Wise-core Programmer Guide (version 0.9)

1. What is wise-core

It is a library to simplify web service invocation from a client point of view aims to be provide a near zero-code solution to find and parse wsdls, select service and endpoint and call operations mapping user defined object model to JAX-WS objects needed to perform the call.

In other words wise-core aims to provide web services client invocation in a dynamic manner.

It’s matter of fact that wsconsume tools is great for java developer, generating needed stub class, but it introduces a new (or renewed ) level of coupling very similar to corba IDL. Generating statically webservice stub you are in fact coupling client and server.

So what is the alternative? Generate these stubs runtime and use dynamic mapping on generated stub.

How wise-core does perform this generic task? In a nutshell it generates classes on the fly using wsconsume runtime API, loading them in current class loader and uses them with Java Reflection API. What we add is a generic mapping API to transform an arbitrary object model in the wsconsume generated ones, make the call and map the answer back again to the custom model using Smooks. Moreover this is achieved keeping the API general enough to plug in other mappers (perhaps custom ones) to transform user defined object into JAX-WS generated objects.

Wise supports standard JAX-WS handlers too and a generic smooks transformation handler to apply transformation to generated SOAP messages; in the next future it will also support easy configuration/API to activate various WS-*.

The key to understand the Wise-core idea is to keep in mind it is an API hiding JAX-WS wsconsume tools to generate JAX-WS stub classes and providing API to invoke them in a dynamic way using mappers to convert your own object model to JAX-WS generated one.

One of the most important aspects of this approach is that Wise delegates everything concerning standards and interoperability to the underlying JAX-WS client implementation (JBossWS in the current implementation). In other words if an hand written webservice client using JBossWS is interoperable and respects standard, the same applies for a Wise-generated client! We are just adding commodities and dynamical transparent generation and access to JAX-WS clients, we are not rewriting client APIs, the well tested and working ones from JBossWS is fine for us 

2 . Who should read this guide

This guide is written for developers who would use Wise-core in their own application to call webservices.

This guide would be very useful also for programmers and architect using JBossESB and would better understand what is happening under the hood during a zero-code webserive invocation based on Wise.

3. API description

We are going to describe here our API, its goals and how it could be used in practice to simplify your webservice client development. Anyway we strongly suggest you to take a look at our javadoc as more complete reference for the API.

The core elements of our API are:

- WSDynamicClient: This is the Wise core class responsible for the invocation of the JAX-WS tools and that handles wsdl retrieval & parsing. It is used to build the list of WSService representing the services published in parsed wsdl. It can also be used to directly get the WSMethod to invoke the specified action on specified port of specified service. It is the base method for "one line of code invocation"
- WSService: represents a single service. It can be used to retrieve the current service endpoints (Ports).
- WSEndpoint: represents an Endpoint(Port) and has utility methods to edit username, password, endpoint address, attach handlers, etc.
- WSMethod: represents a webservice operation(action) invocation and it always refers to a specific endpoint. It is used for effective invocation of a web service action.
- WebParameter: holds single parameter's data required for an invocation
- InvocationResult: holds the webservice's invocation result data. Anyway it returns a Map<String, Object> with webservice's call results, eventually applying a mapping to custom objects using a WiseMapper
- WiseMapper: is a simple interface implemented by any mapper used within wise-core requiring a single method applyMapping.

All the elements mentioned above can be combined and used to perform web service invocation and get results. They basically support two kinds of invocation:

1. One line of code invocation: with this name we mean a near zero code invocation where developer who have well configured Wise just have to know wsdl location, endpoint and port name to invoke the service and get results. For a complete description and sample of this Wise usecase please refer to paragraph 5.1.
2. Interactively explore your wsdl: Wise can support a more interactive style of development exploring all wsdl artifact dynamically loaded. This kind of use is ideal for an interactive ser interface to call the service and is by the way how we are developing our web GUI. For a complete description and sample of this Wise usecase please refer to paragraph 5.2.

