JBOSS WEB-SERVICES AT A GLANCE

Introduction to JBossWS in JBoss Enterprise Application Platform

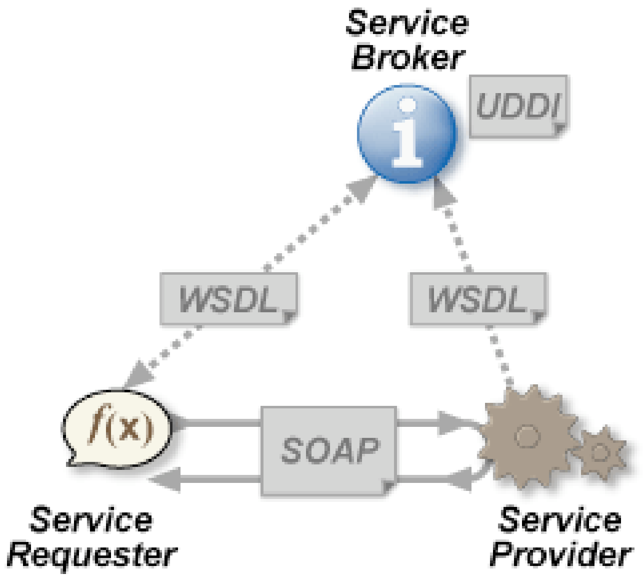


Table of Contents

JBoss Web Services Introduction

Overview

JBossWS is a web service framework developed as part of the JBoss Application Server. It implements the JAX-WS specification that defines a programming model and run-time architecture for implementing web services in Java, targeted at the Java Platform, Enterprise Edition 5 (Java EE 5).

JBossWS integrates with most current JBoss Application Server releases as well as earlier ones, that did implement the J2EE 1.4 specifications. Even though JAX-RPC, the web service specification for J2EE 1.4, is still supported JBossWS does put a clear focus on JAX-WS. A detailed discussion of JAX-WS is outside the scope of this paper, but more information about JAX-WS can be found at: http://jcp.org/en/jsr/detail?id=224. SOA Using Java Web Services, by Mark D. Hansen is also another very good Java web services resource.

This paper will provide a short how to guide on writing web services in JBossWS-Native, as well as the Sun and Apache CXF implementations.

JBossWS Compared to Metro(Sun) and XFire(Now called Apache-CXF)

JBossWS will be a full Java EE 5 web-services implementation, so there is no need to use the Sun or Apache-CXF implementations. But, Red Hat does see value in these alternative web services implementations. JBossWS is built with pluggability in mind. Today, one can swap out the JBossWS-Native implementation, which is our standard, supported version of web services and replace it with Sun's or the Apache CXF implementation. These alternative web services implementations are not supported via subscription today, but if the community adoption rate is high, Red Hat may begin supporting these alternative stacks as early as JBoss Enterprise Application Platform 5.0.

JBossWS Can be used Today for JAX-WS Development

Although JBossWS is not fully Java EE 5 compliant, many Java EE 5 features are present, usable and supported by Red Hat today. JSR-181 annotations, JAXB 2.0, WSEE 1.2 and EJB 3.0 can all be used in JBoss Enterprise Application Platform 4.2 or higher.

Well Documented JBossWS Use Cases

JBossWS does lack a few Java EE 5 features, but there is an impressive list of web service use cases that are well documented.

- *Top Down Development* - JAX-WS web services can be developed by starting with a WSDL and generating Java code using the *wsconsume* tool. The generated code can be deployed to JBoss Enterprise Application Platform with little to no manual XML descriptor file editing.
- *Bottom Up Development* - Existing POJOs(Plain Old Java Objects) and SLSBs(Stateless Session Beans) can be annotated using JSR-181. This annotated code can be deployed to JBoss Enterprise Application Platform with little to no manual XML descriptor file editing.
- *WS-Addressing* - Stateful web-services can be written in JBossWS. SEI clients will attach a client id to each request, per the WS-Addressing specification and the server will associate the client requests and maintain state.
- *WS-Eventing* - A set of operations that allow an event consumer to register (subscribe) with an event producer (source) to receive events (notifications) in an asynchronous fashion.
- *MTOM* - Message Transmission Optimization Mechanism (MTOM) and XML-binary Optimized Packaging (XOP), a means of more efficiently serializing XML Infosets that have certain types of content.
- *Document Literal (Wrapped and Bare), RPC (Literal and Encoded)* - JBossWS provides a great deal of flexibility with respect to SOAP document style. RPC Literal and Document Literal wrapped are the most interoperable document styles.
- *WS-Policy* - Current JBoss implementation can instrument a web-service with policies attached at endpoint, port or port-type scope level only. The Web Services Policy Framework (WS-Policy) provides a general purpose model and corresponding syntax to describe the policies of a Web Service.

