

Getting Started with Metro

Getting Started with Metro

Abstract

This series of articles will familiarize developers with the basics of Metro and introduce some of its main features.

Table of Contents

Building a Simple Metro Application 1

Enabling Advanced Features in a Web Service Application 7

Building a Simple Metro Application

Abstract

The intent of this article is to demonstrate the steps required to build a web service starting both from Java code and from a WSDL document, to deploy that application into a web container, and to build a corresponding web service client application. In this example, the resulting application is portable across JAX-WS 2.0 implementations and do not use any Metro-specific technologies. It is intended as a baseline from which to develop your understanding of the larger Metro stack.

Table of Contents

1. Overview	1
2. Environment Configuration Settings	1
3. Building a JAX-WS Web Service	2
4. Deploying the Web Service to a Web Container	4
5. Building a JAX-WS Web Service Client	5
6. Running the Web Service Client	5
7. Undeploying a JAX-WS Web Service	6

1. Overview

Supporting code samples are included to demonstrate building a JAX-WS web service in the Metro environment. The examples show how to develop a web service both starting from Java source code and starting from an existing WSDL document. For both scenarios, it shows how to develop a corresponding client application from the web service's WSDL document. The examples can be found below:

- From-Java example [[download/wsit-jaxws-fromjava.zip](#)]
- From-WSDL example [[download/wsit-jaxws-fromwsdl.zip](#)]

As mentioned above, these examples do not enable any Metro-specific technologies. However, the following article in this series, [Enabling Advanced Features in a Web Service Application](#), builds on the information presented in this document. It explains configuring a web service and its client to enable advanced features available in Metro.

2. Environment Configuration Settings

2.1. Prerequisites

These series of articles require the following software to be installed on your system:

- JDK 6.0 Update 29 [<http://www.oracle.com/technetwork/java/javase/downloads/index.html>] or later,
- Apache Ant 1.6.5 [<http://ant.apache.org/>] or later,
- web container: either Glassfish v3.x [<http://glassfish.java.net/>] or Apache Tomcat 7.0 [<http://tomcat.apache.org/>]
- Metro Standalone Budle 2.x [<http://metro.java.net/>] (in case of using Apache Tomcat as a web container)

2.2. Adding WSIT (Metro) libraries into your web container

The following steps are required only if using Apache Tomcat as a web container (Glassfish v3 already contains Metro libraries): Unzip downloaded Metro Standalone Bundle and copy all `.jar` files from the `lib/` directory into `<tomcat-install-directory>/endorsed` (where `<tomcat-install-directory>` points to your Apache Tomcat installation directory). Also put a copy of the `servlet-api.jar` library (`<tomcat-install-directory>/lib`) into `endorsed/` `libs`.

2.3. Web Container "Listen" Port

The Java code and configuration files for the examples used in this document presume that the web container is listening on port 8080. Port 8080 is the default "listen" port for both GlassFish (`domain1`) and Tomcat. If you have changed the "listen" port, you will need to edit the example source files to account for that. The following is a list of the files which contain references to the "listen" port:

1. `wsit-jaxws-fromjava/src/fromjava/server/AddWebservice.java`
2. `wsit-jaxws-fromjava/etc/custom-schema.xml`
3. `wsit-jaxws-fromjava/etc/custom-client.xml`
4. `wsit-jaxws-fromjava/etc/build.properties`
5. `wsit-jaxws-fromwsdl/etc/custom-client.xml`
6. `wsit-jaxws-fromwsdl/etc/build.properties`

2.4. Web Container Home Directory

Before building and deploying the web service and its client, the home directory of the web container must be set either as an environment variable or as a property in the respective `build.xml` file.

Environment Variables

Assuming that you are running from the command-line, it is probably simplest to set the appropriate environment variable indicating the web container's "home" directory. For GlassFish, `AS_HOME` should be set to the top-level directory of the GlassFish installation. For Tomcat, `CATALINA_HOME` needs to be set to the Tomcat top-level directory.