4. Configurations

Wise-core configurations are provided by JBoss standard inversion of control Micro Container. Wise-core read standard META-INF/jboss-beans.xml from your classpath. We suggest to put them into your jar, but also include a resources dir in your classpath is fine (and in fact is what is done in our samples). A typical jboss-beans.xml for Wise look like this one:

```
xml version=1.0 encoding=UTF-8?>

<deployment xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="urn:jboss:bean-deployer bean-deployer_1_0.xsd"
    xmlns="urn:jboss:bean-deployer">

    <bean name="WSKernelLocator"
        class="org.jboss.wise.core.jbossmc.KernelUtil">
        <property name="kernel">
            <inject bean="jboss.kernel:service=Kernel" />
        </property>
    </bean>

    <bean name="WiseClientConfiguration" class="org.jboss.wise.core.jbossmc.beans.WiseClientConfiguration" >
      <property name="defaultUserName">foo</property>
      <property name="defaultPassword">pwd</property>
      <property name="defaultTmpDeployDir">/home/oracle/temp</property>
      <property name="logConfig">resources/META-INF/wise-log4j.xml</property>
   </bean>

     <bean name="WSDynamicClientFactory" class="org.jboss.wise.core.jbossmc.beans.ReflectionWSDynamicClientFactory">
        <property name="config"><inject bean="WiseClientConfiguration"></inject> </property>
    </bean>

    <bean name="WSConsumer" class="org.jboss.wise.core.consumer.impl.jbosswsnative.WSImportImpl">
      <property name="keepSource">true</property>
      <property name="verbose">true</property>
   </bean>


</deployment>
```

Here is a description of bean configured in this file and their valid attribute:

4.1 WiseClientConfiguration

It contains all default values Wise could use for some common attribute during a call. Of course they can be overridden in the code.

Property	Description
defaultUserName	It's the default user name used to call webservice requiring authentication.
defaultPassword	I'ts the default password used to call webservice requiring authentication.
defaultTmpDir	It's the temporary directory used by Wise to keep downloaded wsdls and wsconsume generated classes file. **It's very important to understand that this directory is cleaned when Wise perform its first call after JVM starts.** **For this reason you have to provide different tmp dirs if you plan to run wise with more than one JVM.**
logConfig	It's the location of Wise's log4j config file.

4.2 WSDynamicClientFactory

It injects the client factory implementation, deciding in fact which implementation of Wise API you are using. At the moment we provide just this reflection based implementation, but in near future we will have also a Javassist ones. Of course here you can provide your own implementation and inject it.

4.3 WSConsumer

It injects the WSConsumer implementation, deciding which API you are using to consume the wsdl and generate JAX-WS client classes. At the moment we are providing a

org.jboss.wise.core.consumer.impl.jbosswsnative.WSImportImpl. This bean may be used also to set some general properties for class generation:

Property	Description
verbose	boolean value to decide if wsconsume have to use verbose option printing more information during compiling and running classes
keepSource	boolean value to decide if Wise has to keep source java files (and not only .class) in tmp directory. It may be useful to have a look at them in case of problem or debug.

5. Using Wise API

Wise can be used either as near zero code web service invocation framework or as an API to (interactively) explore wsdl generated objects and then perform the invocation through theselected service/endpoint/port.

The first approach is very useful when Wise is integrated in a server side solution, while the second one is ideal when you are building an interactive client with some (or intense) user interaction.

By the way the first approach is the one that has been used while integrating Wise in JBossESB, while the second one is the base on which we are building or web based generic interactive client of web service (Wise-webgui).

5.1 One line of code invocation

A sample may be much more clear than a lot of explanation:

```
WSDynamicClient client = WSDynamicClientFactory.getInstance().getClient("http://127.0.0.1:8080/HelloWorldJDK6/HelloWorldW
WSMethod method = client.getWSMethod("HelloWorldWSJDK6Service", "HelloWorldJDK6Port", "sayHello");
HashMap<String, Object> requestMap = new HashMap<String, Object>();
requestMap.put("toWhom", "SpiderMan");
InvocationResult result = method.invoke(requestMap, null);
```

I can already hear you saying: "hey, you said just 1 line of code, not 5!!". Yes, but if you exclude lines 3 and 4 where we are constructing a Map to put in request parameters that are normally build in other ways from your own program, you can easily compact the other 3

lines in just *one line of code invocation*. By the way keeping 3 lines of code makes the code more readable, but we would remark that conceptually you are writing a single line of code.

