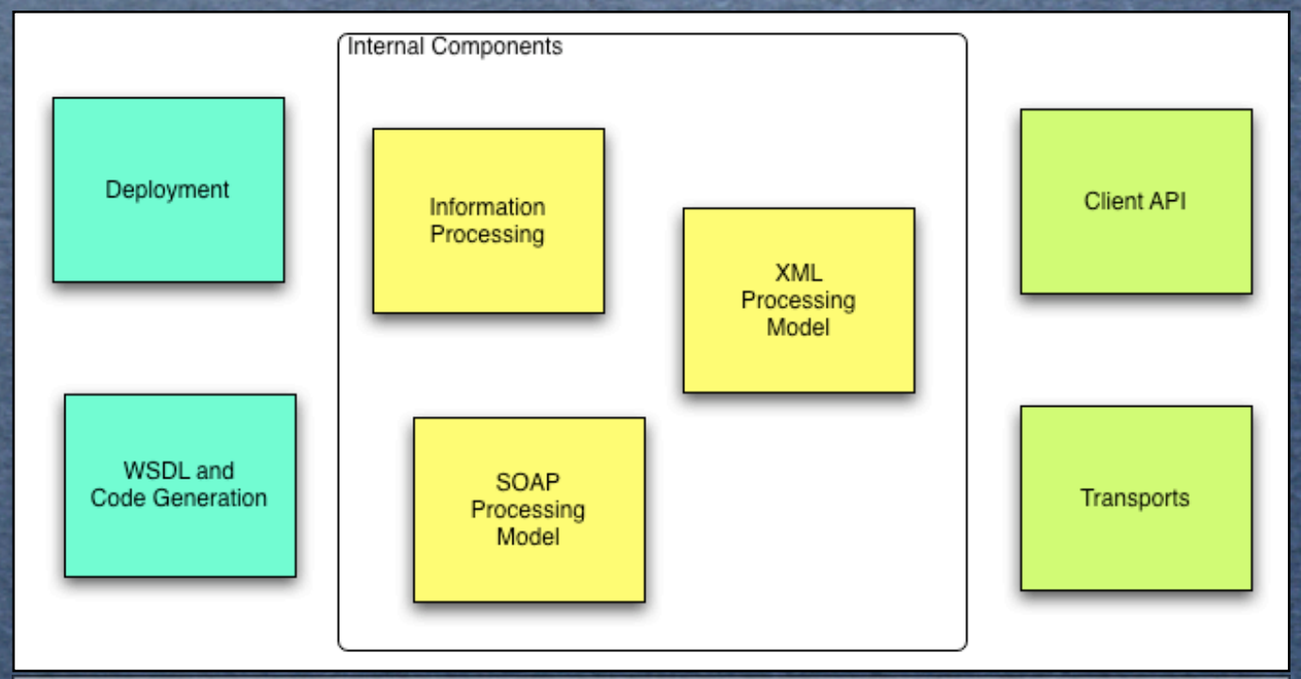
Axis2 Architecture

Core Modules

1. Information Model
2. XML processing Model
3. Soap Processing Model
4. Deployment Model
5. Client API
6. Transports

Other Modules

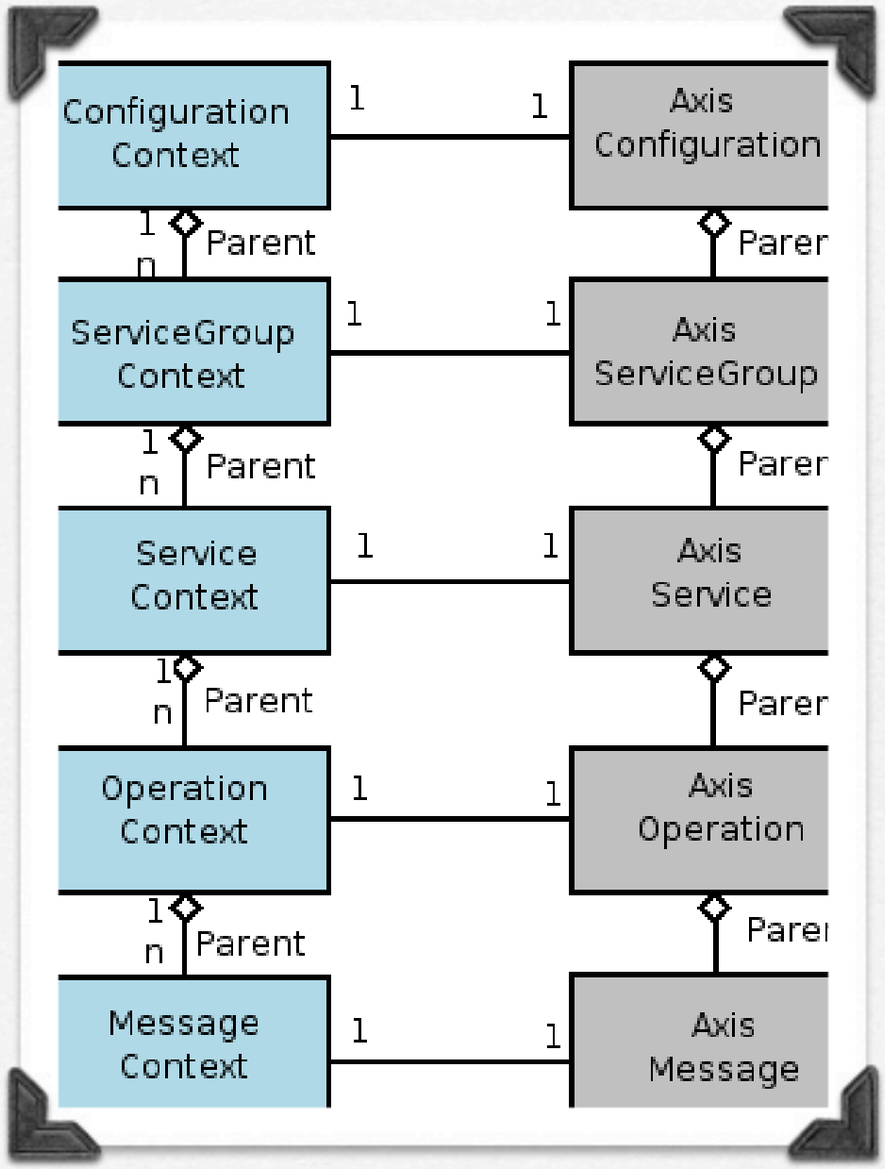
7. Code Generation
8. DataBinding



1. Information Model

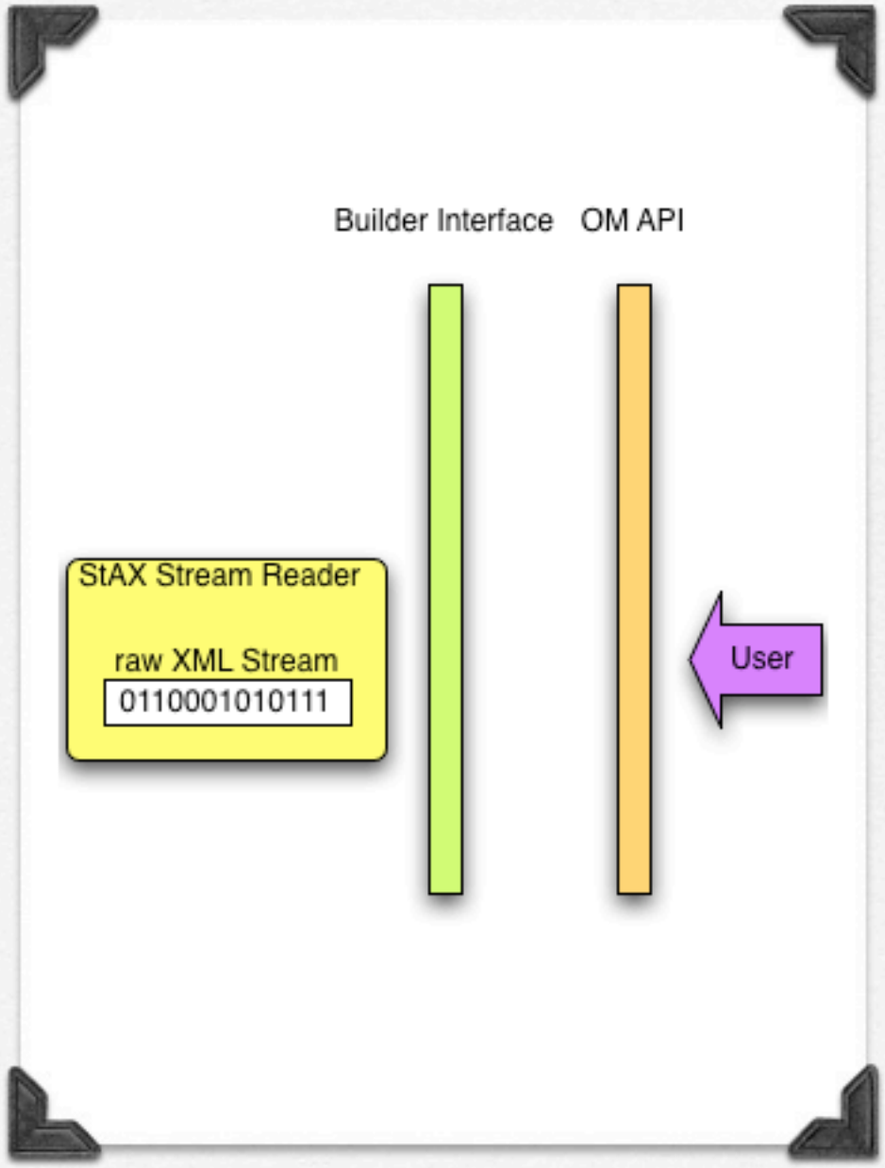
The two hierarchies are connected as shown in the above figure. The Description hierarchy represents the static data. This data may be loaded from a configuration file that exists throughout the lifetime of Axis2. For example, deployed Web services, operations, etc. On the other hand, the context hierarchy holds more dynamic information about the things that have more than one instance (e.g.Message Context).