Ant `build.xml` File

If you would rather not have to set the environment variable for each new terminal session, you can edit the `build.xml` file located at the top-level directory of each of the examples. There are two commented lines, one each for GlassFish (`as.home`) and Tomcat (`catalina.home`). Simply uncomment the appropriate line and edit the value for the directory name.

3. Building a JAX-WS Web Service

3.1. Starting from Java

One way to create a web service application is to start by coding the endpoint in Java. If you are developing your Java web service from scratch or have an existing Java class you wish to expose as a web service, this is the most direct path.

The web service is written as a normal Java class. Then the class and its methods that are to be exposed are annotated with specific web service annotations, `@WebService` and `@WebMethod`. The following code snippet shows an example:

```
@WebService
AddNumbersImpl {
    @WebMethod
    addNumbers( a, b) AddNumbersException {
        (a < 0 || b < 0) {
            AddNumbersException(,
                + a + + b);
        }
        a + b;
    }
}
```

If you are using GlassFish, the web service in the `wsit-jaxws-fromjava` example can be compiled and bundled simply by invoking:

```
ant server
```

If using Tomcat, the command-line would be:

```
ant -Duse.tomcat=true server
```

The `server` target in `build.xml` in turn invokes the tools necessary to process the annotations and compile the sources, and to bundle the Java class files and configuration files into a deployable web archive (WAR file). The WAR file will be `build/war/wsit-jaxws-fromjava.war`. The tools that were called by ant during this step are briefly described next.

The JAX-WS tool **apt** (annotation processing tool) processes the annotated source code and invokes the compiler itself, resulting in the class files for each of the Java source files. In the accompanying `fromjava` example, the **ant** target **build-server-java** in `build.xml` handles this portion of the process. Then the individual class files are bundled together along with the web service's supporting configuration files into the application's WAR file. It is this file that will be deployed to the web container in the next step. The **create-war** target takes care of this.

3.2. Starting from WSDL

Typically, you would start from WSDL to build your web service if you want to implement a web service that is already defined either by a standard or an existing instance of the service. In either case, the WSDL already exists. The JAX-WS **wsimport** tool will process the existing WSDL document, either from a local copy on disk or by retrieving it from a network address. An example of manually accessing a service's WSDL using a web browser is shown below as part of the section on verifying deployment.

As in the previous example, to build the `wsit-jaxws-fromwsdl` service for GlassFish, you can simply invoke:

```
ant server
```

Otherwise for Tomcat use:

```
ant -Duse.tomcat=true server
```

wsimport will take the WSDL description and generate a corresponding Java interface and other supporting classes. Then the Java compiler needs to be called to compile both the user's code and the generated

code. Finally, the class files are bundled together into the WAR file. The details can be seen in the `wsit-jaxws-fromwsdl` `build.xml` file as the **build-server-wsdl** and **create-war** targets.

4. Deploying the Web Service to a Web Container

As a convenience, invoking each sample's **server** target will build that web service's WAR file and immediately deploy it to the web container. However, in some situations, such as after undeploying a web service from its container, it may be useful to deploy the web service without rebuilding it.

For both the *from Java* and *from WSDL* scenarios described above, the resulting application is deployed in the same manner. However, the details of the deployment process differ slightly between the GlassFish and Tomcat web containers.

4.1. Deploying to GlassFish

For development purposes, it is simplest to use the "autodeploy" facility of GlassFish. To do so, copy your application's WAR file to the `autodeploy` directory for the domain to which you want to deploy. If you are using the default domain, `domain1`, set up by the GlassFish installation process, then the appropriate directory path would be `<glassfish-install-home>/domains/domain1/autodeploy`.

