

Developing Application Extensions with Apache Axis2

Chathura Herath, Eran Chinthaka
{chathura,chinthaka}@apache.org

Introduction

When looking back at the transition of web service technologies, several generations are clearly visible. The first generation web services were highly controlled interactions and can be considered as mere tests of feasibility. SOAP engines of that era, notably Apache SOAP, were meant to be “proof of concept” and not at all concerned about performance. The purpose of these first generation SOAP engines was to convince the world that web services are feasible.

Soon the toll of these first generation SOAP engines paid off. Web services were pursued as a suitable technology. More companies started showing their interest and the SOA started taking shape. This period, which can be called as the second generation of web services, required better SOAP engines that were faster. Aspects such as discovery and definition were already standardized and SOAP engines also needed to support these standards. Axis was born as one of these second generation SOAP engines.

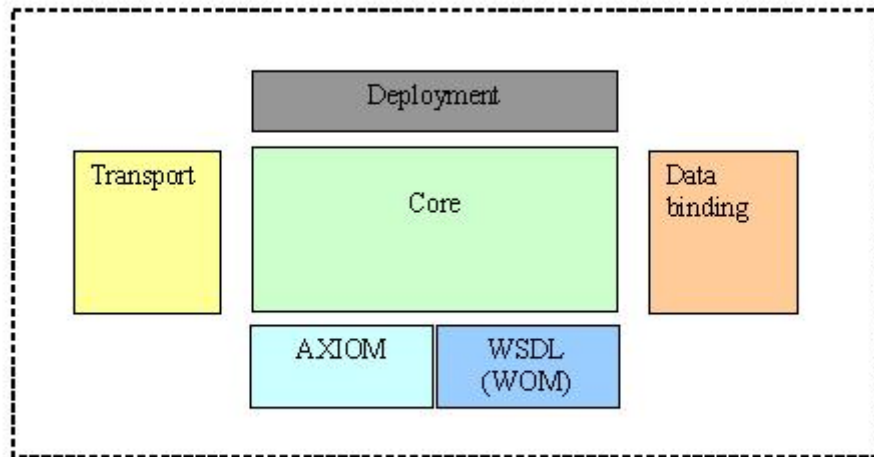
Now the second generation of web services is also coming to an end. Web services are becoming highly demanding and a large number of players have entered the web service arena. Aspects governing different facets of web service interactions have been standardized. The third generation of web services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

Features of Axis2

Axis2 is shaped by the experience from the Axis 1.x family and the advancements in the Web Service stack during the last few years. Axis2 developers take conscious effort to make sure that Axis2 performs better in terms of speed and memory despite the fact that new features and functionalities are added. Following are the key features of Axis2.

1. New XML Object Model
2. Messaging based core
3. Improved deployment Model
4. Pluggable data binding

Birds eye view of Axis2



The figure depicts the abstract appearance of Axis2. It has a core component that encapsulates the main functionality, built on the new XML object, AXIOM and the WSDL processing framework.

There are two major add-ons for the core, namely the transport and the data binding components. The deployment component is laid on top of them all.

Axis2 Installation

Introduction

Axis 2.0 can be downloaded as a [zipped binary](#) or the [source](#). This section describes how Axis2 can be installed either as a standalone server or as part of a J2EE compliant servlet container.

Prerequisites

Axis2 requires the Java Runtime Environment to be properly installed. Axis2 is developed to be run on JRE 1.4 and upwards but it has not been fully tested with the latest JRE 1.5. Hence it is safe to run Axis2 with Java 1.4. If the JRE is not already in place it must be installed to proceed further. For instructions on setting up the JRE in different operating systems, please visit <http://java.sun.com>.

All the required jars are shipped with the binary distribution and if the source distribution is used, running the maven build will automatically download the required jars for you.

Following sections describe how each type of distribution needs to be installed. Since the process with the source distribution is similar to the binary distribution after building, the first section explains the process of building Axis2 from source. If you have the binary

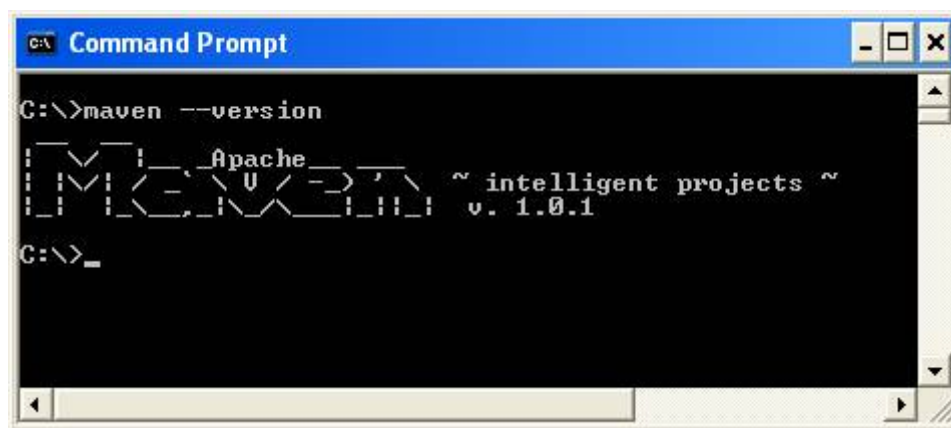
distribution you can skip the build sections and directly go to the binary installation section.

Building Axis2 from source

Setting up the Environment and the tools

The Axis2 build is based on [Maven](#). Hence the only prerequisite to build Axis2 from source distribution is to have Maven installed. Even though extensive instruction guides are available at the Maven site, this guide also contains the easiest path for quick environment setting. Advanced users who wish to know more about Maven can visit [here](#).

For Windows users the easiest way is to download the windows installer package. Once the installer package is run, all the necessary environment variables will be properly set. Once Maven is installed, the success of the installation can be tested by typing `maven version` in the command prompt.



For Unix/Linux users the tar ball or the zip archive is the best option. Once the archive is downloaded expand it to a directory of choice and set the environment variable `MAVEN_HOME` and add `MAVEN_HOME/bin` to the path as well. More instructions for installing Maven in Unix based operating systems can be found [here](#).

Once maven is properly installed it's all that is needed to start building Axis2.

The Axis2 source distribution

The [source distribution](#) is available as a zipped archive. All the necessary build scripts are included with the source distribution. Once the source archive is expanded into a directory of choice, moving to the particular directory and running maven command will build the Axis2 jar file.

```
C:\WINDOWS\system32\cmd.exe
mples-M2.jar'
Copying: from 'D:\Projects\LSF\Axis2\Axis1.0\modules\samples\project.xml' to: 'C:\Documents and Settings\me\.maven\repository\axis\poms\axis2-Samples-M2.pom'
jar:install:

build:end:

init:
[echo] the files are up to date =

create-jar:
[jar] Building jar: D:\Projects\LSF\Axis2\Axis1.0\target\lib\axis2-M2.jar

create-lib:

[war] Building war: D:\Projects\LSF\Axis2\Axis1.0\target\axis2.war
BUILD SUCCESSFUL
Total time: 2 minutes 6 seconds
Finished at: Mon Jun 06 11:10:06 LKT 2005

D:\Projects\LSF\Axis2\Axis1.0>
```

Once the command completes, the binaries (jar files in this case) can be found at a newly created "target" directory.

Note For the first Maven build (if the maven repository is not built first) it will take a while since required jars need to be downloaded. However this is a once only process and will not affect any successive builds.

The default maven build will however build only the Axis2 jar file. To obtain a WAR (Web Archive), "maven war" command should be issued. This will create a complete WAR with the name axis2.war inside the target directory.

Once this build step is complete, the binaries are ready to be deployed.

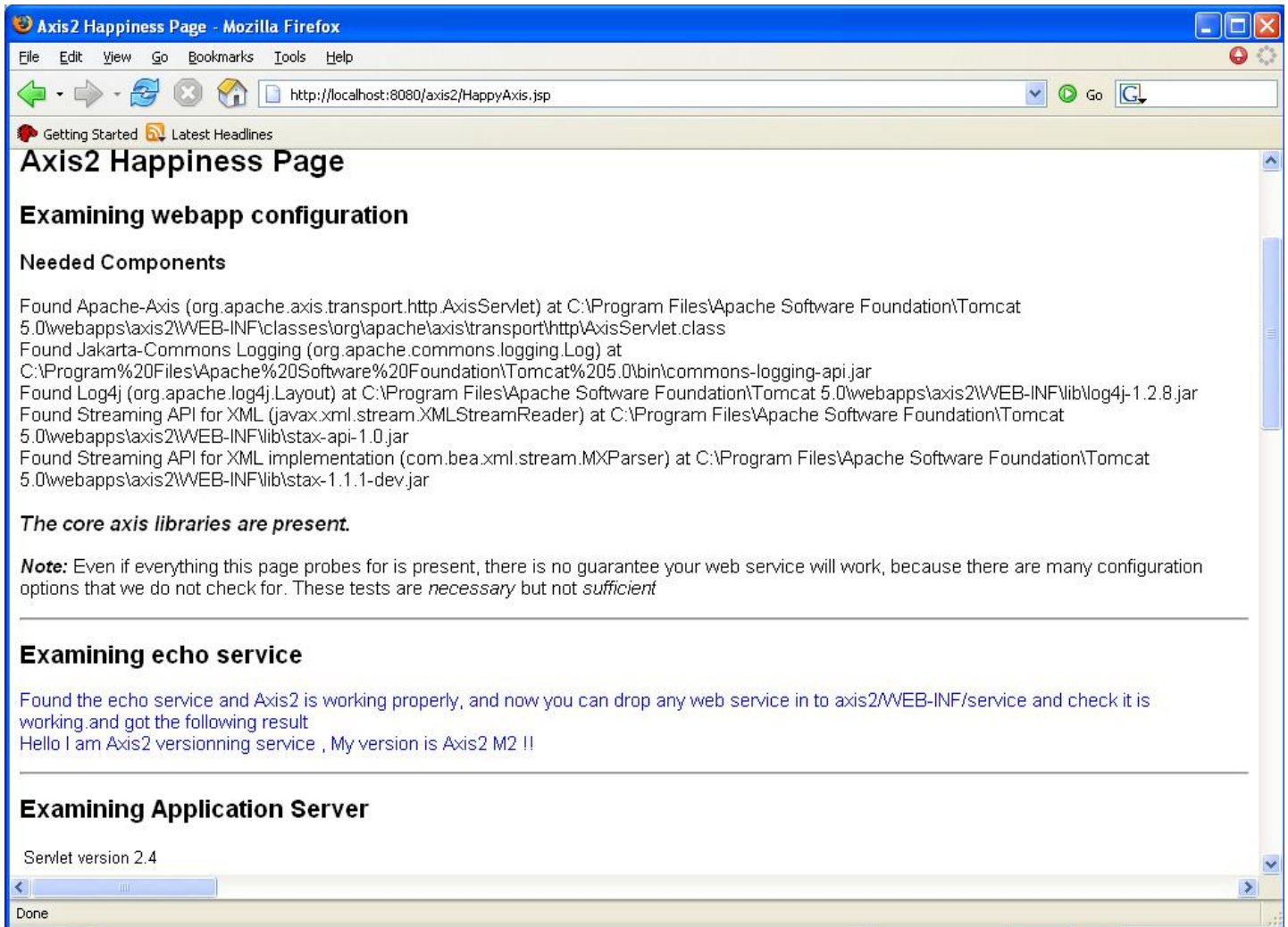
Installing Axis2 in a Servlet container

Installation of the WAR is quite simple. It's a matter of dropping the war in the webapps folders and most servlet containers will automatically install the war. However some servlet containers may require a restart in order to capture the new web application. Please refer your servlet container documentation for more information about this.