There are additional JBossWS use cases like REST, WS-Transaction, UDDI/JAXR. Some are more documented than others, but important to note that the Red Hat team is working hard to deliver very robust, thorough documentation for all noteworthy JBossWS use cases. For now, refer to the JBossWS community user guide wiki, http://jbws.dyndns.org/mediawiki/index.php?title=JAX-WS_User_Guide, for up to date JBossWS user guidance.

JBoss Web Services Tools

Development Environment

Red Hat Development Studio, Eclipse 3.x or any other decent Java IDE will provide a decent JBossWS development environment. There are no JBossWS specific Red Hat Developer Studio plugins yet, but they are planned for a later Red Hat Developer Studio release. The JBossWS plugins will be made available to all Eclipse 3.3+ users. An IDE agnostic approach to developing web services using JBossWS will be used in this paper. Outside of a decent Java editor, the only other development environment dependencies are JBoss Enterprise Application Platform 4.3+ or JBoss Application Server 4.2.2+, Maven 2.0.7+ and a good SOAP request client simulator like soapUI. Unfortunately, Eclipse doesn't provide a very good SOAP client interface, but SOAP UI is a good, free tool. Ant 1.7.0 is optional but also recommended. Maven2 will be used as the web service build tool. Maven2 has an excellent eclipse plugin that'll allow you to do all of your development/builds in the IDE. Ant will be used to deploy built web service archives to JBoss.

Code used in this White Paper

The project for this walk-through can be found at:
http://wiki.jboss.org/wiki/attach?page=JBossWSWhitepaperProject%2Fjbossws-whitepaper.zip. All build scripts, and source code is included in the ZIP archive. However, Maven2, Ant 1.7.0, JBossAS 4.2.2 or JBoss Enterprise Application Platform are needed to build/compile and deploy the project.

Recommended Software

The required software can be downloaded from the Internet:

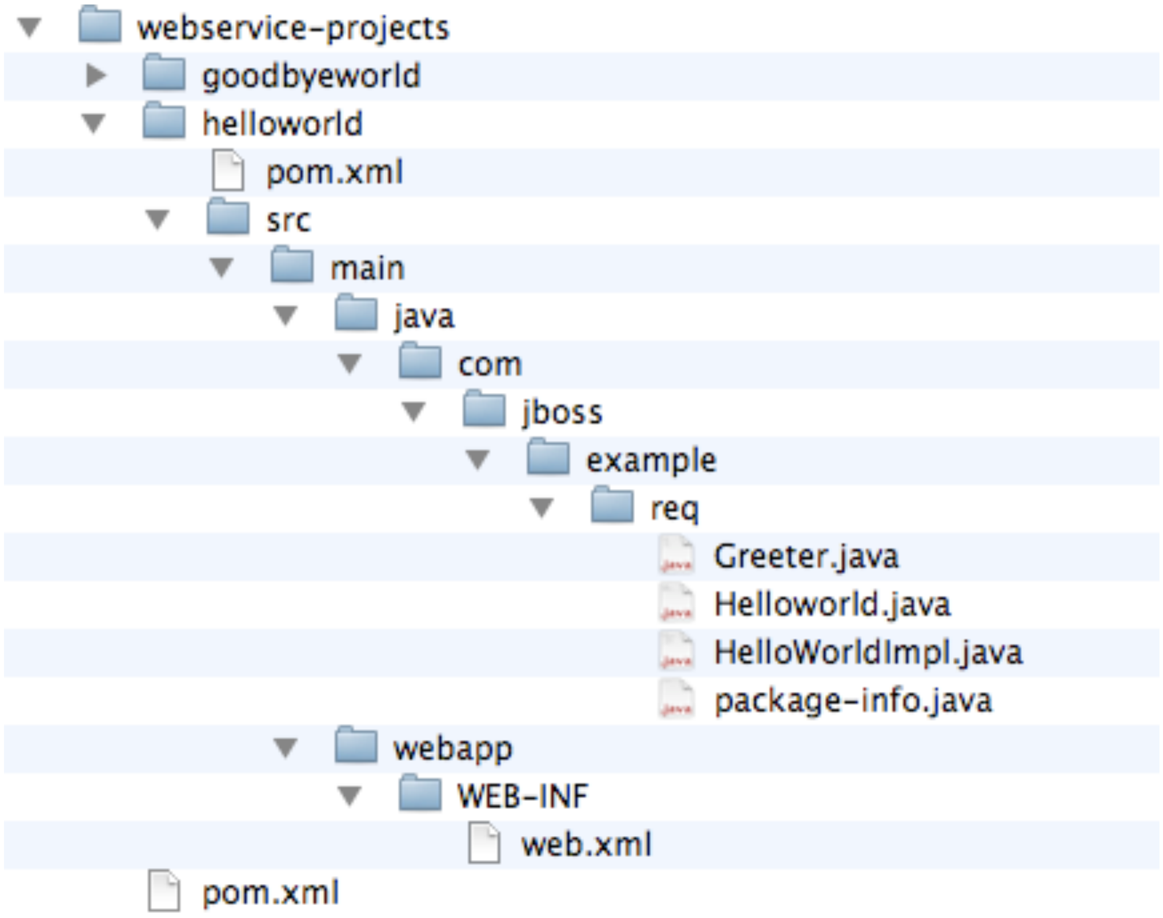
- *IDE (Choose one)*
    - Red Hat Developer Studio - http://www.redhat.com/developers/rhds/
    - Eclipse 3.3 - http://www.eclipse.org/downloads/
- *Enterprise Java Server Runtime (Choose one)*
    - JBoss Enterprise Application Platform 4.3 - https://network.jboss.com/jbossnetwork/restricted/listSoftware.html
    - JBoss Application Server 4.2.2 - http://labs.jboss.com/jbossas/downloads
- *WAR/EAR Build tool*
    - Maven 2.0.7 - http://maven.apache.org/download.html
    - Maven2 eclipse plugin update site - http://m2eclipse.codehaus.org/update/
- *WAR/EAR deployment tool*
    - Apache Ant 1.7.0 - http://ant.apache.org/bindownload.cgi
- *SOAP client simulator tool*
    - soapUI 1.7.6 - http://www.soapui.org/

Proper installation/configuration of the software listed above is outside the scope of this document, but there are plenty of good, web based manuals for the required software.

Project Configuration and Deployment

Project File Hierarchy

This file structure supports multiple WARs and is Maven compliant. Each WAR has it's own pom.xml file, which is described in the next section.



```
webservice-projects
    goodbyeworld
    helloworld
        pom.xml
        src
            main
                java
                    com
                        jboss
                            example
                                req
                                    Greeter.java
                                    Helloworld.java
                                    HelloWorldImpl.java
                                    package-info.java
                webapp
                    WEB-INF
                        web.xml
    pom.xml
```

Maven can create new projects like "helloworld" with a single command line call, but that is outside the scope of this document.

Build/Deploy Configuration - pom.xml and build.xml

While a detailed discussion on Maven2 best practices is outside the scope of this document, it is important to provide insight into how Maven2 can be used in JBossWS development. It is a good idea to create a parent "pom.xml" that defines profile properties:

```
...
<properties>
      <jbossws.home>
            ~/jboss-4.2.2.GA/server/default/deploy/jbossws.sar/
      </jbossws.home>
      <jboss.lib>
            ~/jboss-4.2.2.GA/server/default/lib/
      </jboss.lib>
      <jboss.client.lib>
            ~/jboss-4.2.2.GA/client/
      </jboss.client.lib>
      <jboss.home>
            ~/jboss-4.2.2.GA/
      </jboss.home>
      <webservice.wars.root>
            ~/workspace/webservice-projects
      </webservice.wars.root>
</properties>
...
```

The properties will be referenced later in the parent pom.xml file and in the child project pom.xml files. The "webservice.wars.root" property is a noteworthy reference to the parent directory for multiple web service WAR projects.