The `build.xml` file which accompanies this example has a deploy target for GlassFish. Invoke that target by running **ant** in the top-level directory of the respective examples, either `fromjava` or `fromwsdl`, as follows.

```
ant deploy
```

4.2. Deploying to Tomcat

Tomcat also has an "autodeploy" feature. That feature can be turned off but is enabled by Tomcat's "out of the box" configuration settings. Look in `<tomcat-install-directory>/conf/server.xml` for the value of `"autoDeploy"` if you are unsure. Assuming `"autoDeploy"` is enabled, then copying your application to `<tomcat-install-home>/webapps` is all that is necessary. Again, there is a target in the **ant** `build.xml` file which accompanies this sample. The deploy target can be invoked by running the following command in the example's top-level directory.

```
ant -Duse.tomcat=true deploy
```

4.3. Verifying Successful Deployment

One basic test to verify that the application has deployed properly is to use a web browser to retrieve the application's WSDL from its hosting web container. The following URLs would retrieve the WSDL from each of the two example services. If you are running your web browser and web container on different machines, you will need to replace "localhost" with the name of the machine hosting your web service. It is also worth ensuring that your web container is actually running at this point.

- `http://localhost:8080/wsit-jaxws-fromjava/addnumbers?wsdl`
- `http://localhost:8080/wsit-jaxws-fromwsdl/addnumbers?wsdl`

If the browser displays a pageful of XML, things are working. If not, check the web container logs for any error messages related to the the sample WAR you have just deployed. For GlassFish, the appropriate log can be found at `<glassfish-install-directory>/domains/<your-do-`

main>/logs/server.log. For Tomcat, the appropriate log file will be <tomcat-install-directory>/logs/catalina.out.

5. Building a JAX-WS Web Service Client

Unlike developing a web service provider, the process for creating a web service client application will always start with an existing WSDL document. This process is similar to the steps taken when building a service from an existing WSDL. Typically, the WSDL will be retrieved directly from a web service provider by the **wsimport** tool. Wsimport then generates the corresponding Java source code for the described interface. **javac**, the Java compiler, is then called to compile the source into class files. The programmer's code uses the generated classes to access the web service. Here is an example code snippet:

```
AddNumbersPortType port = AddNumbersService().getAddNumbersPort();
a = 10;
b = 20;
result = port.addNumbers(a,b);
```

For both of the associated examples, invoking

```
ant client
```

or

```
ant -Duse.tomcat=true client
```

will run **wsimport** to retrieve the service's WSDL and compile the source.

6. Running the Web Service Client

For both examples, execute the resulting command-line clients via

```
ant run
```

or

```
ant -Duse.tomcat=true run
```

That target simply runs Java with the name of the client's class, such as **java fromwsdl.client.AddNumbersClient**. However, for convenience the **run** target takes care of passing a list of jar files via Java's **-classpath** option. When you invoke the **run** target, you can expect to see output from the client similar to the following:

```
[java] May 4, 2006 2:45:50 PM
[com.sun.xml.ws.policy.jaxws.PolicyWSDLParserExtension] addClientConfigToMap
[java] WARNING: Optional client configuration file URL is missing. No client
con
figuration is processed.
[java] Invoking addNumbers(10, 20)
[java] The result of adding 10 and 20 is 30.

[java] Invoking addNumbers(-10, 20)
[java] Caught AddNumbersFault_Exception: Numbers: -10, 20
```

The WARNING line above is expected for both of these examples. Given that no Metro technologies are enabled, a configuration file is unnecessary. More information will be provided on Metro configuration files in the following article.

7. Undeploying a JAX-WS Web Service

Undeploying a web service means to disable & remove it from the web container. Clients will no longer be able to use the web service nor will the web service restart without explicit redeployment by the user. During the development process, it is often useful to undeploy a web service. This section explains the necessary steps for both GlassFish and Tomcat.