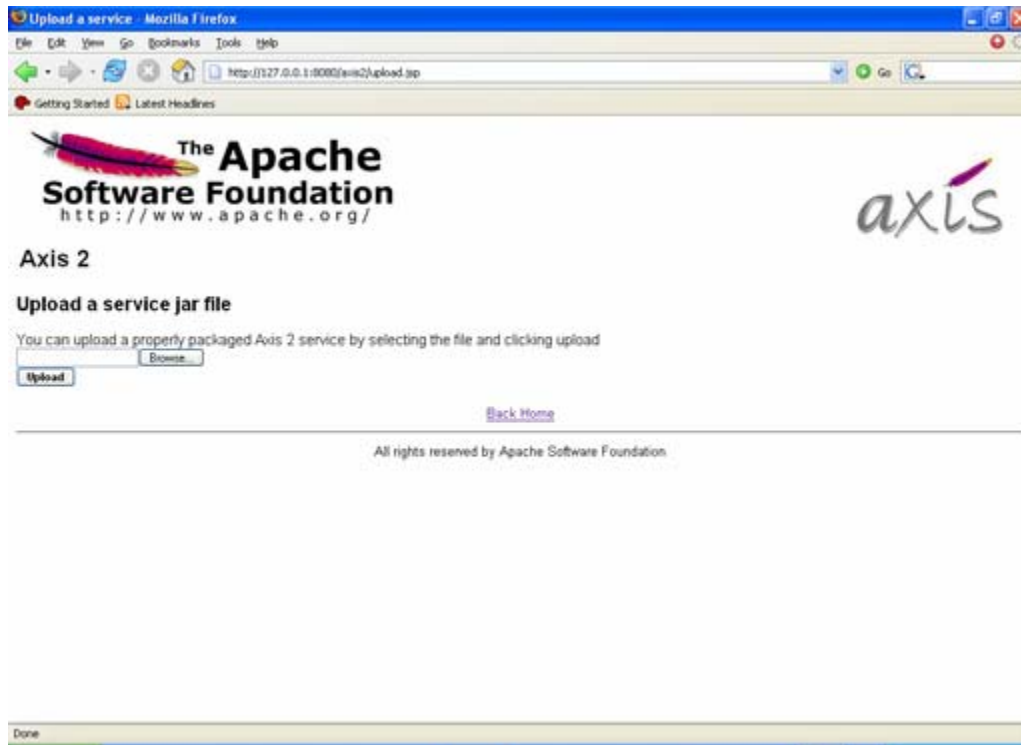
Once the WAR is successfully installed it can be tested by pointing the web browser to the **http:// <host :port>/ axis2**. It should produce the following page.



To ensure that everything is fine and smooth, a probing of the system can be done through the validate link. If the validation fails then the war has failed to install properly or some essential jars are missing. At such a situation the documentation of the particular servlet container should be consulted to find the problem. The following page is a successful validation. Note the statement core Axis2 libraries are present.



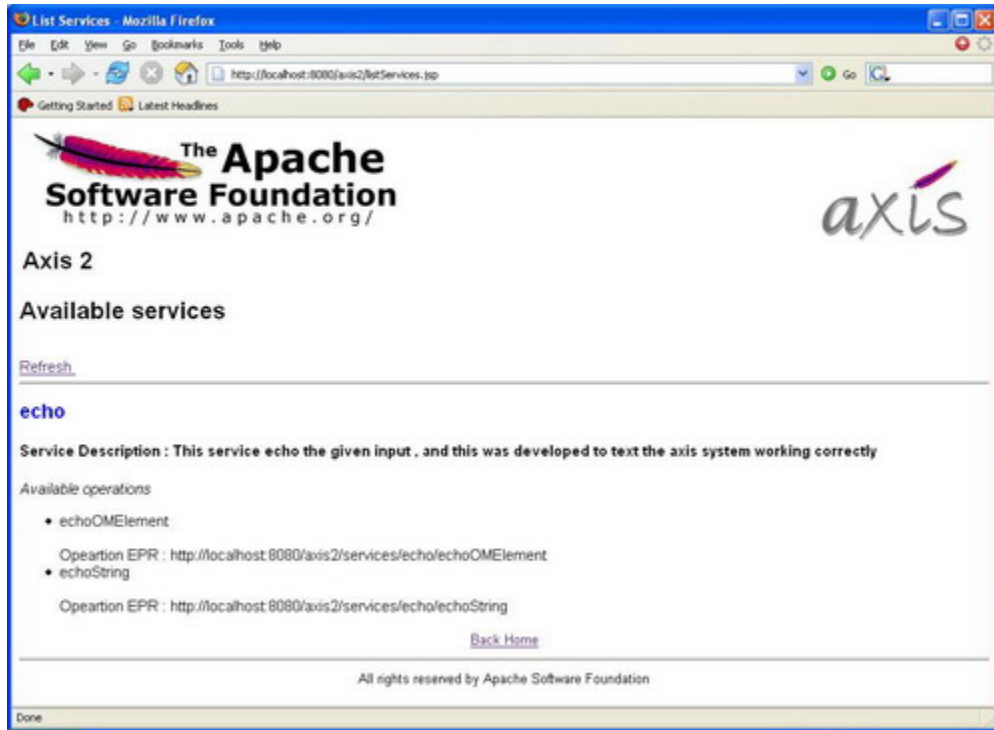
The Axis2 web application also provides an interface to upload services. Once a service is created according to the service specification as described in userguide that jar file can be uploaded using the upload page.



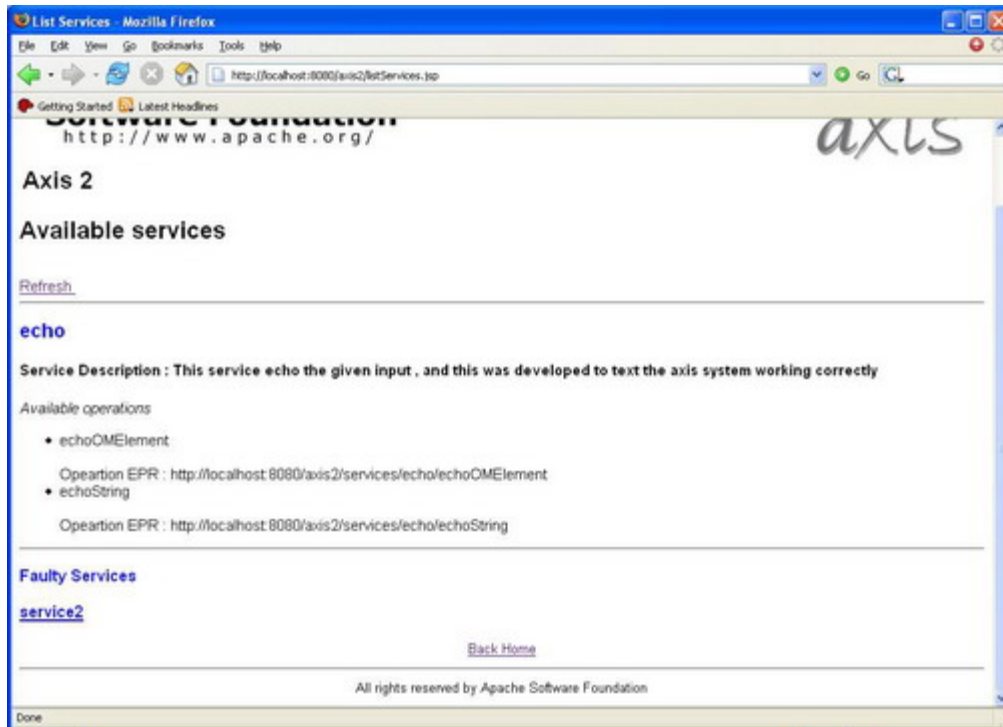
The uploaded jar files will be stored in the default service directory. For Axis2 this will be the <webapps>/axis2/WEB-INF/services directory. Once a service is uploaded it will be instantly installed.

Since Axis2 supports hot deployment one can drop the service jar directly through the file system to the above mentioned services directory and it will also cause the service to be automatically installed without the container being restarted.

To check the successful installation of a service *available services link* is provided. The services and the operations of successfully installed services will be displayed in the available services page.



If the service has deployment time error it will listed out those services as faulty services. And If you click on the link it will show your the deployment fault



Deployment time error message



Axis2 Administration is all about configuring Axis2 at the run time and the configuration will be transient , and more descriptions are available in [Axis2 admin web guide](#)

Installing Axis2 Eclipse plug-in

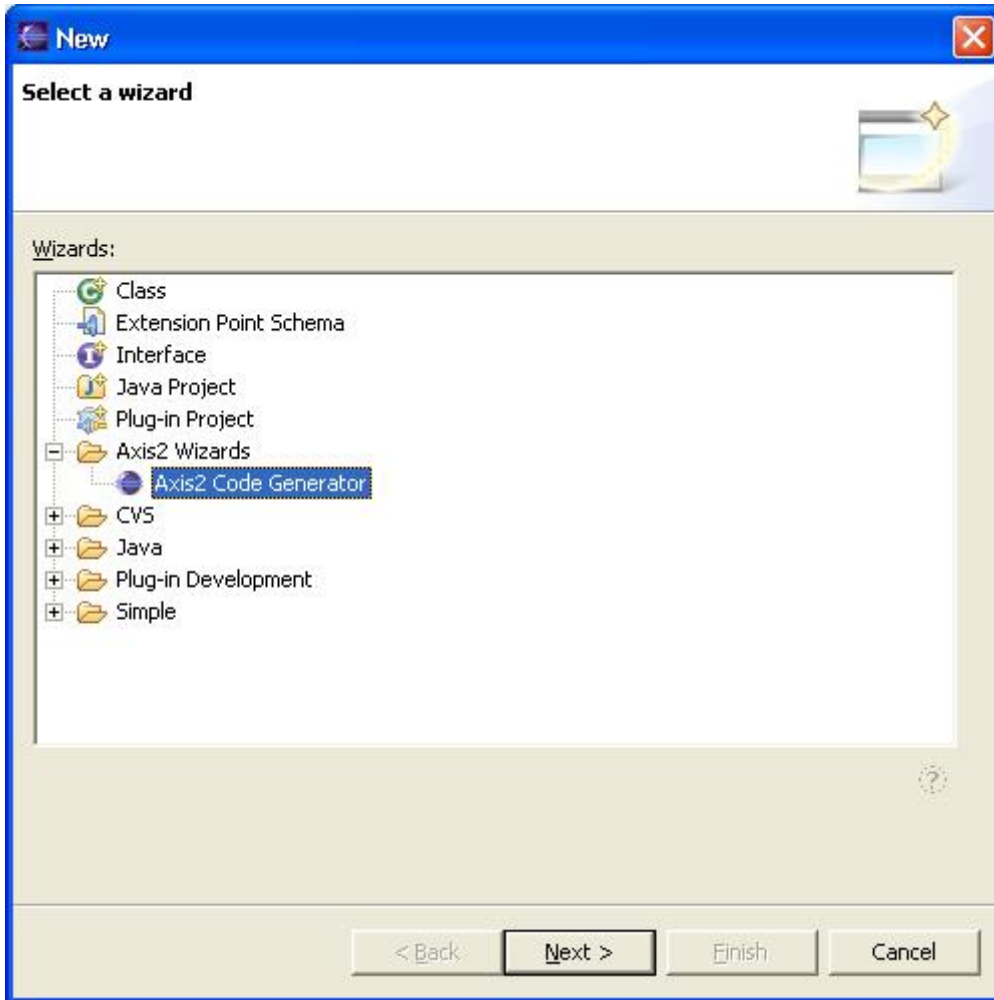
The easiest way to obtain the plug-in would be the binary distribution. The full Axis binary distribution contains the compiled version of this plug-in under the tools directory.

If one needs to build the plugin from source it is not as trivial as running the Maven build. The reason is that the plug-in depends heavily on the Eclipse classes, which are only available in an Eclipse environment. The recommended procedure is to run the create-project.xml (in the "modules\tool" directory of the source distribution) build file which will create two folders (the other one for the Service Archiver tool) and copy the necessary files to relevant folders. Then Eclipse should be configured to open the contents in a PDE project. Please go through the Eclipse documentation to learn how to open projects in the PDE format.