It is also a good idea to define the dependencies in this parent pom.xml file. If JBossWS had a maven repository for its JARs, it would be unnecessary to define them manually, but unfortunately, there isn't a JBossWS maven repository yet. So, one must define them in the pom.xml:

```
...
<dependency>
      <groupId>jbossws</groupId>
      <artifactId>jboss-jaxb</artifactId>
      <version>system</version>
      <scope>system</scope>
      <systemPath>${jbossws.home}/jaxb-api.jar</systemPath>
</dependency>
<dependency>
      <groupId>jbossws</groupId>
      <artifactId>jbossall-client</artifactId>
      <version>system</version>
      <scope>system</scope>
      <systemPath>${jbossws.client.lib}/jbossall-client.jar</systemPath>
</dependency>
<!--additional dependencies -->
...
```

Readers new to Maven2 might be wondering why there is a parent pom.xml file. The justification for this file has everything to do with storing global data in a single file, instead of every JBossWS copy/pasting the same values in every pom.xml file. By using a parent pom.xml, build and deployment settings can be shared across multiple web-service WAR projects. Each WAR project will have its own pom.xml file that looks like:

```
...
<groupId>samples</groupId>
      <artifactId>helloworld</artifactId>
      <packaging>war</packaging>
      <version>1.0</version>
      <name>Helloworld Web Service</name>
      <parent>
            <groupId>samples</groupId>
            <artifactId>jbossws-demo</artifactId>
            <version>1.0</version>
      </parent>
...
```

The "artifactId" should be the name of your web service WAR. By defining, "war" packaging, the Maven will automatically create a WAR archive that JBoss is able to deploy. The parent pom.xml *groupId* is defined, so that Maven pulls dependencies and properties from the parent. This is far better than copy/pasting the same dependencies and server properties for every WAR.