7.1. Undeploying from GlassFish

The **asadmin** command provides the simplest method of undeploying a web service from GlassFish.

```
asadmin undeploy --user admin wsit-jaxws-fromjava
asadmin undeploy --user admin wsit-jaxws-fromwsdl
```

7.2. Undeploying from Tomcat

Undeploying a given web service from Tomcat requires deleting its WAR file from the Tomcat webapps directory. For a typical UNIX scenario the commands below would delete the sample WAR files. Tomcat then automatically undeploys the web service within a few seconds.

```
rm $CATALINA_HOME/webapps/wsit-jaxws-fromjava.war
rm $CATALINA_HOME/webapps/wsit-jaxws-fromwsdl.war
```

Enabling Advanced Features in a Web Service Application

Abstract

This article highlights the steps required to enable Metro-specific advanced functionalities in a web service and its corresponding client application. As with the previous article, two accompanying code samples are included. Again, one starts from Java source code and the other from an existing WSDL document to develop their respective web services. However, this article and its code samples show how WS-Policy can be used to enable WS-Addressing and WS-Reliable Messaging in the web services and their clients.

Table of Contents

1. Overview	7
2. Prerequisites and Environment Configuration	7
3. WSIT Configuration and WS-Policy	8
4. Configuring WSIT in the Web Service	8
5. Building and Deploying the Web Service	9
6. Configuring WSIT in the Web Service Client	9
7. Building and Running a Web Service Client	9
8. Undeploying a Web Service	10

1. Overview

Supporting code samples are included to demonstrate building a web service using WSIT functionality. The examples show how to develop a web service both starting from Java source code and starting from an existing WSDL document. For both cases, it shows how to develop a corresponding client application from the web service's WSDL document. The examples can be found in the WSIT source tree here:

- From-Java example [[download/wsit-enabled-fromjava.zip](#)]
- From-WSDL example [[download/wsit-enabled-fromwsdl.zip](#)]

As you follow along with the sample code, please confirm that you are working in either `wsit-enabled-fromjava` or `wsit-enabled-fromwsdl` rather than one of the previous article's sample code directories, `wsit-jaxws-fromjava` or `wsit-jaxws-fromwsdl`.

2. Prerequisites and Environment Configuration

As in the previous article, the steps in this document require that you have already installed the WSIT jars into your web container. It also requires the following software be installed: JDK 6.0 Update 29 [<http://www.oracle.com/technetwork/java/javase/downloads/index.html>] or later, Apache Ant 1.6.5 [<http://ant.apache.org/>] or later, and a web container: either Glassfish v3.x [<http://glassfish.java.net/>] or Apache Tomcat 7.0 [<http://tomcat.apache.org/>]. Further, your Metro build environment needs to be configured as described in the Environment Configuration Settings section of the previous article.

3. WSIT Configuration and WS-Policy

Advanced web service features are enabled and configured using a mechanism defined by the Web Services Policy Framework [http://specs.xmlsoap.org/ws/2004/09/policy/] (WS-Policy) specification. A web service expresses its requirements and capabilities via policies embedded in the service's WSDL description. A service consumer verifies it can handle the expressed requirements and, optionally, uses server capabilities advertised in policies.

Technologies like Reliable Messaging, Addressing, or Secure Conversations, provides a set of policy assertions it can process. Those assertions provide the necessary configuration details to the Metro run-time to enable proper operation of these features used by a given web service. The assertions may specify particular configuration settings or rely on default settings that are pre-determined by the specific technology. For instance, in the snippet shown below, `wsm:AcknowledgementInterval` and `wsm:InactivityTimeout` are both optional and could be omitted. The following is an XML snippet showing WS-Policy assertions for WS-Addressing and WS-Reliable Messaging:

=

=

=

=

This snippet would be equally valid in either a WSIT configuration file or a web service's WSDL document.