Once you've obtained the plug-in just unzip the content of the plug-in archive to the eclipse plug-in directory (if it is the zipped-binary version) or copy the necessary folders to the eclipse plug-in directory and restart Eclipse.

Note - This plug-in works on Eclipse version 3.0 and upwards

If the plug-in is properly installed you should see a new wizard under the "New" section.(use the File -> New -> Other or Ctrl + N)



More information on Axis2 eclipse plug-in is available [here](http://ws.apache.org/axis2/CodegenToolReference.html) (<http://ws.apache.org/axis2/CodegenToolReference.html>).

Scenario for the Client demonstration – Google Spell checker

Introduction

In order to demonstrate the capabilities of the axis2 client API and the power of its tools, the client demonstration includes the code generation from a publicly available web service and using the generated stubs to access the service.

The selected public Web service is the Google Web service available at <http://www.google.com/apis/> . In a nutshell accessing this service requires one to register at Google and to obtain a personal key code, free of charge. This key code is only allowed a certain number of requests per day. More information of the formalities of registration can be found at http://www.google.com/apis/api_faq.html. The rest of this

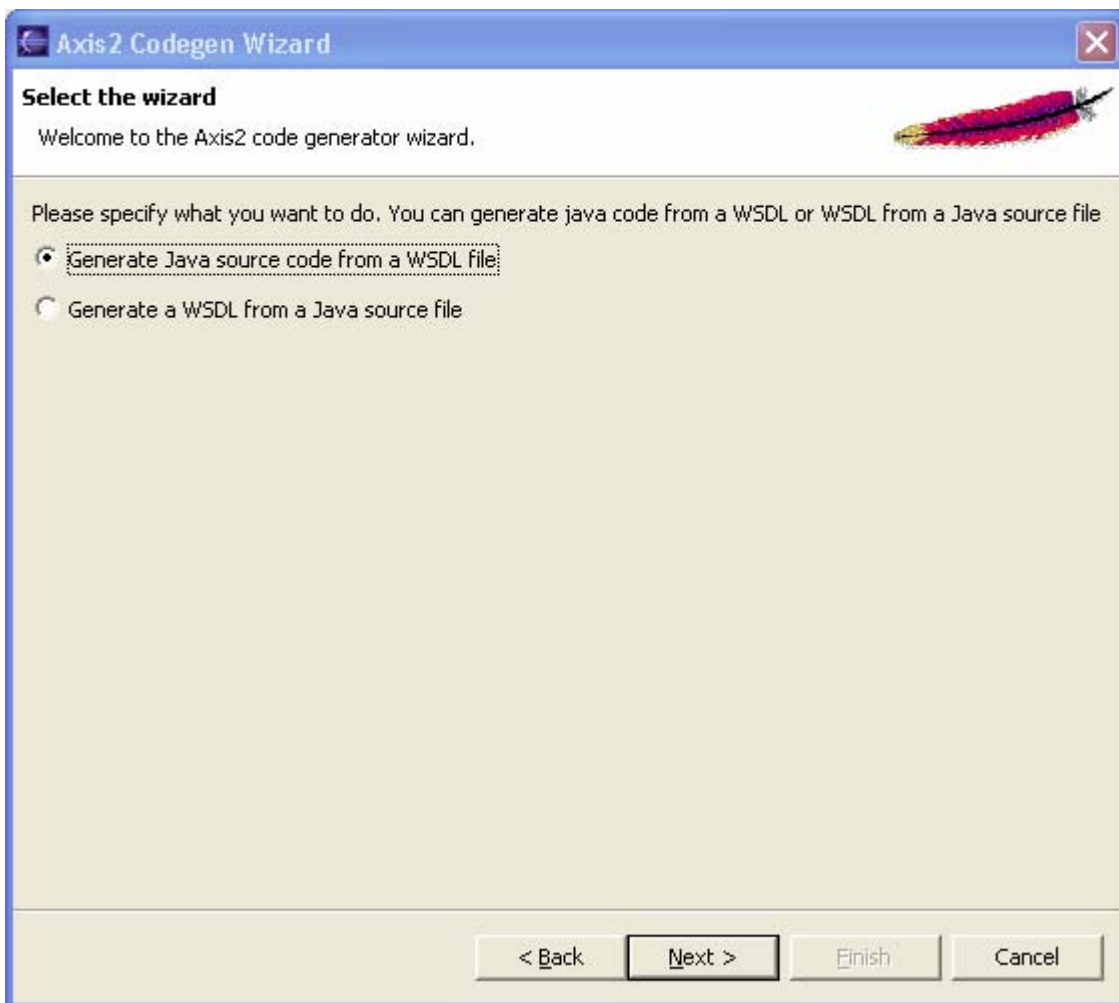
note assumes that you have successfully registered at Google and already has a valid key code.

In Web services, the most important descriptor is the WSDL file of the service. It includes all the information needed for a client to access a service and usually tools are available for code generation, based on the WSDL file. The WSDL for the Google web service is available at <http://api.google.com/GoogleSearch.wsdl>.

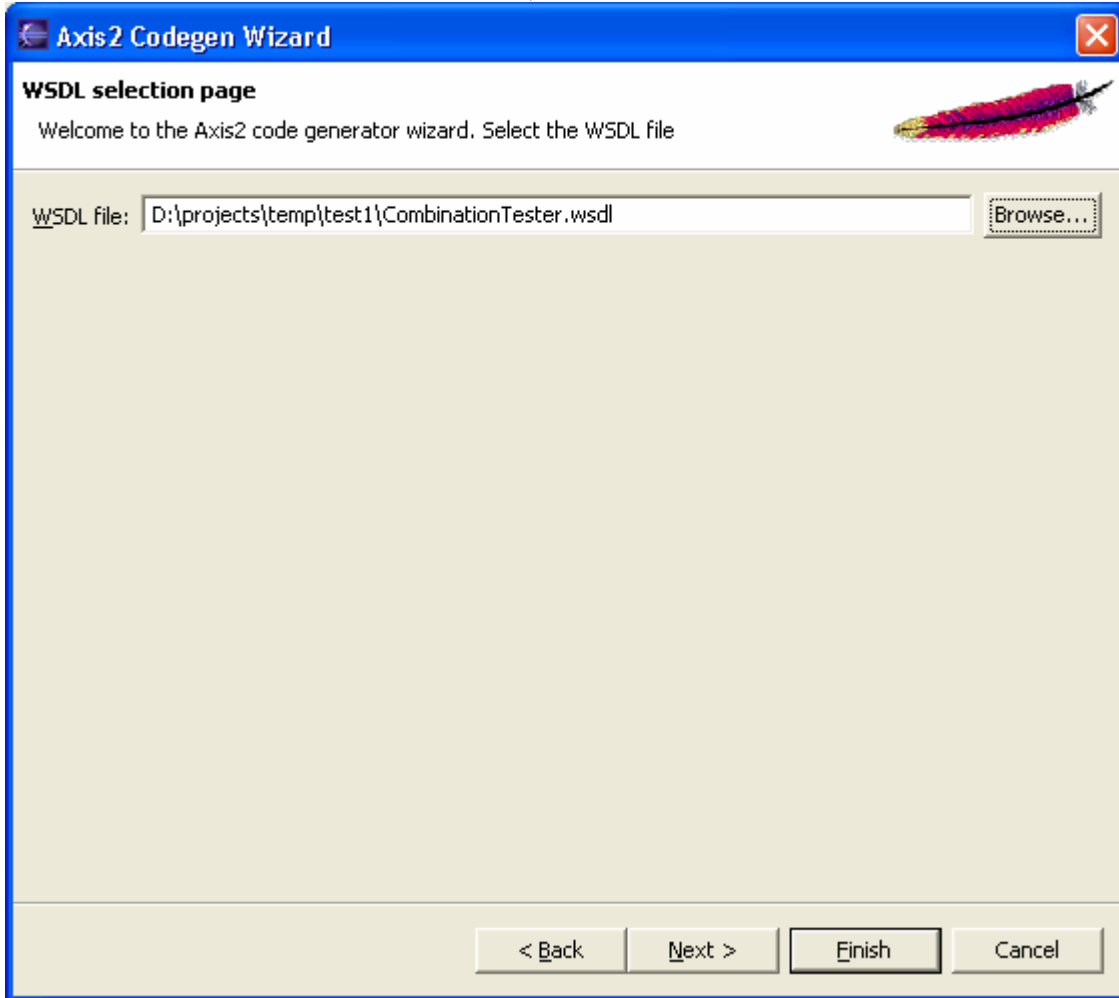
Once the WSDL is known, the next step is to generate the code. Axis2 comes bundled with a code generation tool, which has both the graphical and command line user interfaces. This demonstration uses the eclipse plug-in of the code generation tool.

Running the Code Generator


1. Select the Stub generation wizard




2. Select the WSDL file (Either the URL directly or point to the local file system where the WSDL file is available)



3. Select the options. If only the client side code is needed the default options are usually sufficient. However ticking on the “generate test case” will generate a junit test case that can be used to test the generated stubs

Axis2 Codegen Wizard 

Options 

Set the options for the code generator

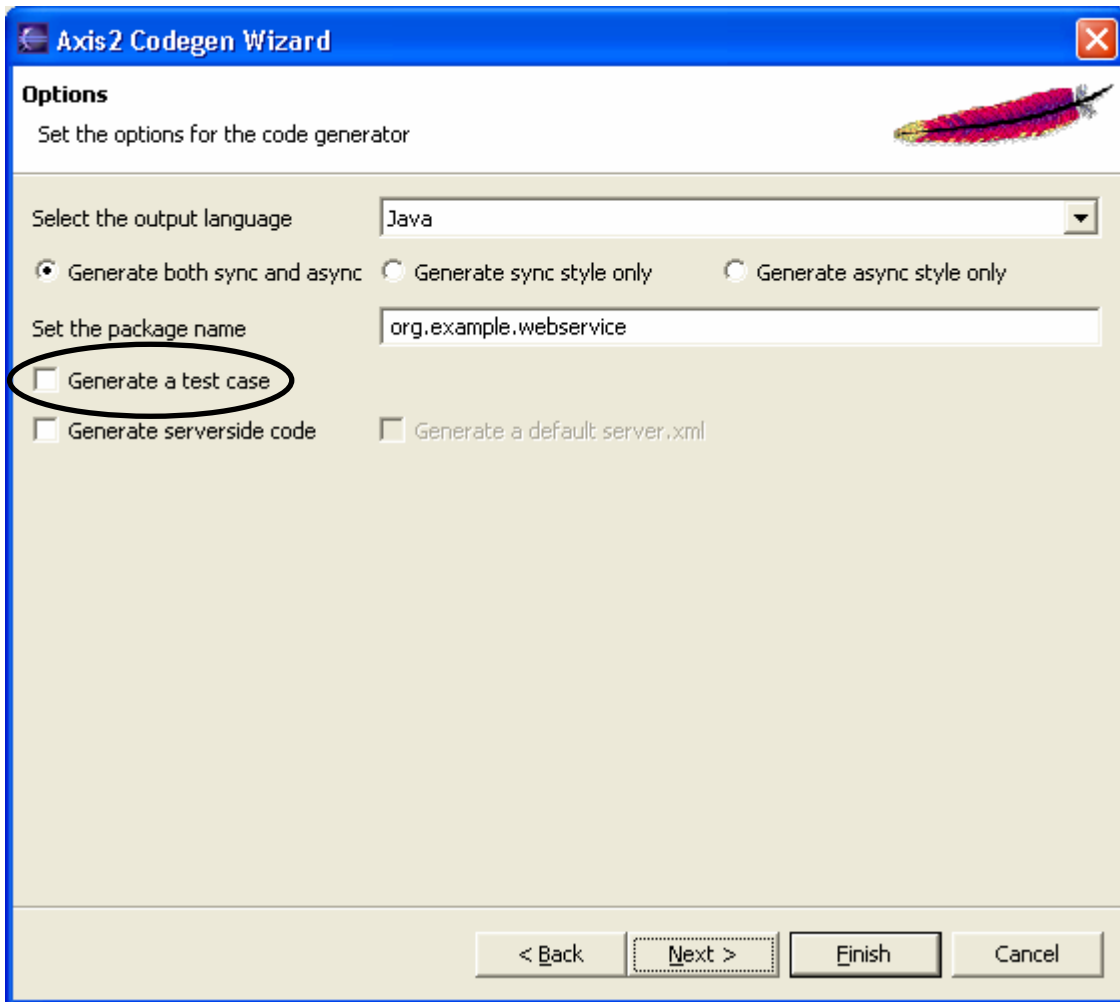
Select the output language:

Generate both sync and async Generate sync style only Generate async style only

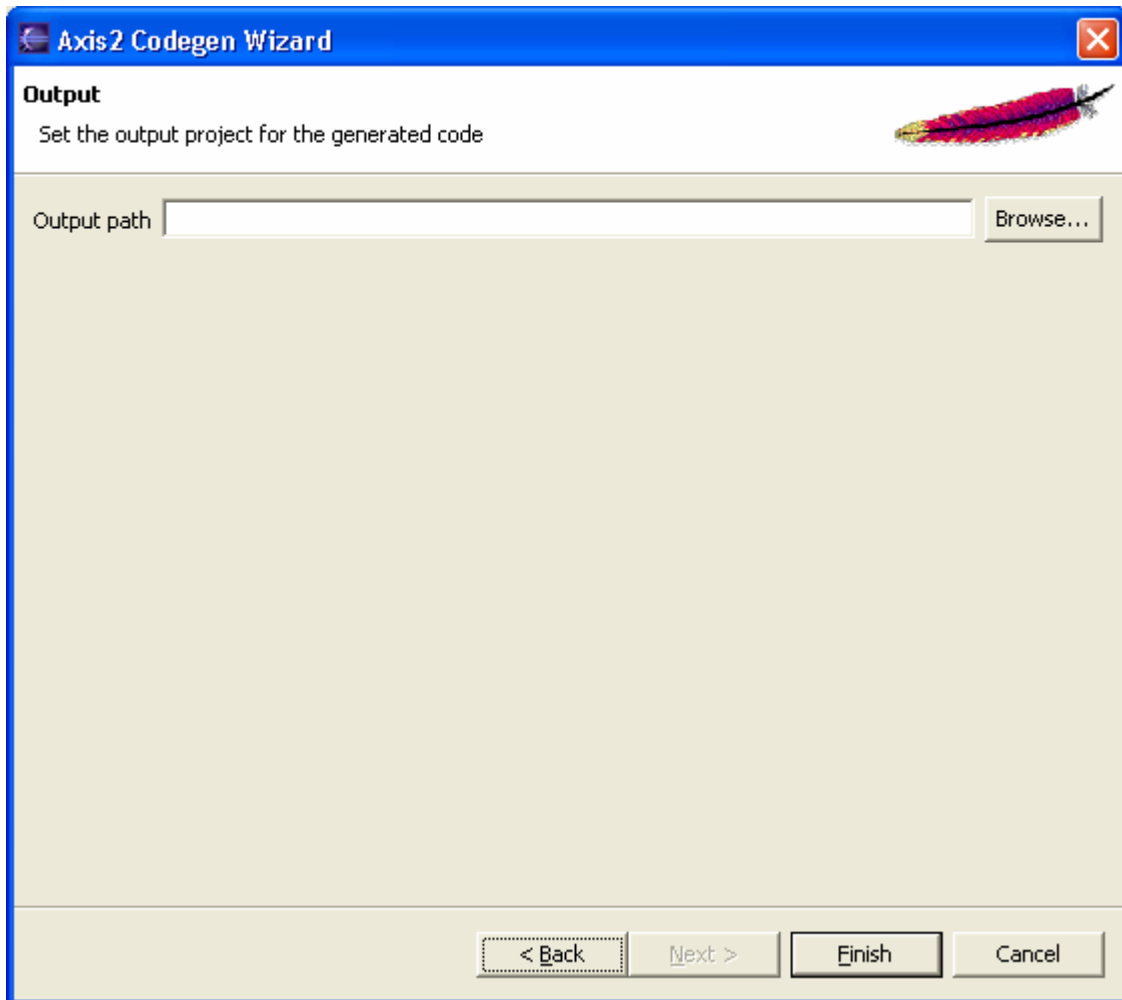
Set the package name:

Generate a test case

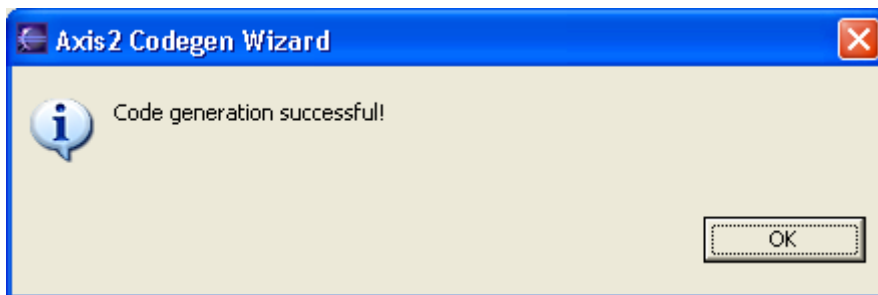
Generate serverside code Generate a default server.xml



4. Select the output location. This will be defaulted to the users home directory



Once the Finish button is pressed, the code will be generated for the given WSDL. This message will be displayed if the code generation is successful.



Well done! We are through the first hurdle.

Examining the generated code

The Generated client code comprises of several types of classes.

1. The *Stub* Implementation

2. The Stub interface
3. The call back handler class for asynchronous call backs (would not be available if the sync only option is selected)
4. the test class(would be available only if the 'generate test class' option is selected)
5. Databinding classes. Found in the *.databinding package.

The most important and ultimately the most useful class here is the Stub implementation. The stub implementation will have two methods per operation in the web service. Here is a sample method signature

```
public <return type> <method name> (<input type> param)
throws java.rmi.RemoteException;

public void start<method name> (<input type> param,final
<callback handler> callback) throws
java.rmi.RemoteException;
```

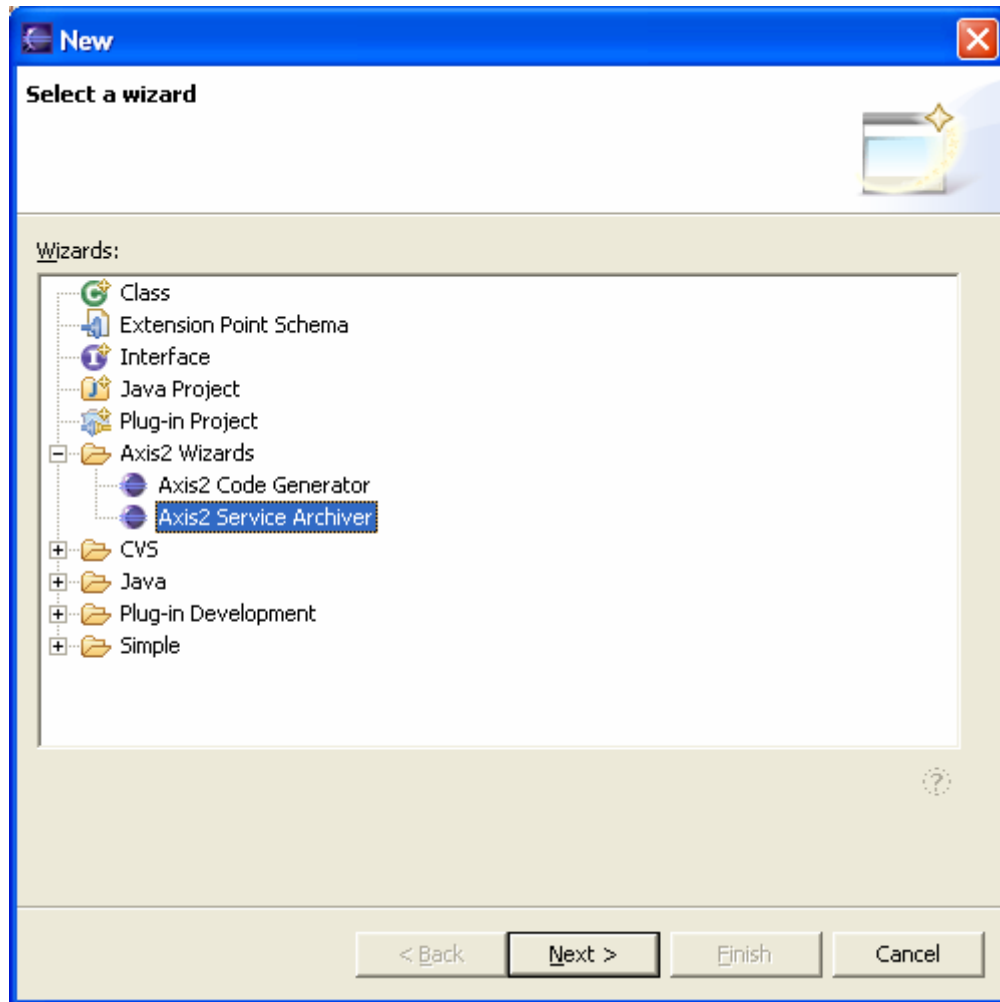
First method includes code to call the service synchronously. This method is the equivalent of the usual method call that blocks until the processing is done.

The second method call is somewhat special. It is the asynchronous method call and requires a callback handler as a parameter

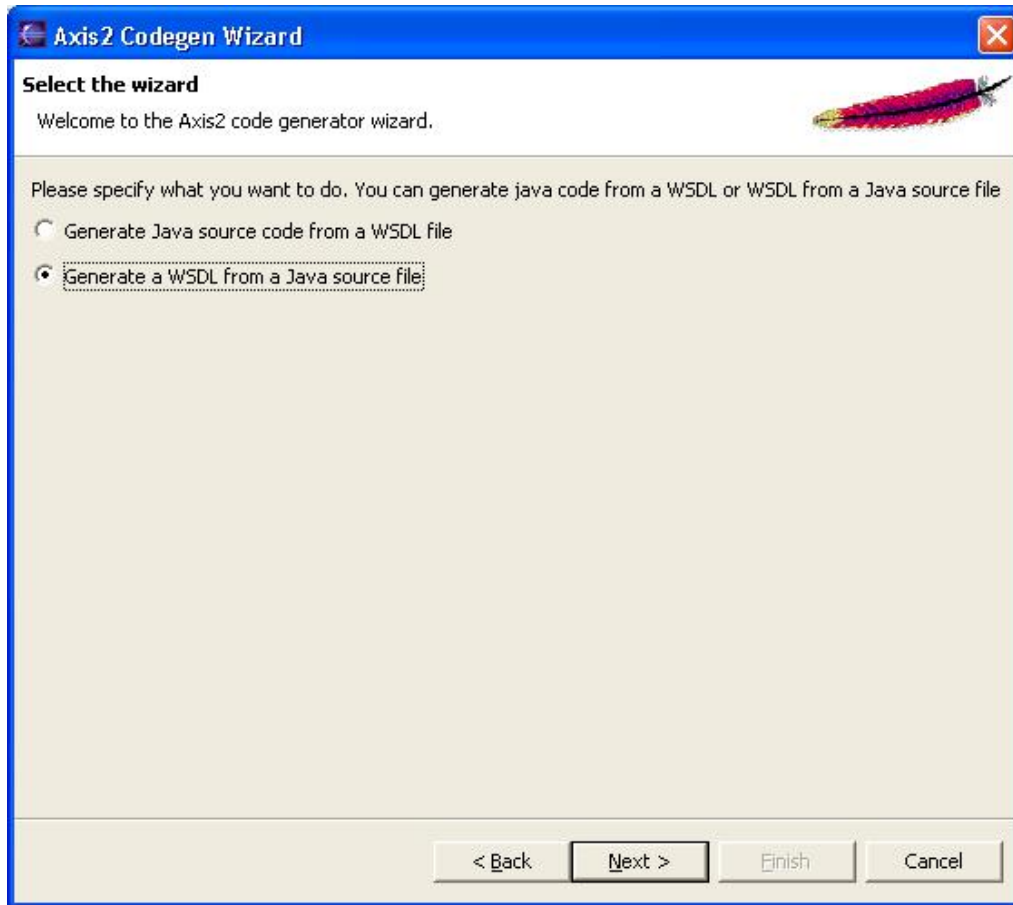
Implementing And Developing Services In Axis2

Web service is intended to provide interoperability between heterogeneous environments. So the contract that governs the communication is a key. The WSDL defines the contract that governs the message exchange. This tutorial will start the development of the web service from the contract the WSDL.