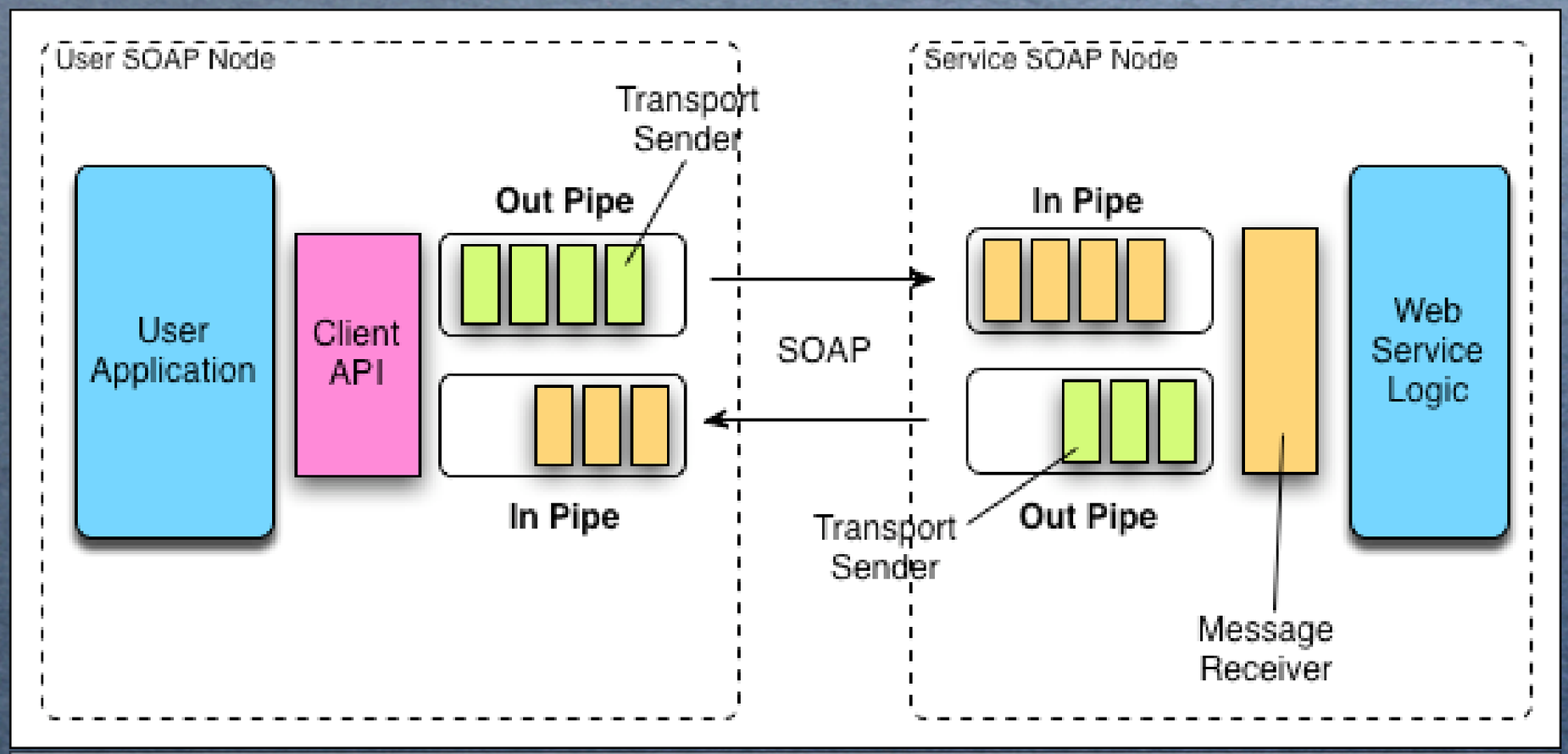
These two hierarchies create a model that provides the ability to search for key value pairs. When the values are searched at a given level, they are searched while moving up the hierarchy until a match is found. In the resulting model, the lower levels override the values in the upper levels. For example, when a value is looked up in the Message Context and is not found, it would be looked up in the Operation Context, etc, up the hierarchy. The Search is first done up the hierarchy, and if the starting point is a Context then it searches in the Description hierarchy as well.



2. XML Processing Model

OM stands for Object Model (also known as AXIOM - AXis Object Model) and refers to the XML infoset model that is initially developed for Apache Axis2. XML infoset refers to the information included inside the XML, and for programmatic manipulation it is convenient to have a representation of this XML infoset in a language specific manner. For an object oriented language the obvious choice is a model made up of objects. DOM and JDOM are two such XML models. OM is conceptually similar to such an XML model by its external behavior but deep down it is very much different. The objective of this tutorial is to introduce the basics of OM and explain the best practices to be followed while using OM. However, before diving in to the deep end of OM it is better to skim the surface and see what it is all about!

