Wise-core Programmer Guide (version 1.1)

What's new

In this version we have totally changed the way to configure Wise Core. Why? We have moved all configuration to the code, passing all (in fact few) config during WSDynamicClient instantiation. In other words we have removed our dependency on JBoss MK, to be more flexible and compatible with JBoss ESB and JBoss AS 5. Please refer to section 4 for further details

Another important change is how we are managing multithread use of our API, providing a thread pool of endpoints, permitting a concurrent safe and performant invocation of WSMethod even if the used JBossWS native stack doesn't provide a threadsafe access to generated classes (see WISE-25) . In general a lot of work have been done to ensure better performance and to avoid problems in concurrent use of our API.

Finally the project have been mavenized and a lot of unit, integration and stress tests have been integrated in our build lifecycle.

For more details, please have a look at our JIRA

You can also refer to the previous versions of this document.

version 1.0

1. What is wise-core

It is a library to simplify web service invocation from a client point of view; it aims at providing 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 required to perform the call.

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

It's a matter of fact that wsconsume tools is great for java developer, generating required stub classes, but it introduces a new (or renewed ) level of coupling very similar to Corba IDL. When statically generating webservice stubs, you are in fact coupling client and server.

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

How does wise-core perform this generic task? In a nutshell it generates classes on the fly using wsconsume runtime API, loading them in current class loader and using 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; currently there's also a basic support for some WS-* specifications, which will be futher developed in the next future.

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 to 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 like to use Wise-core in their own application to call webservices.

This guide would also be very useful to programmers and architects using JBoss ESB.

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 a 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 and 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". Each single instance of this class is responsible of its own temp files, smooks instance and so on. It is importanto to call close() method to dispose resources and clean temp directories.
- 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 setting properties on WSDynamicClientBuilder during WSDynamicClient instance creation

Properties and their purpose are well documented in WSDynamicClientBuilder javadoc. Here is and example on how to leverage the builder to get the WSDynamicClient:

```
[...]
URL wsdlURL = new URL(getServerHostAndPort() + /wsaandwsse/WSAandWSSE?wsdl);

    WSDynamicClientBuilder clientBuilder = WSDynamicClientFactory.getJAXWSClientBuilder();
    WSDynamicClient client = clientBuilder.tmpDir(target/temp/wise).verbose(true).keepSource(true)
        .securityConfigUrl(WEB-INF/wsaandwsse/jboss-wsse-client.xml).securityConfigName(Standard WSSecurity Client)
        .wsdlURL(wsdlURL.toString()).build();
[...]
```

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:

```
WSMethod method = client.getWSMethod("HelloService", "HelloWorldBeanPort", "echo");
Map<String, Object> args = new java.util.HashMap<String, Object>();
args.put("arg0", "from-wise-client");
InvocationResult result = method.invoke(args, null);
```

I can already hear you saying: "hey, you said just 1 line of code, not 4!!". Yes, but if you exclude lines 2 and 3 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 2 lines in just *one line of code invocation*. By the way keeping 2 or 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 integration test called WiseIntegrationBasicTest using exactly this code. Of course thre are few more lines of code to create client and to make assertion on the results, but trust us they are very few line of code.

5.2 Interactively explore your wsdl objects

Here too an example would be good:

```
try {
        WSDynamicClientBuilder clientBuilder = WSDynamicClientFactory.getJAXWSClientBuilder();
        WSDynamicClient client = clientBuilder.tmpDir("target/temp/wise").verbose(true).keepSource(true)
          .wsdlURL("http://127.0.0.1:8080/InteractiveHelloWorld/InteractiveHelloWorldWS?wsdl").build();
        Map<String, WSService> services = client.processServices();
        System.out.println("Available services are:");
        for (String key : services.keySet()) {
            System.out.println(key);
        }
        System.out.println("Selectting 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("Selectting 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("Selectting 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);
        System.out.println(result.getMapRequestAndResult(null, null));
        System.out.println(result.getMapRequestAndResult(null, requestMap));
        client.close();
    } catch (Exception e) {
        e.printStackTrace();
    }
```

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 of course WS-*.