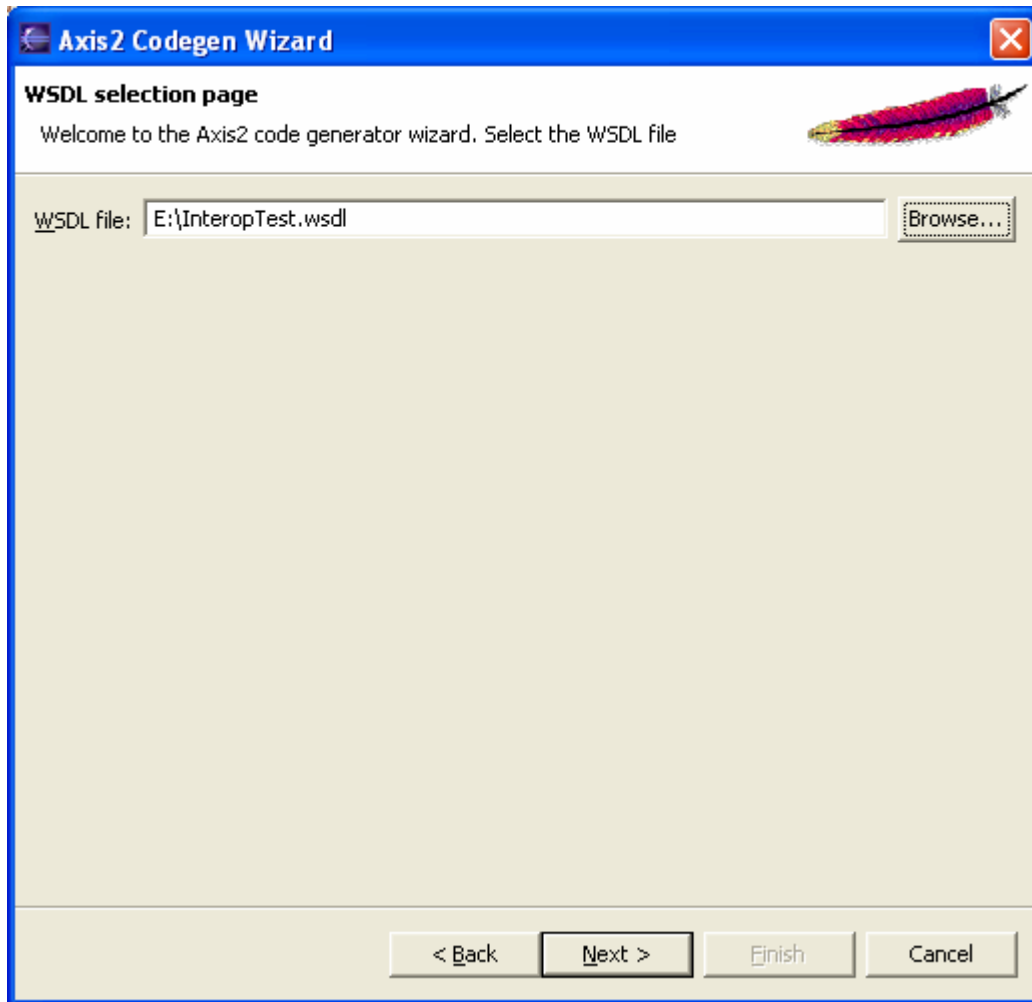
Axis2 distribution ships an Eclipse pug-in that is capable of generated required server skeletons and the required types that are necessary for the service deployment. Once the plug-ins are properly integrated to the Eclipse the Wizards will be available as shown in the following figure.



In the first screen of the Wizard the user is supposed to select whether he wants to start from the WSDL and then code generate or starting from a java artifact the WSDL to be generated. In this case the generation of java source given a WSDL file will be described.

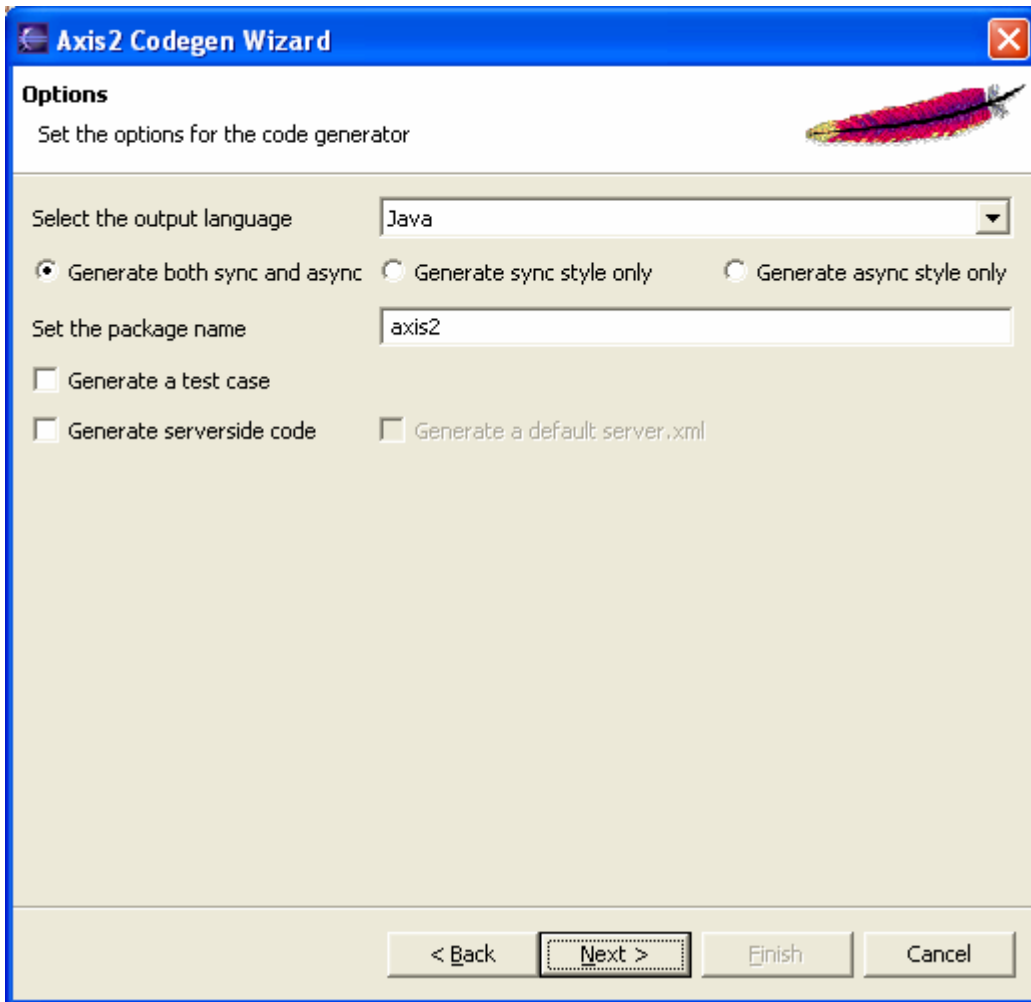


Next the Wizard will require the user to specify the WSDL file that the code generation should be based on.

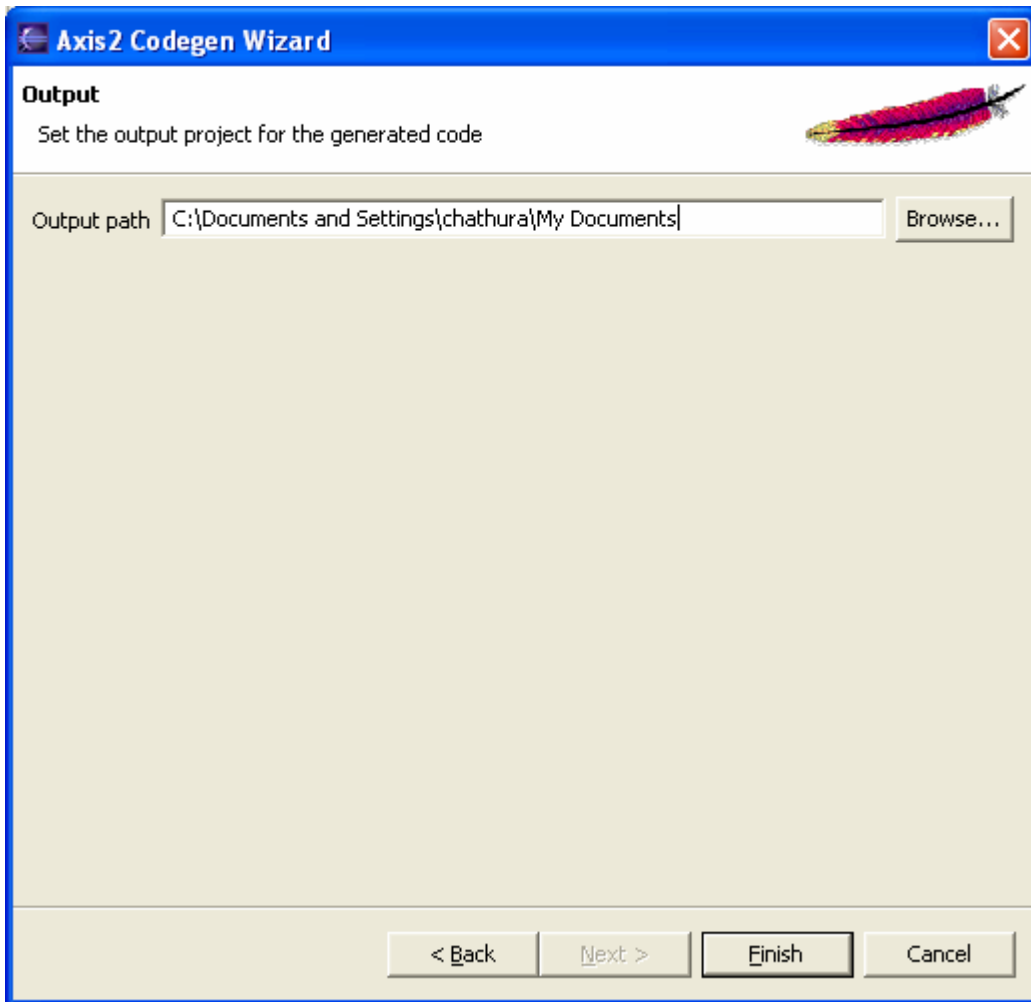


Next the user can specify the more specific things like the output language, programming model, etc of which are explained below.

- Output Language- Language of the generated stub.
- Programming model- Blocking or non Blocking.
- Package Name - Package of the generated code.
- Test case – To generate a unit test case that will test the generated code.
- Generate server side – The default code generation is for the server side. This option will enable the server side code generation.



Next the user can select the output location of the generated code.



Notes about generated code : *One can generate server side skeletons and/or client stubs using this tool or both along with a unit test case.*

Client side generated code

The client side generated code can be generated either using the blocking API (synchronous) or non-blocking API (asynchronous) or both. This can be selected from ticking the relevant check boxes in the code generation tool.