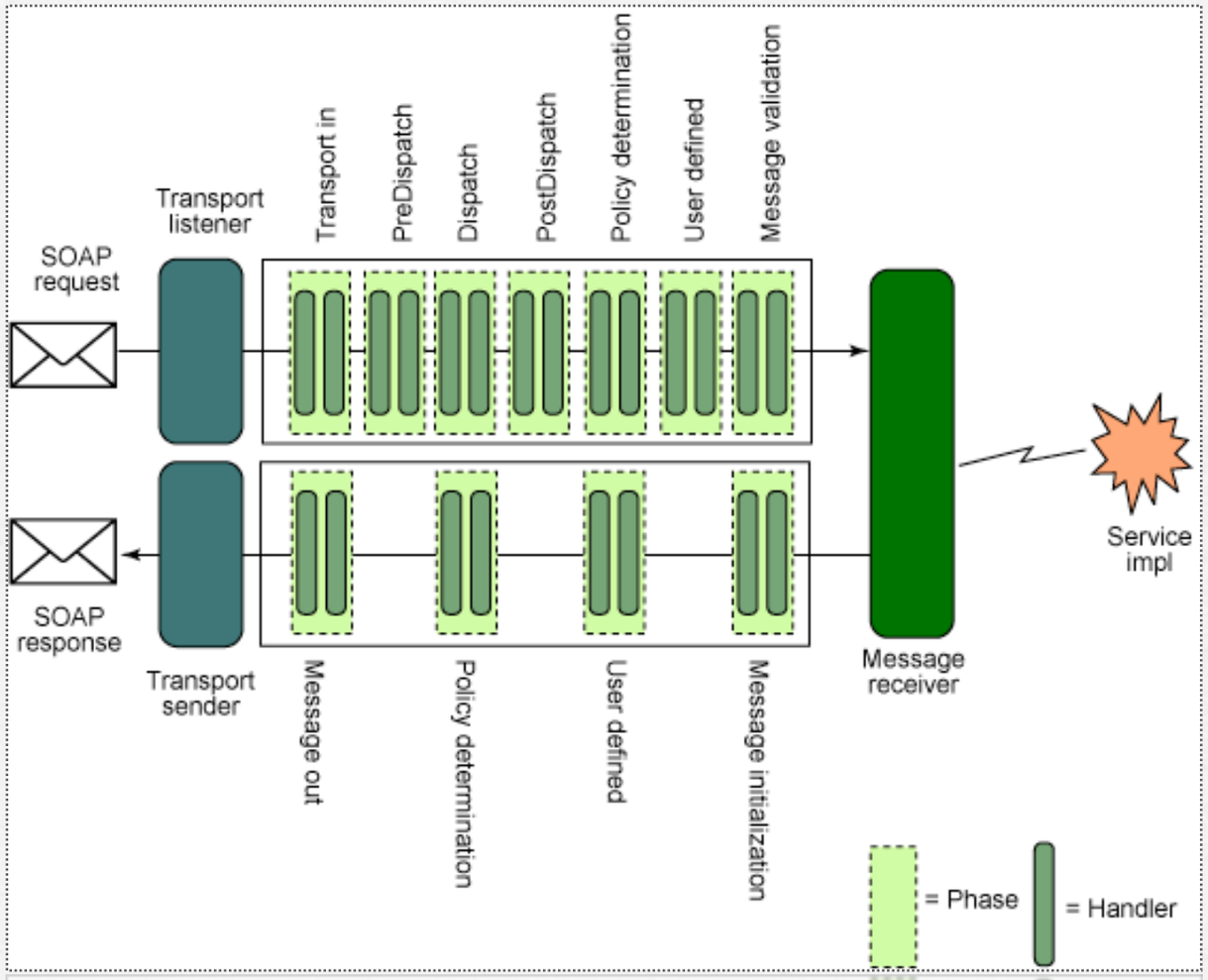




3. Soap Processing Model

InPipe & OutPipe

The architecture identified two basic actions a SOAP processor should perform, sending and receiving SOAP messages. The architecture provides two Pipes ('Flows'), to perform these two basic actions. The Axis Engine or the driver of Axis2 defines two methods send() and receive() to implement these two Pipes. The two pipes are named In Pipe and Out Pipe, and the complex Message Exchange Patterns (MEPs) are constructed by combining these two pipes.



Handlers

Extensibility of the SOAP processing model is provided through handlers. When a SOAP message is being processed, the handlers that are registered will be executed. The handlers act as interceptors and they process parts of the SOAP message and provide add-on services.

When a SOAP message is being sent through the Client API, an Out Pipe would begin, the Out Pipe invokes the handlers and end with a Transport Sender that sends the SOAP message to the target endpoint. The SOAP message is received by a Transport Receiver at the target endpoint, which reads the SOAP message and starts the In Pipe. The In Pipe consists of handlers and ends with the Message Receiver, which consumes the SOAP message.

The processing explained above happens for each and every SOAP message that is exchanged. After processing one message, Axis2 may decide to create other SOAP messages, in which case more complex message patterns emerge.

The two pipes does not differentiate between the Server and the Client. The SOAP Processing Model handles the complexity and provides two abstract pipes to the user. The different areas or the stages of the pipes are given names, and according to Axis2 slang, they are named 'phases'. A Handler always runs inside a phase, and the phase provides a mechanism to specify the ordering of handlers. Both Pipes have built-in phases, and both define the areas for 'User Phases' which can be defined by the user.

Incoming Message Phases

1. Transport Phase - The handlers are in the phase that processes transport specific information such as validating incoming messages by looking at various transport headers, adding data into message context, etc.

2. Pre-Dispatch Phase- The main functionality of the handlers in this phase is to populate message context to do the dispatching. For example, processing of addressing headers of the SOAP message, if any, happens in this phase. Addressing handlers extract information and put them in to the message context.

3. Dispatch Phase - The Dispatchers run in this phase and try to find the correct service and operation this particular message is destined for.
   The post condition of the dispatch phase (any phase can contain a post condition) checks whether a service and an operation were found by the dispatchers. If not, the execution will halt and give a "service not found' error.

4. User Defined Phases - Users can engage their custom handlers here.

5. Message Validation Phase - Once the user level execution has taken place, this phase validates whether SOAP Message Processing has taken place correctly.

6. Message Processing Phase - The Business logic of the SOAP message is executed here. A Message Receiver is registered with each Operation. This message receiver (associated to the particular operation) will be executed as the last handler of this phase.

Outgoing Message Phases

1. Message Initialize Phase - First phase of the Out Pipe. Serves as the placeholder for the custom handlers.

2. User Phases - Executes handlers in user-defined phases.

3. Transports Phase - Executes any transport handlers taken from the associated transport configuration. The last handler would be a transport sender which will send the SOAP message to the target endpoint.

Extending the SOAP Processing Model with Handlers & Modules

Axis2 defines an entity called a 'module' that can introduce handlers and Web service operations. A Module in terms of Axis2 usually acts as a convenient packaging that includes:

- A set of handlers and
- An associated descriptor which includes the phase rules

A service, operation, or the system may engage a module. Once the module is engaged, the handlers and the operations defined in the module are added to the entity that engaged them.

Modules cannot be added (no hot deployment) while the Axis2 engine is running, but they will be available once the system is restarted.

4. Service Deployment Model

The Axis2.xml File
Service Archive
Module Archive

axis2.xml file

This file holds the global configuration for the client and server, and provides the following information:

1. The global parameters
2. Registered transport-in and transport-outs
3. User-defined phase names
4. Modules that are engaged globally (to all services)
5. Globally defined Message Receivers

```
<phaseOrder type="InFlow">
    <!--  System pre defined phases       -->
    <phase name="Transport">
        <handler name="RequestURIBasedDispatcher"
                 class="org.apache.axis2.engine.Req
            <order phase="Transport"/>
        </handler>
        <handler name="SOAPActionBasedDispatcher"
                 class="org.apache.axis2.engine.SOA
            <order phase="Transport"/>
        </handler>
    </phase>
    <phase name="Security"/>
    <phase name="PreDispatch"/>
    <phase name="simpawsPhase"/>
    <phase name="Dispatch" class="org.apache.axis2.
        <handler name="AddressingBasedDispatcher"
                 class="org.apache.axis2.engine.Add
            <order phase="Dispatch"/>
```

Service Archive

The Service archive must have a META-INF/services.xml file and may contain the dependent classes. The services.xml file has the following information.