That's pretty much all there is to the build process. Maven2 is quite handy and this solution will work with any IDE or no IDE at all. One problem with Maven2 is that it lacks the ability to deploy WARs from a Maven WAR repository to the JBoss deployment directory. This can be remedied by extending Maven's "deploy" target with an ant callout:

```
<plugin>
   <artifactId>maven-antrun-plugin</artifactId>
   <executions>
      <execution>
         <phase>deploy</phase>
            <configuration>
                  <tasks>
                     <ant antfile="${webservice.wars.root}/build.xml"
                          target="deploy" inheritAll="true">
                        <property name="project.name" value="helloworld" />
                        <property name="jboss.deploy.dir"
                               value="${jboss.home}/server/default/deploy" />
               </ant>
            </tasks>
            </configuration>
            <goals>
               <goal>run</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

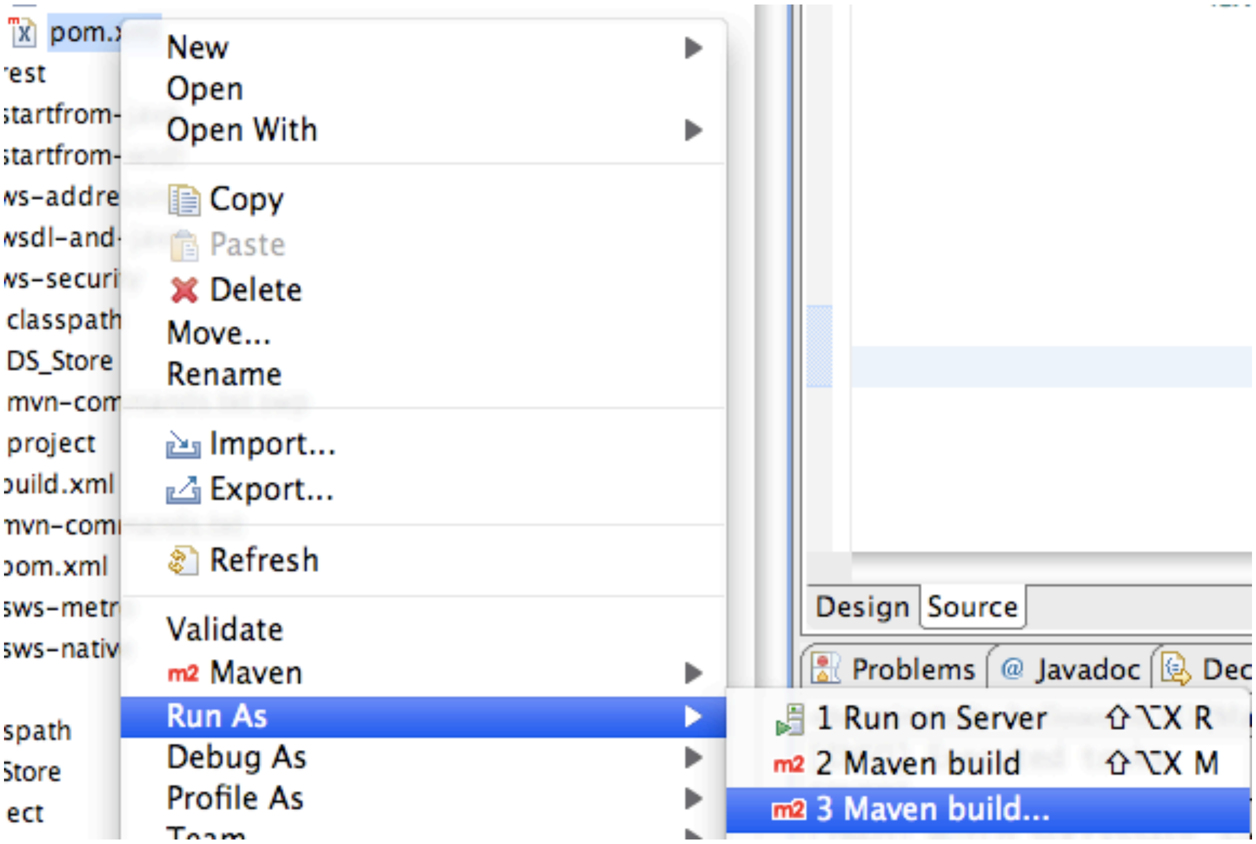
The text in bold is the ant callout. Properties are passed from Maven to Ant and a simple ant deployment target is executed. The Ant *build.xml deploy* target looks like:

```
<target name="deploy">
   <echo message="Deploying ${project.name} to JBoss"/>
      <copy todir="${jboss.deploy.dir}"
            file="${basedir}/target/${project.name}.war" />
</target>
```

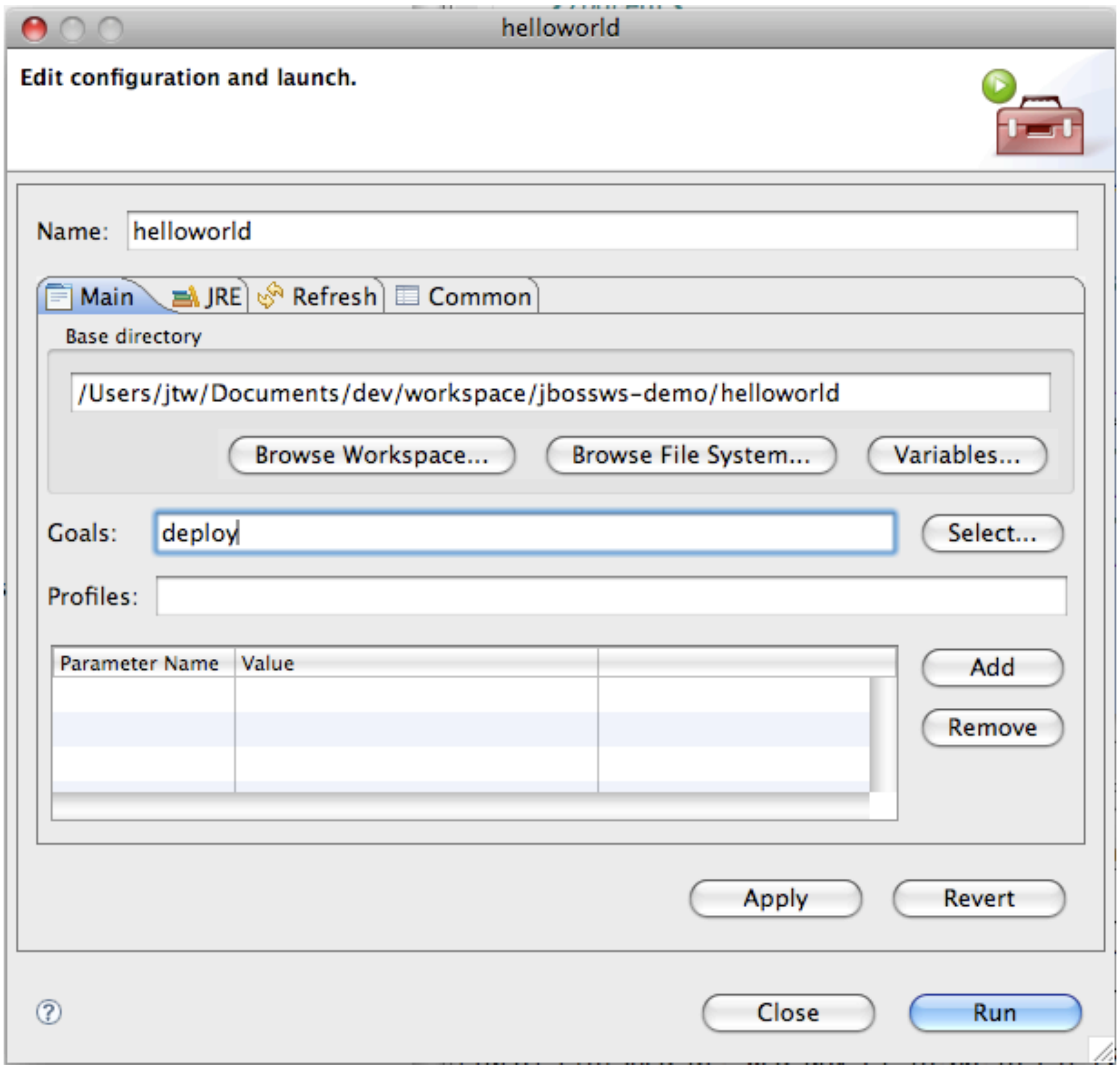
It is important to note that this simple Ant callout from Maven2 involves far less effort than writing the entire build script in Ant. Better yet, web service project build and deployment can be executed with a single Maven target, “deploy”.

Build/Deploy in Eclipse using Maven2 plugin

There is a nice Maven2 plugin for Eclipse that provides a convenient way of building and deploying web service projects. Detailed instructions for installing/configuring the plugin are outside the scope of this document, but here are a few screenshots that show how easy it is to execute Maven2 builds in Eclipse:



Right click on any pom.xml and select “Maven build...” to setup the desired execution target.


helloworld

Edit configuration and launch.

Name: helloworld

Base directory

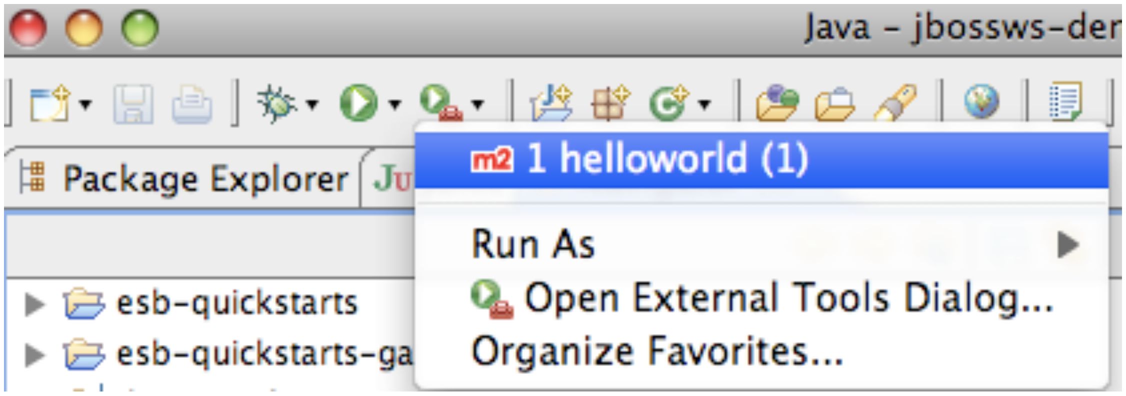
/Users/jtw/Documents/dev/workspace/jbossws-demo/helloworld

Goals: deploy

Profiles:

Parameter Name	Value

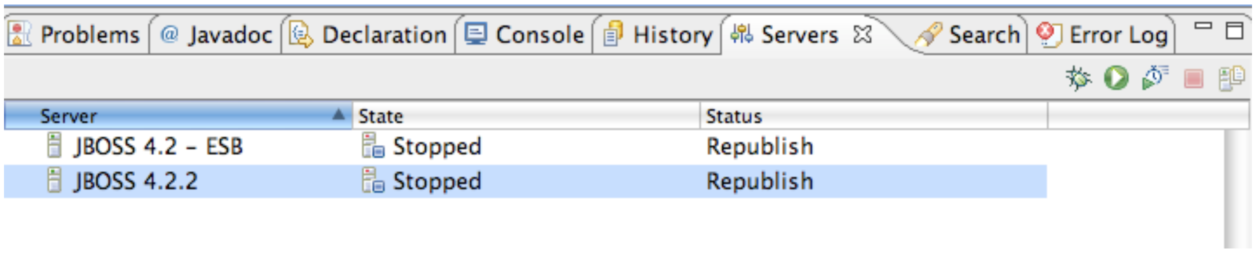
Specify the desired goal, for example "deploy", and click "Run". The project should now build and deploy properly. This is a one-time setup.



From now on, you can trigger a build/deploy by clicking on the Eclipse external tools runner. It would be nice if the IDE generated maven scripts with defined dependencies and integrated WTP based deployment, but this approach is a productive alternative that is IDE agnostic.

Server Configuration in Eclipse

Starting and stopping JBoss AS in Eclipse isn't required, but if a JBoss 4.2.2 or JBoss Enterprise Application Platform server is configured in the "Servers" view, it will be easy to start/stop the server in Eclipse.