- **Blocking API** -Once the service invocation is called, the client application hangs and gets the control back only when the operation completes and the client receives a response or a fault. This is the simplest way of invoking Web Services and suites many business situations.
- **Non-Blocking API** - A callback or polling based API and hence once a service invocation is called, the client application immediately gets the control back and

the response is retrieved using the callback object provided. This approach provides the flexibility to the client application to invoke several Web Services simultaneously without blocking on the operation already invoked.

The generated code will consist of following artifacts if both blocking and non-blocking cases are selected.

- An interface with the name of the port-type of the bound interface of the service which will have all the methods that will be available under that particular service.
- A stub that implements the above interface and hides the engine level complexity and presents a nice API to invoke a particular operation. For a given WSDL operation “foo”, the stub will contain two methods if both synchronous and asynchronous option is selected.
 - Method foo(<parameters>) which will be used during a synchronous web service call
 - Method startFoo(<parameters>, <callbackhandler>) which will be used during an asynchronous web service call.
- A callback handler class with two methods for each method “foo” with empty method stubs. This class is only applicable in the case of the asynchronous web service invocation.
 - Method receiveResultFoo(<result object>) which will be called by the engine once the asynchronous invocation receives a result.
 - Method receiveErrorFoo(<exception>) which will be called by the engine if an unexpected result occur during the invocation.
- Set of classes that will be used for data binding.

Server side generated code

Server side generated code will basically consist of a skeleton with empty method stubs so that use can fill it up and deploy it as the service implementation. Generated artifacts include a Message Receiver, thus the selection of the programming model (i.e. whether its synchronous or asynchronous) will be encapsulated in that particular Message Receiver.

The generated code will consist of following artifacts.

- A skeleton that has methods with empty bodies which will become the actual service implementation once the method bodies are filled with the proper business logic.
- A Message Receiver which will be deployed along with the service implementation.

- A service.xml file which will be generated based on the default configurations, which is required when deploying the service.
- Set of classes that will be used for data binding.

Generate test code

In the code generation tool if the user selects the “generate a test case” option, the code generation tool will generate files required to test the generated code in a package <user package>/test.

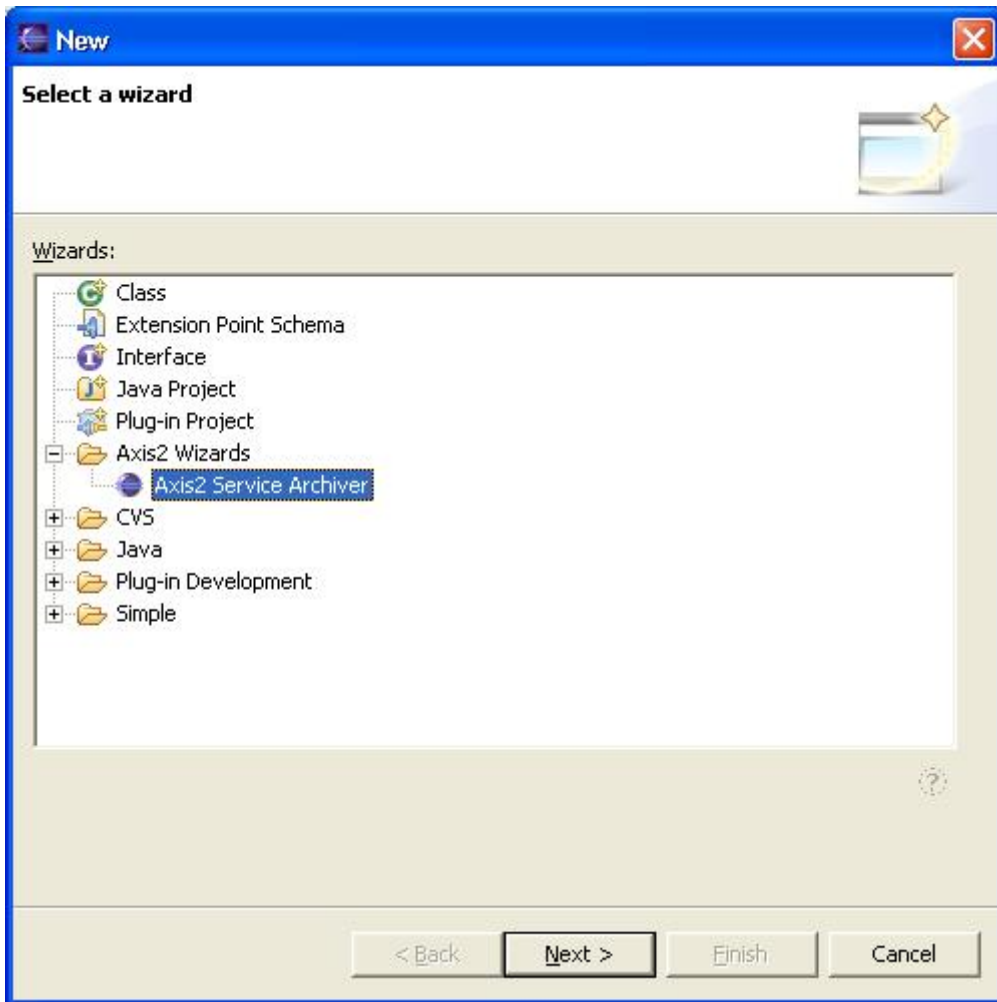
The generated code will consist of following artifacts.

- A dummy implementation of the server skeleton.
- A service.xml file to deploy the service.
- A unit test case that will setup a server and deploy the dummy implementation of the server skeleton and use the client stub to invoke each one of the services.

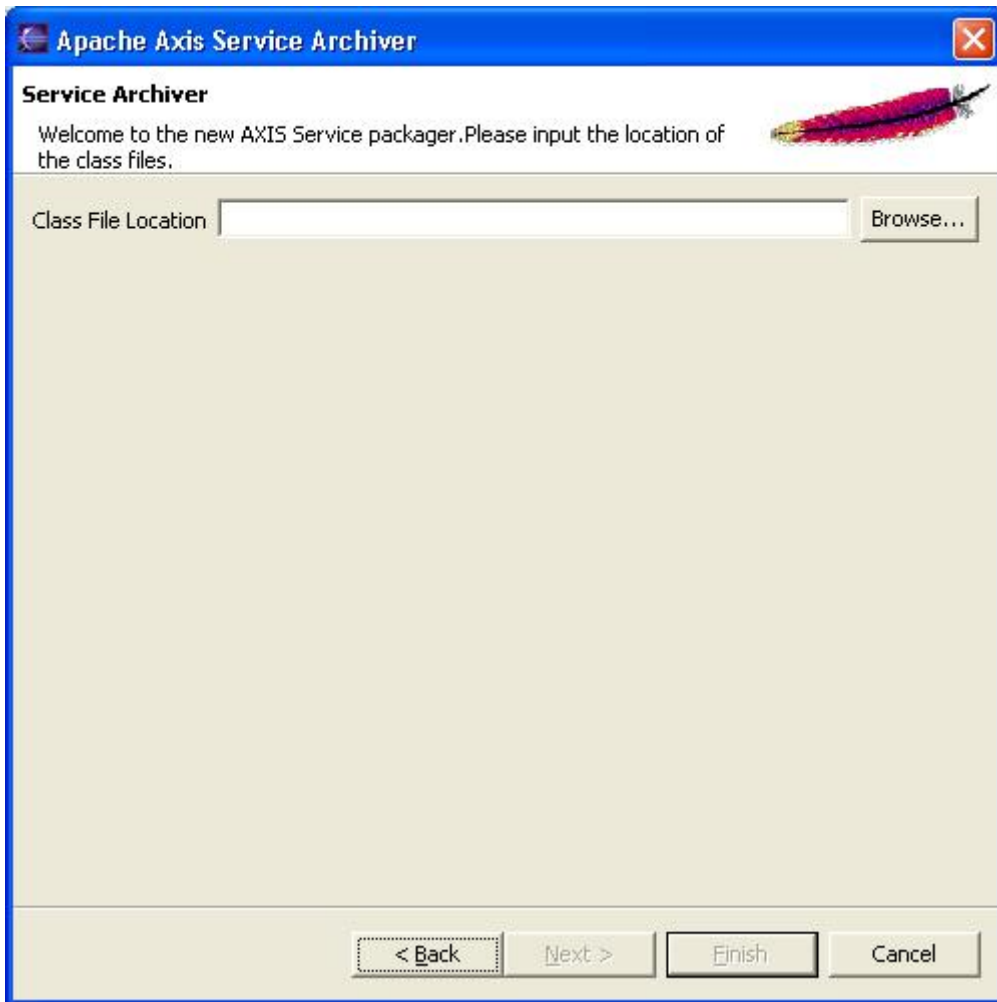
Bundling the service

Axis2 comes with a simple service archiver tool which provides easy to use functionality to develop a axis archive or an "aar" file or a "jar" file that can be deployed as a web service to the Axis2. This tool is in the form of an Eclipse plug-in and can be downloaded from the downloads section. The following sections describe how the tool can be used.

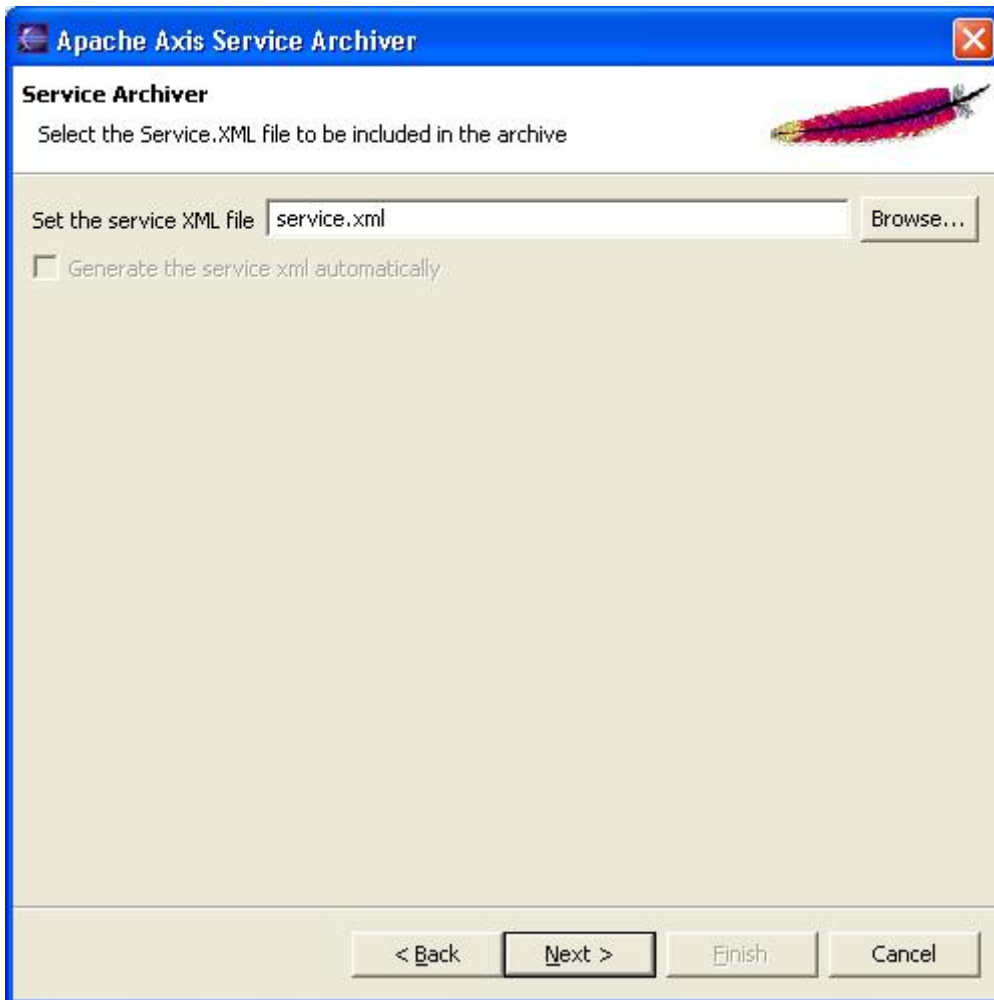
If the plug-in is properly installed you should see a new wizard under the "New" section.(use the File -> New -> Other or Ctrl + N)