1. Service level parameters
2. Modules that are engaged at service level
3. Service Specific Message Receivers
4. Operations inside the service

```
<service>

    <description>
        META-WS component
    </description>

    <module ref="simpaws" />

    <parameter name="ServiceClass" locked="false">
        alice.simpaws.metaws.BridgeAgent
    </parameter>

    <operation name="toBridgeAgent">
        <messageReceiver
class="org.apache.axis2.receivers.RawXMLINOutMessag
    </operation>

</service>
```

Module Archive

Module archive must have a META-INF/module.xml file and dependent classes. The module.xml file has Module parameters and the Operations defined in the module.

The deployment model first finds the axis2.xml file and builds the global configuration. Then it checks for the module archives and then for the service archives.

Hot deployment is only allowed for services.

```
<module name="addressing">
    <Description>This is WS-Addressing implementat
    <InFlow>
        <handler name="AddressingFinalInHandler" c
            <order phase="PreDispatch"/>
        </handler>
        <handler name="AddressingSubmissionInHandl
            <order phase="PreDispatch"/>
        </handler>
        <handler name="AddressingWSDLValidationHan
            <order phase="Dispatch" after="Instanc
        </handler>
    </InFlow>

    <OutFlow>
        <!-- We don't need to have two handlers to
        <handler name="AddressingOutHandler" class
            <order phase="MessageOut"/>
        </handler>
    </OutFlow>

    <OutFaultFlow>
        <handler name="AddressingOutHandler" class
            <order phase="MessageOut"/>
        </handler>
    </OutFaultFlow>

    <InFaultFlow>
        <handler name="AddressingFinalInHandler" c
            <order phase="PreDispatch"/>
        </handler>
        <handler name="AddressingSubmissionInHandl
            <order phase="PreDispatch"/>
        </handler>
        <!-- AddressingWSDLValidationHandler not pr
    </InFaultFlow>
</module>
```

5. Client API

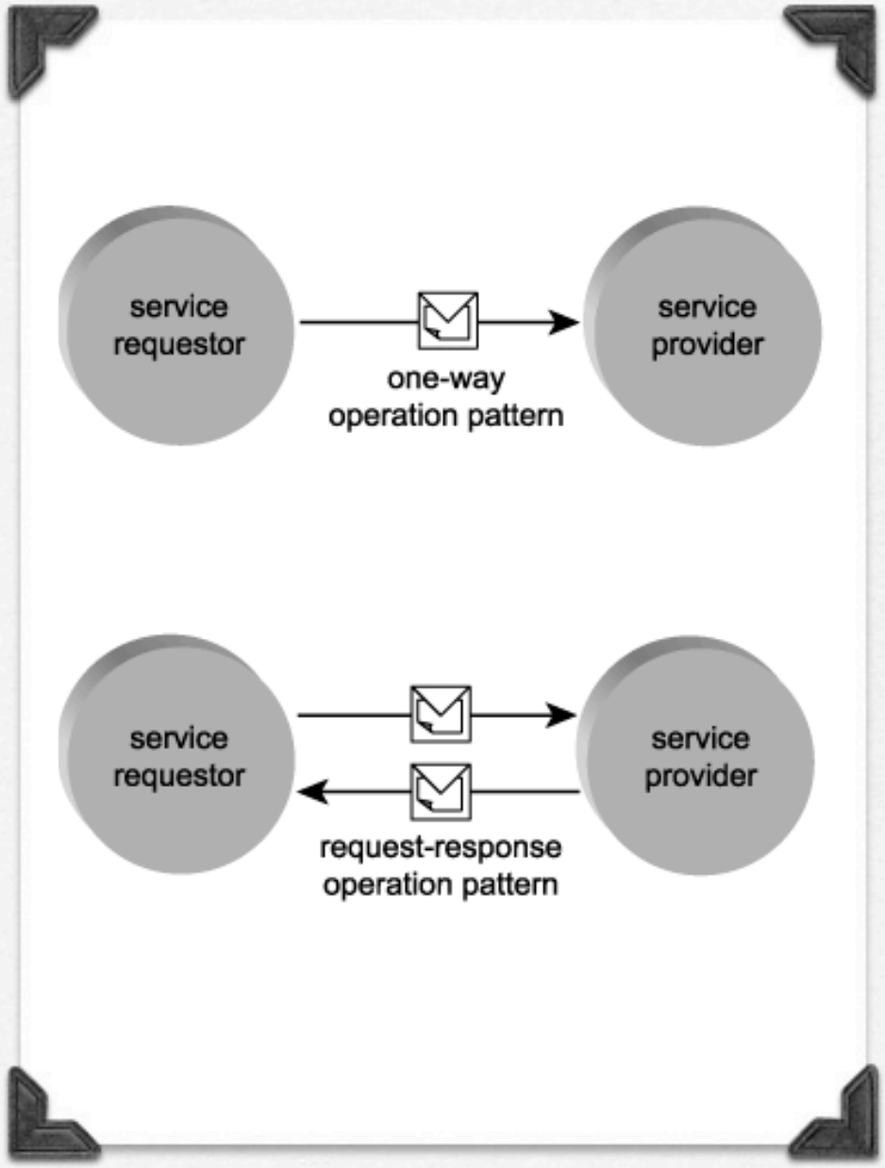
Message Exchange Pattern (MEP)
Transport Behavior (One-Way or Two-Way)
Synchronous / Asynchronous Calling

Message Exchange Pattern (MEP)

Although all SOAP messages carry the same structure, the ways in which they are used can be combined into a number of different "message exchange patterns", or MEPs. The two major message exchange patterns are:

- In-Out: in this MEP, the client sends a SOAP message to the server, which processes the message and sends a response back. This is probably the most commonly used MEP, and is useful for tasks such as searching for information or submitting information in situations in where acknowledgment is important.

