Wise-core Programmer Guide (version 2.0.1 +)

What's new (2.0.x)

Wise-core 2.0.1 is a minor release including bug fixes as well as few new features over 2.0.0 version:

- added functionality for generating request message preview
- added new "tree-like" view for dealing in a detyped way with parameters and results

What's new (2.1.x)

Wise-core 2.1.0 is a minor release including bug fixes as well as few new features over 2.0.x versions:

- support for JDK 1.8
- support for WildFly container
- Smooks dependency is not mandatory anymore
- Arquillian based testsuite

What is wise-core

Wise-core is a library to simplify web service invocation from a client point of view; it aims at providing a near zero-code solution to parse wsdls, select service/port and call operations by mapping a user defined object model to the JAX-WS objects required to perform the call.
In other words wise-core aims at providing web services client invocation in a dynamic manner.

While basic JAX-WS tool for wsdl-to-java generation (like wsconsume) are great for most Java developer usecases, the generated stub classes kind of introduce a new (or renewed :)) level of coupling very similar to Corba IDL; by generating statical webservice stubs you actually couple client and server.
So what is the alternative?
Generating stubs at runtime and using 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 / Apache CXF in the current implementation). In other words if an hand written webservice client using JBossWS is interoperable and respects standards, 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 with us.

API description

Below is a description of Wise-core API, its goals and how it can 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.
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 interface to call the service and is by the way how we are developed our web GUI.

Configuration

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).wsdlURL(wsdlURL.toString()).build();
[...]
```

Wise API usage

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 built our web based generic interactive client of web service (Wise-webgui).

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.

Interactive wsdl exploration

```
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.

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

The core idea of Wise is to allow 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 are 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 for this mapping. Mappers are applied on both the WSMethod invocation and the results coming from InvocationResult. Of course the first one maps the custom model to the JAX-WS one, while the second one takes care of the other way.
Wise provides a Smooks based mapper, but it should be easy to write your own.

Tree view

The section above on interactive wsdl exporation explains how users can figure out the structure of described services and their parameter and result types. As previously explained, users can then leverage WiseMappers (e.g. based on Smooks) to bridge the generated model to a custom object model. However, mappers do not really help a lot when the goal is programmatically testing endpoints. For this reason, a tree model (think about e.g. the DOM tree approach) can be generated from the Wise core model to describe each parameter type of a given WSMethod. Each Element of the tree would allow getting children and setting values (for leaves only). An ElementBuilderFactory can be used to get available ElementBuilder implementation which comes with *buildTree* method for generating the Element tree given a Type (and Object of that type, if available, to read actual leaf values).
Users can hence effectively convert the request parameter types of WebParameter into Element trees, easily populate them, write the trees to objects (*Element::toObject()*) and perform the invocation using them.
Similarly, the response object can be converted in a result Element tree. Here is an example:

```
//select a method and the parameter(s) to deal with...
```

```
WSMethod method = client.getWSMethod("RegistrationServiceImplService",
"RegistrationServiceImplPort", "Register");
Map<String, ? extends WebParameter> pars = method.getWebParams();
WebParameter customerPar = pars.get("Customer");

//browse the paramter structure and eventually populate the parameter...
ElementBuilder builder =
ElementBuilderFactory.getElementBuilder().client(client).request(true).useDefautValuesForNullLeaves(tru
Element customerElement = builder.buildTree(customerPar.getType(), customerPar.getName(),
null, true);
customerElement.getChildByName("id").setValue("1234");
customerElement.getChildByName("name").getChildByName("firstName").setValue("Foo");
customerElement.getChildByName("name").getChildByName("lastName").setValue("Bar");
customerElement.getChildByName("name").getChildByName("middleName").setValue("The");

//convert back the tree to an Object and perform the invocation...
Map<String, Object> args = new java.util.HashMap<String, Object>();
args.put(customerElement.getName(), customerElement.toObject());
InvocationResult result = method.invoke(args);
```

The parameters type info is completely hidden to the user, who basically ends up setting string values for leaves elements. Strings are parsed into proper primitives (and wrappers) depending on the actual parameter type. Wise default Element impl is currently able to convert values to String, Character, all numerical types, QName, XMLGregorianCalendar and Duration classes.
So, leveraging Wise, a user can invoke / test an endpoint by browsing its contract and setting parameters in a fully dynamic and detyped way, while still being sure the generated SOAP request is compliant with the contract requirements and constraints.

Adding standard JAXWS 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.

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.

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.

Adding your own Handler

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

Request message preview

The WSMethod class comes with a *writeRequestPreview* method which allows generating the current request message preview for the specified parameters. No message actually goes on the wire.

Requirements and dependencies

Wise-core depends on JBossWS, in particular on JBossWS-CXF stack, given that's what is used on JBoss AS 7 / WildFly. You can have a look at the Maven project dependencies by running:

```
mvn dependency:tree
```

Getting started

First of all, make sure to have the JBoss.org Maven repository properly setup.
The relevant artifacts for the current release are available on the public repository.

Setting Wise dependencies in your application

The binary distribution contains the required dependency libraries to simply setting classpath for an application willing to use Wise.
Maven based application should set dependencies on the ***org.jboss.wise:wise-core-cxf*** artifact. If the application is meant to run in-container on JBoss AS 7 / WildFly, you should evaluate excluding some of Wise core dependencies from the target application deployment archive, as they are already available on the application server. Consider having a look at the *incontainer* sample included in the distribution or at the *Wise WebGUI* for details on how to achieve that by using a *jboss-deployment-structure.xml* descriptor.

Building from sources

Should you want to build from source and install artifacts into the local Maven repository, that's done as for any other Maven 3 project, by running:

```
mvn clean install
```

Testsuite

You can run the integration testsuite as follows:

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

Where ofcourse 'localhost' is the host for your currently running JBoss AS / WildFly.

Wise-core 2.0 has been fully tested on JBoss AS 7.1.1.Final.
Wise-core 2.1.0 has been fully tested on WildFly 8.0.0.Final, 8.1.0.Final, 8.2.1.Final, 9.0.2.Final and 10.0.0.Final.

Samples

Plese refer to sample specific direcotory README.txt for a full description of the samples and how to run them.