5.4 Go Dynamic, use Ruby

Yes, Wise supports Ruby too (through JRuby). We have a sample in our code base invoking HelloWorld from a Ruby 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, as for Groovy, supporting Ruby means much more than this. With a so muchhigh dynamic language you could inspect dynamically generated classes. In other words with Ruby 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 Ruby? Haven't Ruby its own web service client? Yes, but with Wise you will have a client for Ruby certified for JAX-WS standard, supporting authentication, MTOM, standard JAX-WS handlers, and of course 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. Extensions (WS-* and MTOM)

We tried to respect the Wise's easy to use approach also designing APIs to enable and use most common extensions (MTOM and WS-*) in Wise.

There are in Wise's API an interface (WSExtensionEnabler) defining a WSExtension to be enabled on an endpoint using wise-core client APIs. The basic idea is to add all WSExtension you want to enable to a WSEndpoint using addWSExtension method. WSExtension implementation are meant to be pure declarative class delegating all their operations to a "visitor" class injected into the system with IOC Different Visitors implement EnablerDelegate and have to take care to implement necessary steps to implement various WSExtension for the JAXWS implementation for which they are supposed to work.

A snippet of use may clarify how much simple it can be from API's user poit of view

```
WSMethod method = client.getWSMethod("MTOMWSService", "MTOMPort", "sayHello");
method.getEndpoint().addWSExtension(new MTOMEnabler(client));
```

It enable mtom on *MTOMPort* endpoint.

The same can be applied for example to easily enable both WSSecurity and WSAddressing on endpoint. Take a look to this amazing snippet:

```
WSDynamicClientBuilder clientBuilder = WSDynamicClientFactory.getJAXWSClientBuilder();
WSDynamicClient client = clientBuilder.tmpDir("target/temp/wise").verbose(true).keepSource(true)
    .securityConfigUrl("WEB-INF/wsaandwsse/jboss-wsse-client.xml").securityConfigName("Standard WSSecurity Client")
    .wsdlURL(wsdlURL.toString()).build();

WSMethod method = client.getWSMethod("WSAandWSSEService", "WSAandWSSEImplPort", "echoUserType");
```

```
method.getEndpoint().addWSExtension(new WSSecurityEnabler(client));
method.getEndpoint().addWSExtension(new WSAddressingEnabler(client));
method.getEndpoint().addHandler(new LoggingHandler());
```

These few lines of code are all what you need to enable WS-SE and WS-A!

Take a look to our samples and integration tests to get some complete working code about (section 10)

We plan to support in the same manner also some other WS-* in future versions.

9. JAX-RS Client Support

WISE has a preliminary JAX-RS client support to allow writing clients that can invoke JAX-RS services. The code snippet below showed how this works:

```
// Sent HTTP GET request to query customer info
System.out.println("Sent HTTP GET request to query customer info");
RSDynamicClient client = WSDynamicClientFactory.getInstance().getJAXRSClient(
                        "http://localhost:9000/customerservice/customers/123",
                        RSDynamicClient.HttpMethod.GET, null,
                        "application/xml");
InvocationResult result = client.invoke();
String response = (String) result.getResult().get(InvocationResult.RESPONSE);
System.out.println(response);


// Sent HTTP PUT request to update customer info
System.out.println("\n");
System.out.println("Sent HTTP PUT request to update customer info");
client = WSDynamicClientFactory.getInstance().getJAXRSClient(
                "http://localhost:9000/customerservice/customers",
                RSDynamicClient.HttpMethod.PUT, "application/xml",
                "application/xml");
JaxrsClient jaxrsClient = new JaxrsClient();
InputStream request = jaxrsClient.getClass().getResourceAsStream("resources/update_customer.xml");
result = client.invoke(request, null);
response = (String) result.getResult().get(InvocationResult.RESPONSE);
int statusCode = ((Integer) result.getResult().get(InvocationResult.STATUS)).intValue();
System.out.println("Response status code: " + statusCode);
System.out.println("Response body: ");
System.out.println(response);
```

```
// Sent HTTP POST request to add customer
System.out.println("\n");
System.out.println("Sent HTTP POST request to add customer");
client = WSDynamicClientFactory.getInstance().getJAXRSClient(
                "http://localhost:9000/customerservice/customers",
                RSDynamicClient.HttpMethod.POST, "application/xml",
                "application/xml");
request = jaxrsClient.getClass().getResourceAsStream("resources/add_customer.xml");
result = client.invoke(request, null);
response = (String) result.getResult().get(InvocationResult.RESPONSE);
statusCode = ((Integer) result.getResult().get(InvocationResult.STATUS)).intValue();


System.out.println("Response status code: " + statusCode);
System.out.println("Response body: ");
System.out.println(response);
```