- In-Only: In this MEP, the client sends a message to the server without expecting a response. You may use this MEP for activities such as pinging a server to wake it up, reporting logging information for which you do not need an acknowledgment and so on



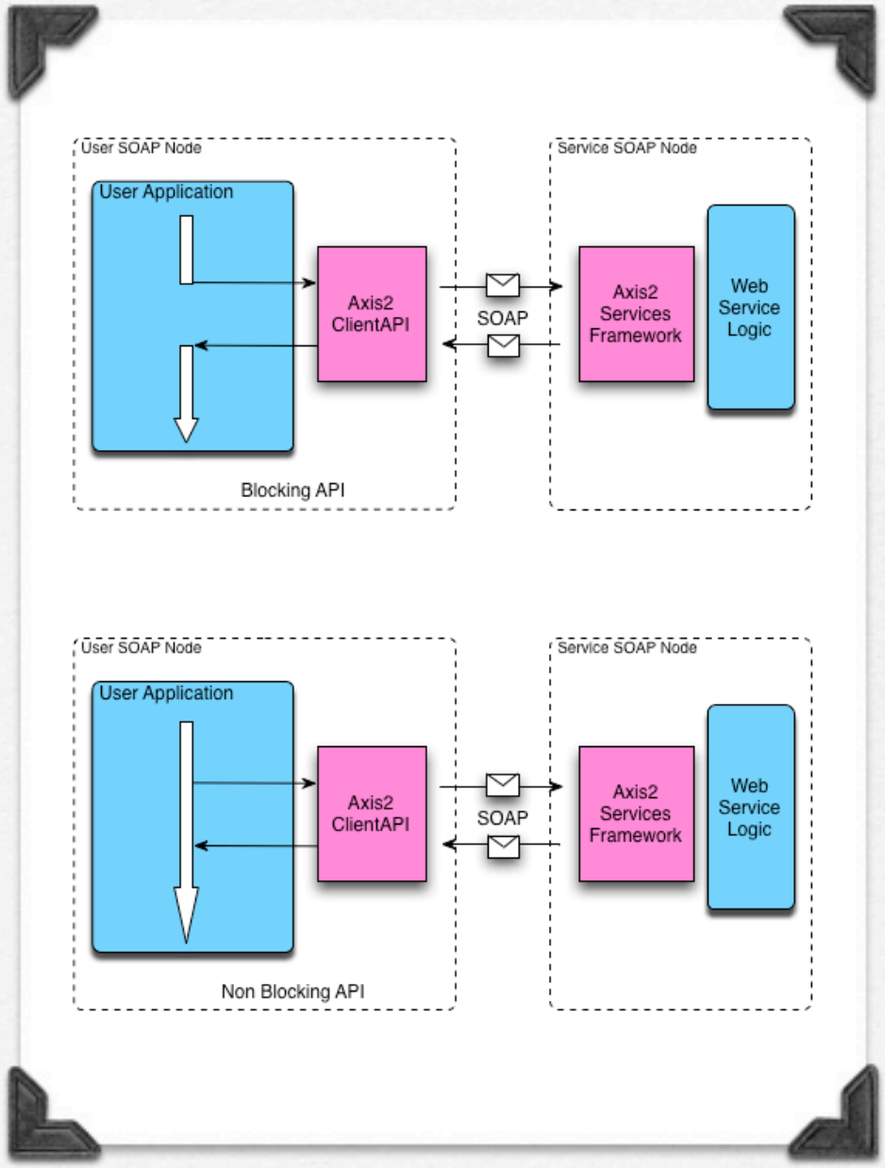
Synchronous / Asynchronous Calling

Many Web service engines provide users with Blocking and Non-Blocking client APIs.

- **Blocking API** - Once the service invocation is called, the client application hangs and only regains control when the operation completes, after which the client receives a response or a fault. This is the simplest way of invoking Web services, and it also suites many business situations.

- **Non-Blocking API** - This is a callback or polling based API. Hence once a service invocation is called, the client application immediately regains control and the response is retrieved using the callback object provided. This approach provides flexibility to the client application to invoke several Web services simultaneously without blocking the operation already invoked.

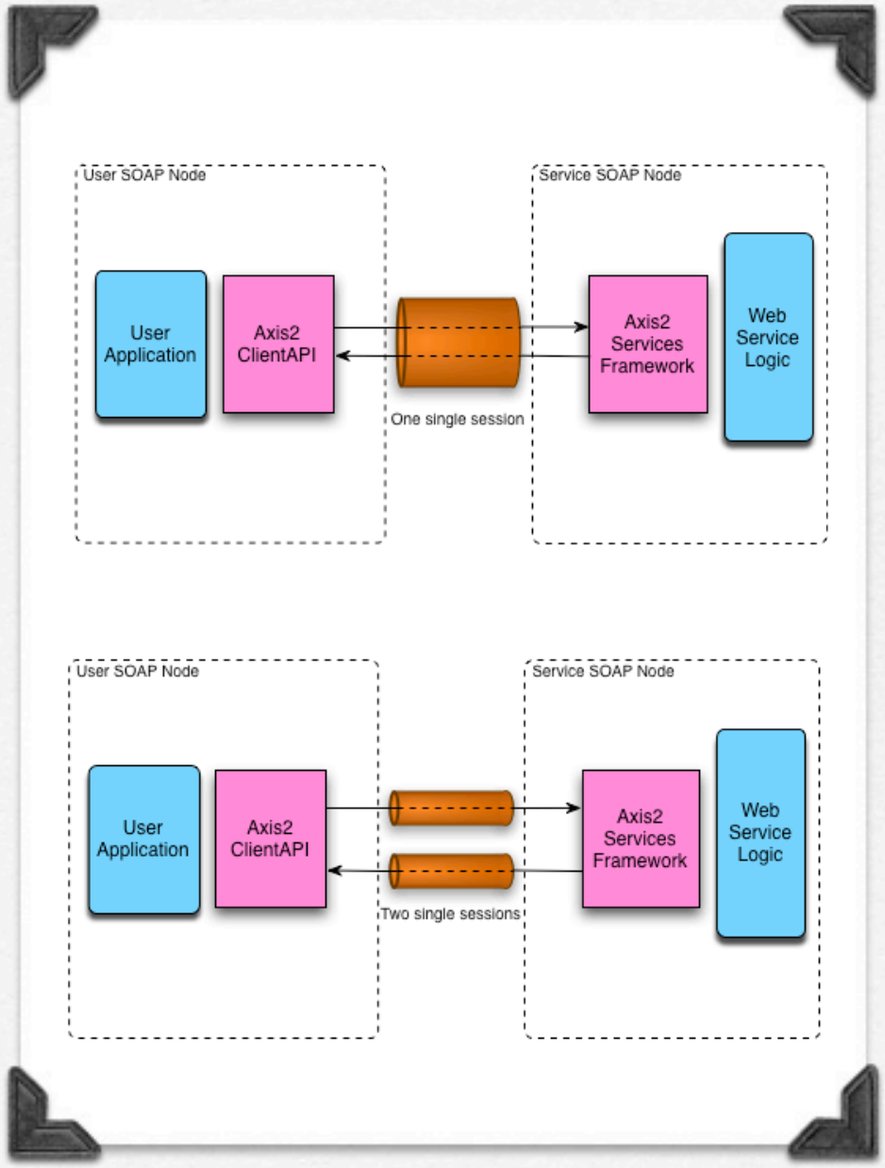
Both mechanisms work at the API level. Let's name the asynchronous behavior that we can get using the Non-Blocking API as API Level Asynchrony.