You can find a running sample called ***HelloWorld*** using exactly this code in our samples directory (both for JDK5 and JDK6).

5.2 Interactively explore your wsdl objects

Here too an example would be good:

```
WSDynamicClient client = WSDynamicClientFactory.getInstance().getClient("http://127.0.0.1:8080/InteractiveHelloWorldJDK6/Int
Map<String, WSService> services = client.processServices();
System.out.println("Available services are:");
for (String key : services.keySet()) {
   System.out.println(key);
}
System.out.println("Selecting the first one");
Map<String, WSEndpoint> endpoints = services.values().iterator().next().processEndpoints();
System.out.println("Available endpoints are:");
for (String key : endpoints.keySet()) {
   System.out.println(key);
}
System.out.println("Selecting the first one");
Map<String, WSMethod> methods = endpoints.values().iterator().next().getWSMethods();
System.out.println("Available methods are:");
for (String key : methods.keySet()) {
   System.out.println(key);
}
System.out.println("Selecting the first one");
WSMethod method = methods.values().iterator().next();
HashMap<String, Object> requestMap = new HashMap<String, Object>();
requestMap.put("toWhom", "SpiderMan");
InvocationResult result = method.invoke(requestMap, null);
```

You can find a running sample called ***IntercativeHelloWorld*** using exactly this code in our samples directory.

5.3 Go Dynamic, use Groovy

Yes, Wise supports Groovy. We have a sample in our code base invoking HelloWorld from a Groovy script. We also have an example using closure to invoke a web service through Wise, just changing the input parameters to demonstrate how much Wise can be efficient

caching all artifacts and having performance very similar to what you get with native use of JAX-WS client classes after the first invocation.

But supporting Groovy means much more than this. Groovy is an high dynamic language and so it can be used to inspect dynamically generated classes. In other words with Groovy you can avoid to use Wise mappers, directly and dynamically using our generated classes. We will provide much more documentation and samples in next releases of Wise; until then, just use your creativity, and contribute reporting your experiments in the user forums.

But why using Wise from Groovy? Haven't Groovy its own web service client? Yes, but with Wise you will have a client for Groovy certified for JAX-WS standard, supporting authentication, MTOM, standard JAX-WS handlers, and in next future WS-*.

6. WiseMapper: from your own object model to the generated JAX-WS model and vice versa

The core idea of Wise is to permit users to call webservice using their own object model, loading at runtime (and hiding) the JAX-WS generated client classes. Of course developers who have a complex object model and/or using a webservice with a complex model have to provide some kind of mapping between them.

This task is done by applying a WiseMapper which is responsible of this mapping. Mappers are applied both to WSMethod invocation and results coming from InvocationResult. Of course the first one maps from the custom model to the JAX-WS one, while the second takes care of the other way.

Wise provide a Smooks based mapper, but it should be easy to write your own.

6.1 SmooksWiseMapper

For any information about Smooks and its own configuration file please refer to its own documentation

When writing a Smooks config file to use with Wise, you need to consider that the object model generated by JAX-WS tools isn't available at compile time since Wise generates it at runtime. Even if in most cases you can infer the JAX-WS generated classes and properties names from the wsdl document, sometimes it might be useful to set keepSource = true in Wise's jboss-beans.xml to have the opportunity to take a look to generated classes.

We provide a sample demonstrating use of smooks mapper in our code base named ***usingSmooks***.

6.2 Writing your own Mapper

You just have to implement WiseMapper interface.

7. Adding standard JAX-WS handlers

WSEndPoint class has a method to add standard JAX-WS handlers to the endpoint. Wise takes care of the handler chain construction and ensures your client side handlers are fired during any invocations.

We provide two standard handlers: one to log the request/response SOAP message for any invocation and one applying Smooks transformation on your SOAP content.