4. Configuring WSIT in the Web Service

4.1. Starting from Java

When developing a web service from scratch or based on an existing Java class, WSIT features are enabled using a configuration file. That file, `wsit-fromjava.server.AddNumberImpl.xml`, is written in WSDL format. An example configuration file can be found in the accompanying samples:

- `wsit-enabled-fromjava/etc/wsit-fromjava.server.AddNumbersImpl.xml` [samples/wsit-enabled-fromjava/etc/wsit-fromjava.server.AddNumbersImpl.xml]

The configuration file settings will be incorporated dynamically by the WSIT run-time into the WSDL it generates for the web service. So when a client requests the service's WSDL, the run-time will embed into the WSDL any publically visible policy assertions contained in the configuration file. For the example link above, the Addressing and Reliable Messaging assertions would be part of the WSDL as seen by the client.

Note

`wsit.xml` must be in the `WEB-INF` sub-directories of the application's WAR file when it is deployed to the web container. Otherwise, the WSIT run-time environment will not find it.

4.2. Starting from WSDL

When developing a web service starting from an existing WSDL, the situation is actually simpler. The policy assertions needed to enable various WSIT technologies will already be embedded in the WSDL document. Here is an example WSDL document in the accompanying samples:

- wsit-enabled-fromwsdl/etc/AddNumbers.wsdl [samples/wsit-enabled-fromwsdl/etc/AddNumbers.wsdl]

5. Building and Deploying the Web Service

Once configured, a WSIT-enabled web service is built and deployed in the same manner as a standard JAX-WS web service. If you are not familiar with those steps, please review the following sections from Building a Simple Metro Application: Building a JAX-WS Web Service and Deploying the Web Service to a Web Container. However, the URLs needed to verify the respective web services differ from the previous article's examples and are listed below:

- <http://localhost:8080/wsit-enabled-fromjava/addnumbers?wsdl>
- <http://localhost:8080/wsit-enabled-fromwsdl/addnumbers?wsdl>

6. Configuring WSIT in the Web Service Client

Client-side configuration of WSIT functionality is largely automatic in the WSIT environment. The WSDL document seen by the client will already contain the WSIT policy assertions. Those assertions describe any requirements from the server as well as any optional features the client may use. The WSIT build tools and run-time environment will detect the WSDL's policy assertions and configure themselves appropriately, if possible. If an unsupported assertion is found, an error message describing the problem will be displayed.

7. Building and Running a Web Service Client

As with the web service itself, building and running a WSIT-enabled client application is identical to running a standard JAX-WS client application. Those steps are described in the following sections of the previous article: Building a JAX-WS Web Service Client and Running the Web Service Client. You can expect to see output from the client similar to the following:

```
[java] Invoking addNumbers(10, 20)
[java] The result of adding 10 and 20 is 30.
[java]
[java] Invoking addNumbers(-10, 20)
[java] Caught AddNumbersFault_Exception: Numbers: -10, 20
[java] 12.1.2012 15:34:37 [com.sun.xml.ws.rx.rm.runtime.ClientTube]
      closeSequences
[java] INFO: WSRM1157: Waiting for sequence
      [ uuid:6ecc55a3-78cf-4e8f-9b18-87ffa6fbb8b0 ] state change to [ CLOSED ] has
      timed out after 3 000 milliseconds

[java] 12.1.2012 15:34:40 [com.sun.xml.ws.rx.rm.runtime.ClientTube]
      closeRmSession
[java] INFO: WSRM1157: Waiting for sequence
      [ uuid:6ecc55a3-78cf-4e8f-9b18-87ffa6fbb8b0 ] state change to
      [ TERMINATING ] has timed out after 3 000 milliseconds
```

8. Undeploying a Web Service

As described in Undeploying a JAX-WS Web Service, to undeploy a web service means to both disable and remove it from the web container. This section provides the necessary commands to undeploy this article's sample web services from both GlassFish and Tomcat.

8.1. Undeploying from GlassFish

```
asadmin undeploy --user admin wsit-enabled-fromjava  
asadmin undeploy --user admin wsit-enabled-fromwsdl
```

8.2. Undeploying from Tomcat

```
rm $CATALINA_HOME/webapps/wsit-enabled-fromjava.war  
rm $CATALINA_HOME/webapps/wsit-enabled-fromwsdl.war
```