Transport Behavior (One-Way or Two-Way)

Both mechanisms, Blocking API and Non-Blocking API, use single transport connections to send the request and to receive the response. They severely lag the capability of using two transport connections for the request and the response (either One-Way or Two-Way). So both these mechanisms fail to address the problem of long running transactions (the transport connection may time-out before the operation completes).

A possible solution would be to use two separate transport connections for request and response. The asynchronous behavior that we gain using this solution can be called Transport Level Asynchrony.



API Level Asynchrony and Transport Level Asynchrony

By combining API Level Asynchrony and Transport Level Asynchrony, we can obtain four different invocation patterns for Web services as shown in the following table.

API (Blocking/Non-Blocking)	Dual Transports (Yes/No)	Description
Blocking	No	The simplest and more familiar invocation pattern
Non-Blocking	No	Using callbacks or polling
Blocking	Yes	This is useful when the service operation is IN-OUT in nature but the transport used is One-Way (e.g. SMTP)
Non-Blocking	Yes	This is can be used to gain the maximum asynchronous behavior. Non blocking at the API level and also at the transport level.

6. Transports

The incoming transport is the transport via which the AxisEngine receives the message. The outgoing transport is decided based on the addressing information (wsa:ReplyTo and wsa:FaultTo). If addressing information is not available and if the server is trying to respond, then the out going transport will be the outputstream of the incoming transport (if it is two-way transport).

At the client side, the user is free to specify the transport to be used.

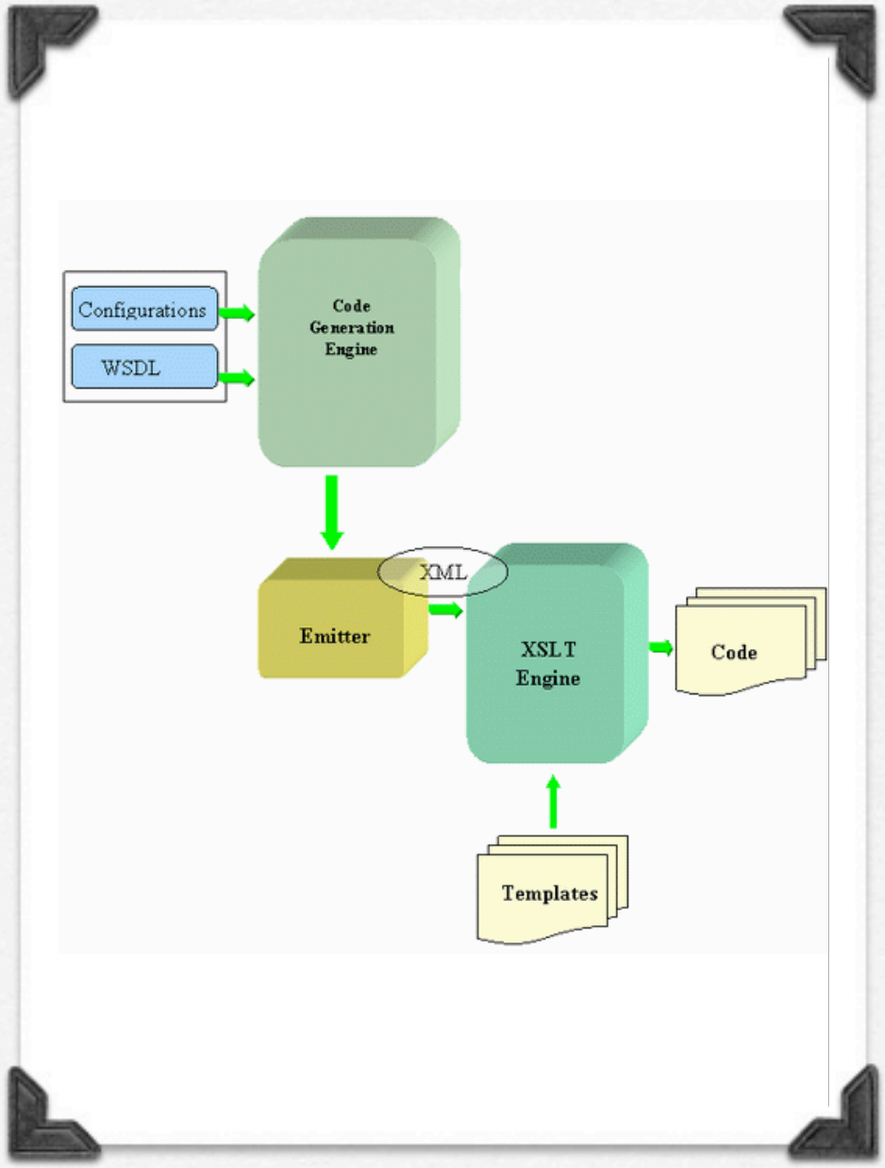
Axis2 presently supports the following transports:

- HTTP - In HTTP transport, the transport listener is a servlet or org.apache.axis2.transport.http.SimpleHTTPServer provided by Axis2. The transport sender uses commons-httpclient to connect and send the SOAP message.
- TCP - This is the simplest transport, but needs the WS - Addressing support to be functional.
- SMTP - This works off a single email account. Transport receiver is a thread that checks for emails in fixed time intervals.
- JMS - The Java Message Service API is a Java Message Oriented Middleware (MOM) API for sending messages between two or more clients.