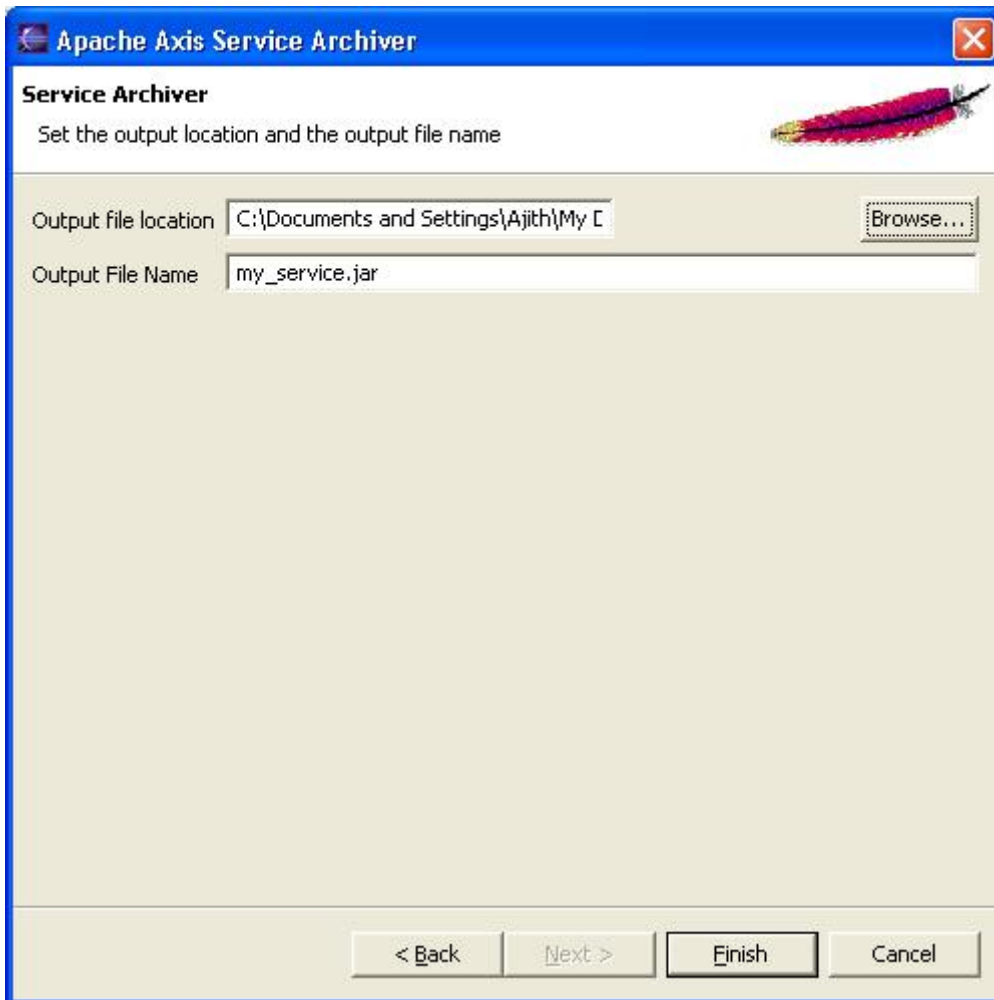
Selecting the wizard and pressing the next button will start the code generator wizard. Following is the first wizard page.



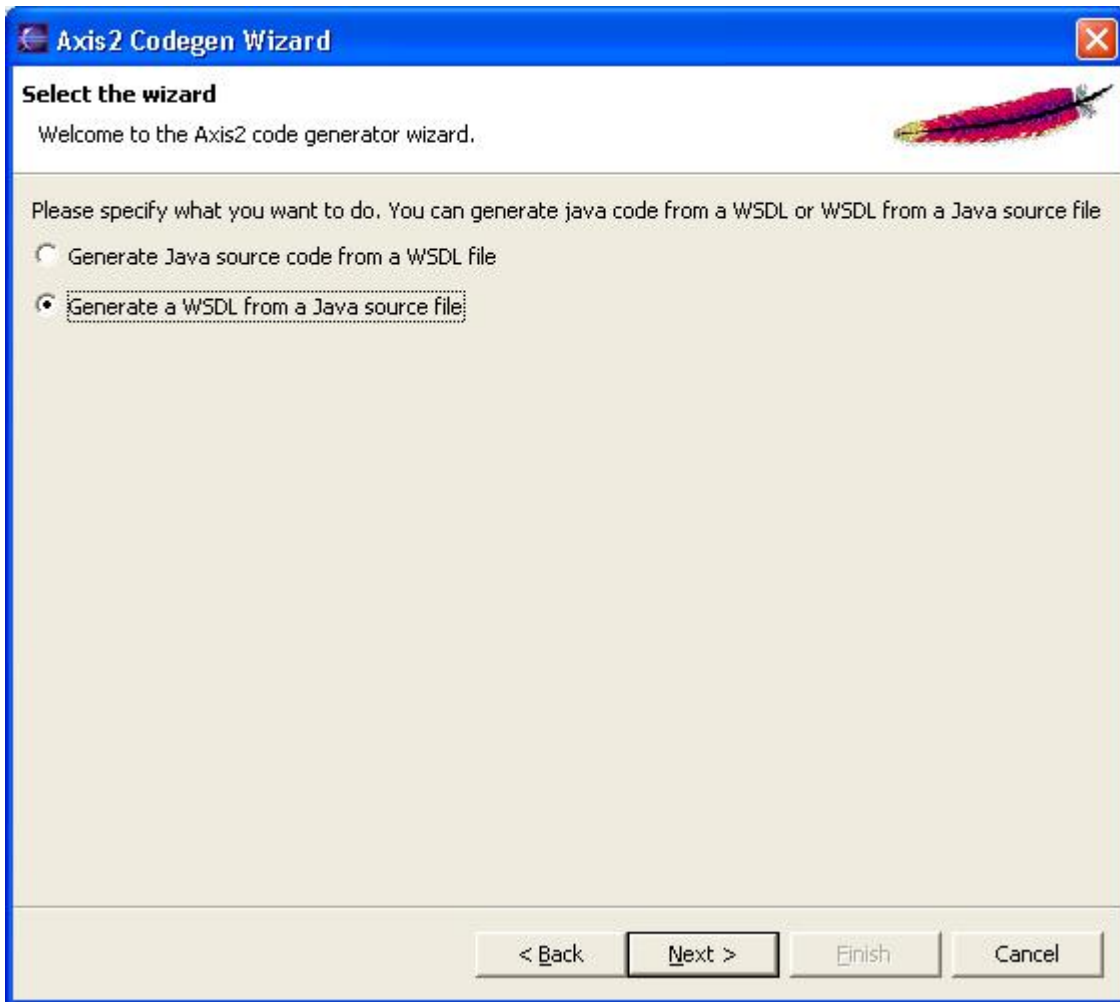
Once the class file folder is given (which should be a folder in the file system) the next button will be enabled. Page 2 of the wizard requires you to locate the server.xml



Note that the automatic generation of the service.xml is disabled for this release. After the service.xml is located you can point to the output location and specify the file name.



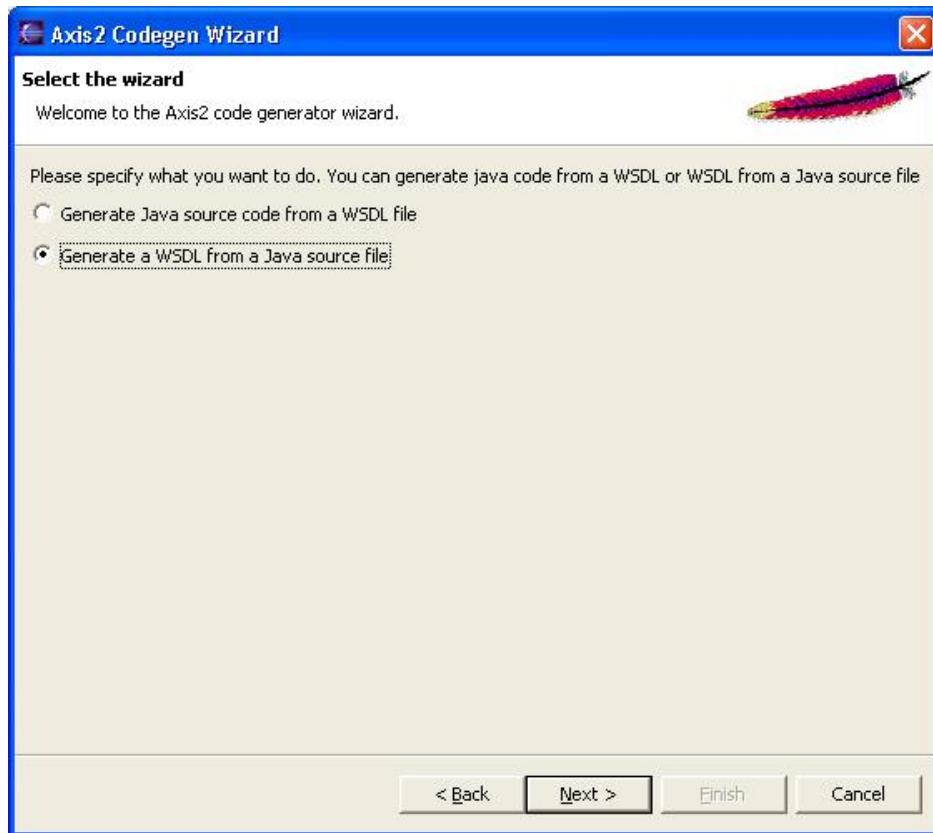
Once all the parameters are filled, finishing the wizard will generate the service archive. This service archive can be hot deployed to the axis2.



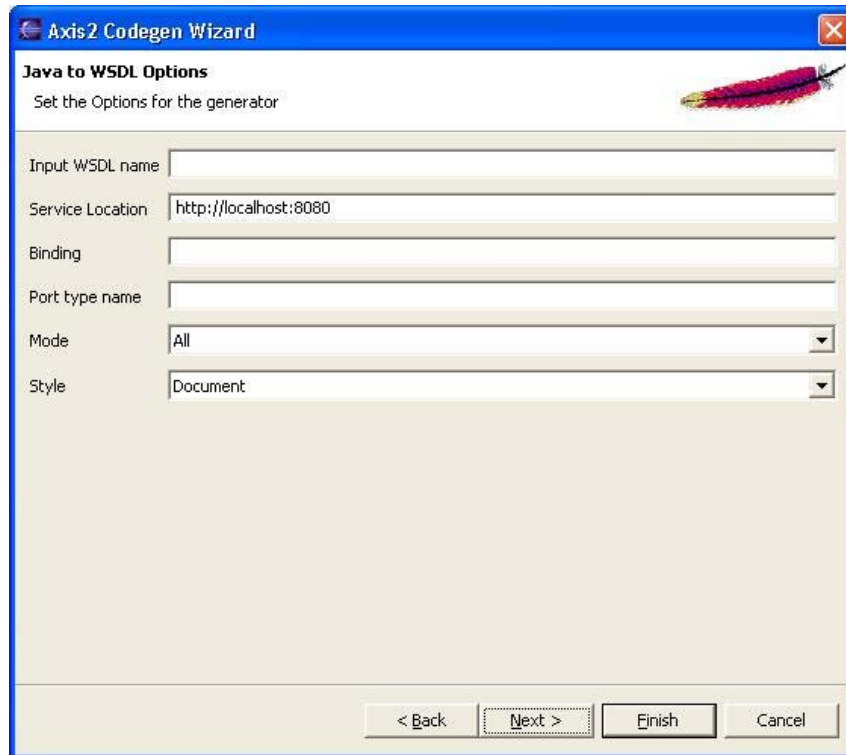
WSDL Generation

It is logical to start the development of the web service by first writing the WSDL, but in some situations the reverse happens, where the implementation is already written and the user may want to generate the relevant web service artifacts. This can be done by using the java2WSDL tool which is also provided as an Eclipse plug-in.