7.1 Logging Handler

This simple SOAPHandler will output the contents of incoming and outgoing messages. It checks the MESSAGE_OUTBOUND_PROPERTY in the context to see if this is an outgoing or incoming message. Finally, it writes a brief message to the print stream and outputs the message.

7.2 Smooks Handler

A SOAPHandler extension. It applies Smooks transformations on SOAP messages. The transformation can also use freemarker, using provided javaBeans map to get values. It can apply transformation on inbound messages only, outbound ones only or both, depending on setInBoundHandlingEnabled(boolean) and setOutBoundHandlingEnabled(boolean) methods.

Take a look at our unit test org.jboss.wise.core.mapper.SmooksMapperTest in test-src directory.

7.3 Adding your own Handler

Since Wise's handlers are JAX-WS standard handlers, you just have to provide a class that implements SOAPHandler<SOAPMessageContext>

8. Requirements and dependencies

The current implementation depends on JBossWS. Take a look to our samples to better understand which libraries are required.

A special note is needed about some JAX-WS jars included, since it have been slightly modified to make Wise and JBossWS client work properly.

8.1 Special Note on jaxws-rt and jbossws-native-jaxws jars on JDK5

The problem is caused because JBossWS uses 2 different jaxws provider impl (one for tooling, the Sun's RI, and one for everything else, it's own implementation) Of course due to the spi mechanism of loading provider, you can't have both in classpath since the one loaded first takes prevalence on the other. Chaper 6.2.1 of JAX-WS 2.0 specification defines also the algorithm to load this Provider:

```
/**
   * This method uses the algorithm below using the JAXWS Provider as an example.
   *
   * 1. If a resource with the name of META-INF/services/javax.xml.ws.spi.Provider exists, then
   * its first line, if present, is used as the UTF-8 encoded name of the implementation class.
   *
   * 2. If the ${java.home}/lib/jaxws.properties file exists and it is readable by the
   * java.util.Properties.load(InputStream) method and it contains an entry whose key is
   * javax.xml.ws.spi.Provider, then the value of that entry is used as the name of the implementation class.
   *
   * 3. If a system property with the name javax.xml.ws.spi.Provider is defined, then its value is used
   * as the name of the implementation class.
   *
   * 4. Finally, a default implementation class name is used.
   */
```

For this reason, in order to use both implementations, Wise requires modified jar artifacts without the META-INF/services/javax.xml.ws.spi.Provider file; then the Provider implementation to be used is set in WSImporterImpl through a System property.

In practise this means you have to use the version of this two libraries that we ship with Wise and not those provided by JBossWS or other sources.The problem isn't present with JDK6 since it uses its internal JAX-WS implementation (coming with the JDK) which uses the same provider used by wsconsume tool avoiding any kind of conflict.

[Show »]

Stefano Maestri - 02/Oct/08 03:37 PM The problem is caused because jbossws use 2 different jaxws provider impl (one for tooling, the sun's RI, and one for all the rest, it's own implementation) Of course for the spi mechanism of loading provider it can't use both in classpath since the first one loaded will take prevalence on the other. Chaper 6.2.1 of JAX-Ws 2.0 specification define also the algorithm to load this Provider: /** * This method uses the algorithm below using the JAXWS Provider as an example. * * 1. If a resource with the name of META-INF/services/javax.xml.ws.spi.Provider exists, then * its first line, if present, is used as the UTF-8 encoded name of the implementation class. * * 2. If the ${java.home}/lib/jaxws.properties file exists and it is readable by the * java.util.Properties.load(InputStream) method and it contains an entry whose key is * javax.xml.ws.spi.Provider, then the value of that entry is used as the name of the implementation class. * * 3. If a system property with the name javax.xml.ws.spi.Provider is defined, then its value is used * as the name of the implementation class. * * 4. Finally, a default implementation class name is used. */ So it's matter of fact that to work with both implementation wise needs to have modified jars without the META-INF/services/javax.xml.ws.spi.Provider file an then change System property in WSImporterImpl. It's all done on SF's svn R358