7. Code Generation

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

The basic approach is to set the code generator to generate an XML, and parse it with a template to generate the code file.

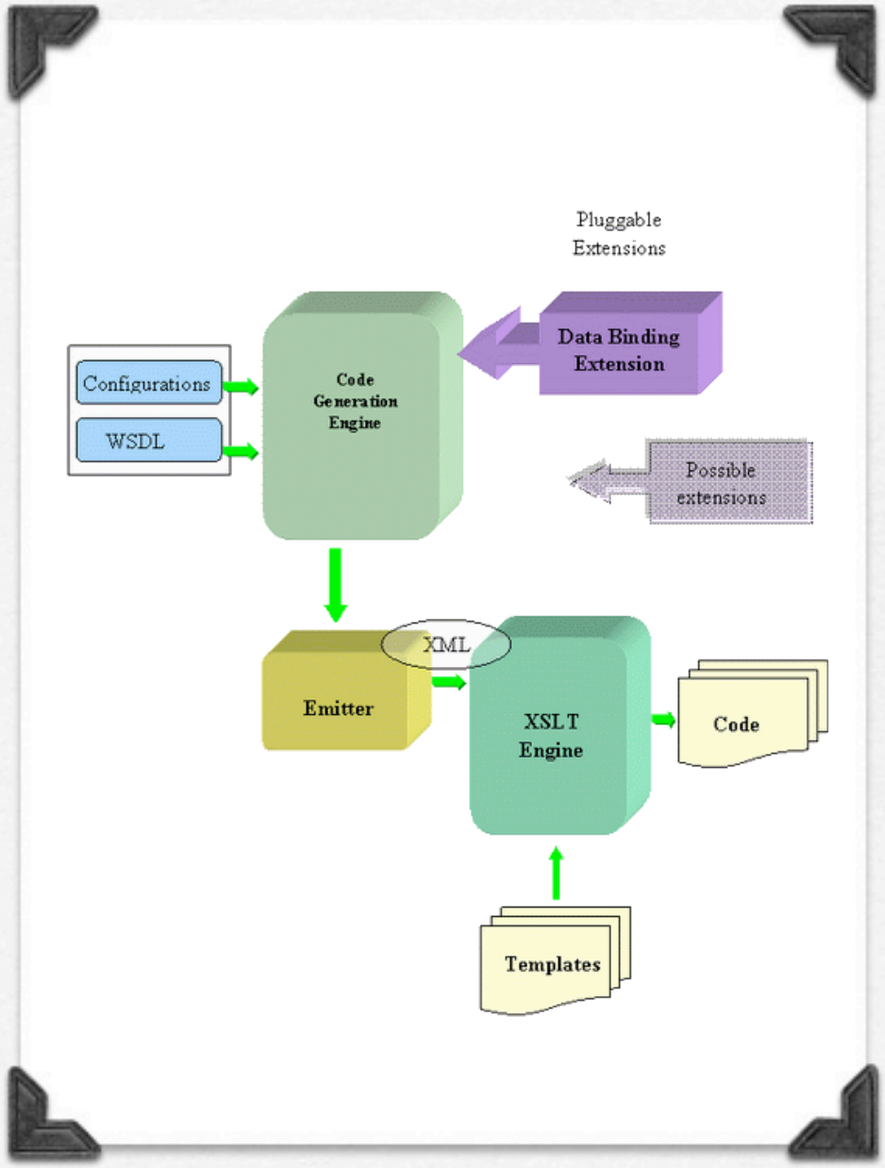


8. DataBinding

Databinding has not been included in the core deliberately, and hence the code generation allows different data binding frameworks to be plugged in. This is done through an extension mechanism where the codegen engine first calls the extensions and then executes the core emitter. The extensions populate a map of QNames vs. class names that is passed to the code generator on which the emitter operates on.

The following databinding extensions are available:

- ADB - ADB (Axis Data Binding) is a simple framework that allows simple schemas to be compiled. It is lightweight and simple, works off StAX and fairly performant. However, it does not support the complete set of schema constructs and is likely to complain for certain schemas!
- XMLBeans - XMLbeans claims that it supports the complete schema specification, and it is preferred if full schema support is needed!
- JAX-Me - JaxMe support has been added in a similar manner to XMLbeans and serves as another option for the user
- JibX - This is the most recent addition to the family of databinding extensions, and it is also another option the users have for data binding.



Tutorial

SOAP Service Node

1. WAR - (Web Archive) Distribution
2. Building Service (WSDL)
3. SOAP Monitor
4. WS-Addressing
5. Engaging Modules

SOAP Client Node - Invocations

1. One Way - Fire&Forget
2. One Way - Request&Response (Blocking)
3. One Way - Request&Response (NonBlocking)
4. Two Way - Request&Response (NonBlocking)

Resources

- Apache Axis2 Architecture Guide
  http://ws.apache.org/axis2/1_2/Axis2ArchitectureGuide.html

- Apache Axis2 User's Guide
  http://ws.apache.org/axis2/1_2/userguide.html

- Apache Axis2 Advanced User's Guide
  http://ws.apache.org/axis2/1_2/adv-userguide.html

- Writing Web Services Using Apache Axis2's Primary APIs
  http://ws.apache.org/axis2/1_2/xmlbased-server.html

- Writing Web Services Clients Using Apache Axis2's Primary APIs
  http://ws.apache.org/axis2/1_2/dii.html