In the first screen of the Code Generation tool the user should select the “Generate a WSDL from a Java Source option” which will guide the user through the WSDL generation process.



In the next step the user should point to the implementation class and the application will pick up the probable methods that can be possible candidates for the Web service operations. The user can select the methods of the implementation class that he/she desires to be exposed as web service operations.



Finally the user can select the output location and the WSDL file name and then Finish so that the WSDL file will be generated.

Developing Extensions – Supporting Custom MEP

Axis 2 architecture has provided for a pluggable extension to interface the Axis2 Engine and the service implementation. In other words the Axis2 engine uses a pluggable component called a “Message Receiver” handle the interactions with the actual service implementation. From the point of Axis2 engine an incoming message gets its headers processed by the handlers in the “In” chain(or pipe) and the payload or the body of the message will be delivered to a “Message Receiver” who knows what to do with it. Axis2 supports first class support for the ability to plug-in new or different Message Receivers through deployment mechanisms.

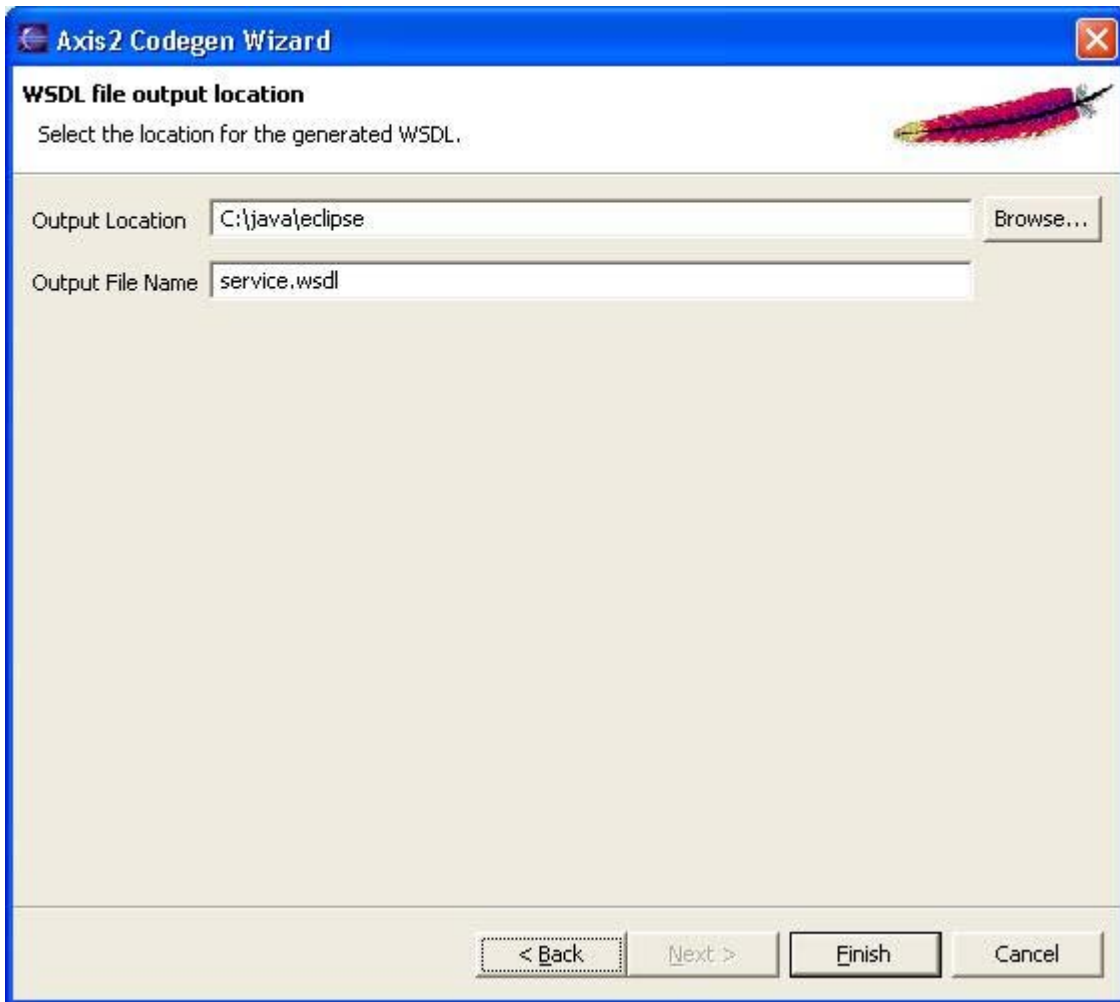
Message Receiver’s behavior is based on the Message Exchange pattern and the fact that the service invocation is blocking or non-blocking. Message receivers can fully control the synchronous or asynchronous behavior of the service invocation and same service can be invoked synchronously or asynchronously by associating the service with relevant message receivers. Since it is a valid requirement to change the programming model to synchronous or to asynchronous, the Axis2 deployment provides room to deploy and bind specific Message Receivers during the service deployment. In addition to this Axis2 ships most frequently used Message Receivers such as InOutSyncMessageReceiver, InOutAsyncMessageReceiver, etc.

Axis2 code generation tool generates a Message Receiver for each time code generation happens for a particular service and this enables the service implementation to be tightly bound, where necessary, by way of a java method invocation rather than reflectively calling the operations of the service invocation(Which the generalized Message Receivers such as InOutSyncMessageReceiver do.).

Further the fact that the actual web service implementation can be decoupled with the Axis2 engines using a Message Receiver will enable the users to write the service implementation in languages other than java that compiles to the java platform. There are few examples for such providers in the source distribution.

The Message exchange patterns are not frequently used extension, thus axis on the run does not support the deployment of a service that has a totally new Message Exchange Pattern. Axis2 has a well defined context hierarchy and among them the OperationContext handles the context with respect to different Message Exchange Patterns. Current implementation of the OperationContext supports all eight Message Exchange Patterns defined in the WSDL 2.0 specification.

To introduce a new MEP other than the ones that are defined in the WSDL 2.0 specification, One must extend the org.apache.context.OperationContext that will be capable of identifying where does a given message fits in the given Message Exchange pattern. Then the user needs to write a new Message receiver that knows what to do with all the incoming messages of the given MEP and how those will get consumed by the service implementation and how and when would the results or out messages should be sent out. Though there is no on the run mechanism in Axis2 to adapt to a completely new MEP other than those of which are defined in the WSDL 2.0 specification, it can be easily extended to make it work properly for any given Message Exchange Pattern.



RESTful Web Services Support

Axis2 can be configured as a REST Container and can be used to send and receive restful web services. The REST Web Services can be access in two ways, using HTTP GET and POST.

Doing REST web services with HTTP POST

REST support can be enabled in the Server side by adding the following line to the axis.xml file.

```
<parameter name="enableREST" locked="xsd:false">true</parameter>
```

But it act both as a REST endpoint as well as a SOAP endpoint. When a Message is received if the content type is text/xml and the SOAP Action Header is missing the

Message is considered as a RESTful Message. Else they are treated as usual SOAP Messages.

On sending a message out, the fact that the message is RESTful or not, can be decided from the client API or by deployment descriptor of the client.

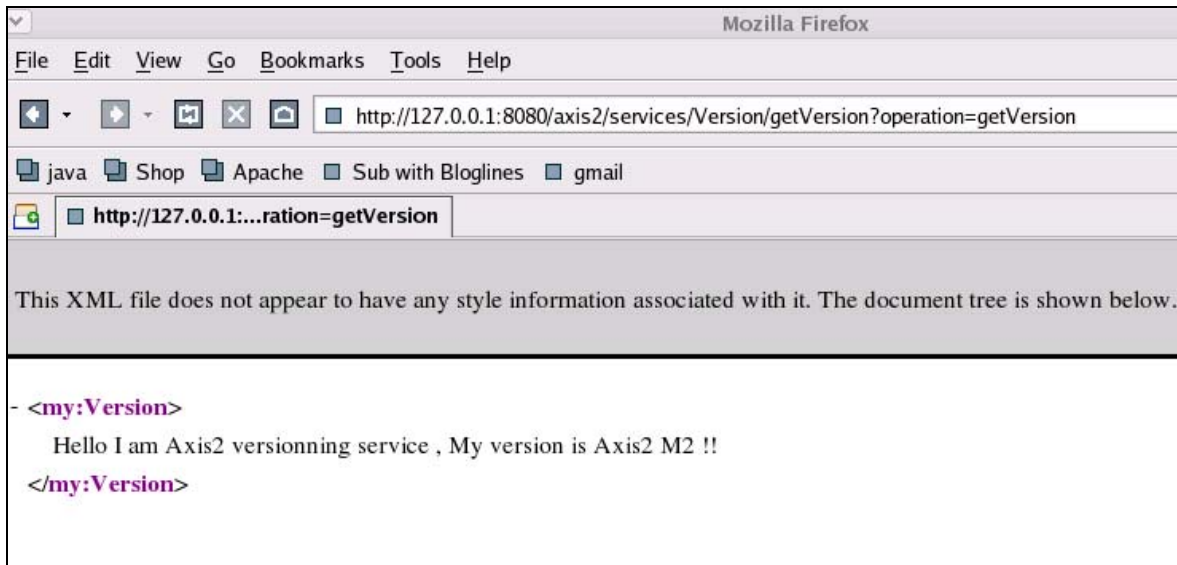
- By adding an entry in the client.xml file similar to that of the axis.xml.
- Through client API e.g.
`call.set(Constants.Configuration.DO_REST, "true")`

Access a REST Web Service Via HTTP GET

Axis2 lets the user to access Web Service that has simple type parameters via the HTTP GET. For example following URL requests the version service Via HTTP GET. But the Web Services arrived via GET assumes REST. Other parameters are converted in to the XML and put them in to the SOAP Body.

`http://127.0.0.1:8080/axis2/services/Version/getVersion?operation=getVersion`

Result can be viewed in the browser as shown below.



A GET request should fulfill following conditions

1. All the parameters the Web Service should accept MUST be simple types (e.g. String, int, float, long)
2. There must be a parameter called operation that states the name of the operation

For an example request

`http://127.0.0.1:8080/axis2/services/Version/getVersion?operation=getVersion` will be converted to the following SOAP Message for processing by Axis2

```
<soapenv:Envelope
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/03/addressing">
  <soapenv:Body>
    <axis2:getVersion
      xmlns:axis2="http://ws.apache.org/goGetWithREST" ></axis2:getVersion>
    </soapenv:Body>
  </soapenv:Envelope>
```

MTOM Support

Users may need to transmit a SOAP message together with attachments in various like images, drawings, xml docs, etc. Such data are often in a particular binary format. Traditionally, two techniques for dealing with opaque data in XML have been used, namely; "by value" or "by reference." The former is achieved by embedding opaque data as element or attribute content. XML supports opaque data as content through the use of either base64 or hexadecimal text encoding. It is well-known that base64 encoded data expands by a factor of 1.33x original size, and that hexadecimal encoded data expands by a factor of 2x, assuming an underlying UTF-8 text encoding. Also of concern is the overhead in processing costs (both real and perceived) for these formats, especially when decoding back into raw binary. When comparing base64 decoding to a straight-through copy of opaque data, the throughput of at least one popular programming system decreased by a factor of 3 or more.

Resources

- Apache Axis2 official site (<http://ws.apache.org/axis2>)