9.1 Create RSDynamicClient

The main interface you need to use is the getJAXRSClient method from WSDynamicClientFactory. See below:

```
  /**
   * Return an instance of RSDynamicClient taken from cache if possible, generate and initialise if not.
   *
   * @param endpointURL
   * @param httpMethod
   * @param produceMediaTypes
   * @param consumeMediaTypes
   * @return an instance of {@link RSDynamicClient} already initialized, ready to be called
   */
  public RSDynamicClient getJAXRSClient( String endpointURL,
                                RSDynamicClient.HttpMethod httpMethod,
                                String produceMediaTypes,
                                String consumeMediaTypes ) {
     ......

  }
```

9.2 Return responses

Get the response body:

```
InvocationResult result = client.invoke();
String response = (String) result.getResult().get(InvocationResult.RESPONSE);
System.out.println(response);
```

Get the response code:

```
InvocationResult result = client.invoke(request, null);
String response = (String) result.getResult().get(InvocationResult.RESPONSE);
int statusCode = ((Integer) result.getResult().get(InvocationResult.STATUS)).intValue();
```

Get the response headers:

```
Not supported yet in this version
```

10. Requirements and dependencies

The current implementation depends on JBossWS-native stack. We will support in next releases also other stacks, and it will be done with our SpiLoader mechanism. For other dependencies please have a look to our maven's dependecy tree:

```
mvn dependency:tree
```

11. How tou use it

We have a set of samples demonstrating how to use wise in a standalone application. We have two different kind of samples: integration tests running in our test suite (with some multithread stress tests too) and pure ant samples.

Please refer to JBossESB quickstar samples for Wise/ESB integration example of use.

11.1 Integration tests

There is a maven module of the project called integration which runs our integration and stress tests.

Of course the sources for this module are available along with all the others. Feel free to join us on user forum and ask any question.

Both integration and stress tests need a running application server (JBoss 4.x or 5.x) with a JBossWS-Native stack deployed on it (other stacks should works too on server side, but this is our tested configurations).

To run the integration test just run (from main directory):

```
mvn integration-test -Djboss.bind.address=localhost
```

To run stress tests:

```
mvn integration-test -Djboss.bind.address=localhost -Pstress.tests
```

Remember you can also play with number of parallell threads editing integration/pom.xml file under stress.tests profile section.

11.2 Samples

Plese refer to sample specific direcotory README.txt for a full description of the sample itself.

Any directory, except for *lib* and *ant*, contains a single example. The directory's name suggests which kind of test you will find in. Only in case we think example needs further explanations, you will find a local README.txt inside its directory.

lib directory contains library referred by examples. *ant* directory contains build.xml imported from all examples' build.xml.

How to run examples?

1. Enter in specific example directory 

2. Edit resources/META-INF/jboss-beans.xml and change properties according to your environment (i.e defaultTmpDeployDir) if needed.

3. Edit resources/META-INF/wise-log4j.xml and change properties according to your environment if needed.

4. Edit build.properties changing "JBossHome" and "ServerConfig" property to point to your JBossAS instance

5. Start your JBossAS instance (of course it have to provide JBossWS)

6. type "ant deployTestWS" to deploy server side content (aka the ws against example will run)

7. type "ant runTest" to run the client side example

8. type "ant undeployTestWS" to undeploy server side content

9. Have a look to the code.

If something changes for a specific example you will find instructions on local README.txt

have fun.