Server	State	Status
JBOSS 4.2 – ESB	Stopped	Republish
JBOSS 4.2.2	Stopped	Republish

The Eclipse WTP(Web Tools Project) publishing capabilities are not available for JBossWS projects yet, but a future version of Red Hat Developer Studio will support web service project publishing through WTP.

Web Service Code Walk Through

com.jboss.example.req.Helloworld.java

It is a good idea to create a web service interface class.

```
@WebService(name = "HelloWorldPort")
public interface Helloworld {

      @WebMethod
      public Greeter sayHello(@WebParam(name="greeting")
                              ArrayList<Greeter> greeters);

}
```

@Webservice is required, but both @WebMethod and @WebParam are optional. The @Web-Param make JBossWS generated WSDL more human readable. The @WebMethod is used to self-document the fact that this method will be exposed as a web service. Without a:

```
  @SOAPBinding(style = SOAPBinding.Style.RPC)
```

override annotation, JBossWS will create and deploy a Document/Literal Wrapped WSDL.

com.jboss.example.req.Greeter.java

The web service request will pass an ArrayList of Greeter domain objects. This is a simple DTO with a couple setters/getters. The server will also return a Greeter response object.

```
public class Greeter {

      private String name;
      private String saying;

      public String getName() {
            return name;
      }

      public void setName(String name) {
            this.name = name;
      }

      public String getSaying() {
            return saying;
      }

      public void setSaying(String saying) {
            this.saying = saying;
      }
```

This class exists to demonstrate the JAXB 2.0 bindings in action. Notice that there are no JSR-181 annotations in this class. JBossWS will auto-bind this POJO because it is defined in the Java interface web method signature.

com.jboss.example.req.HelloworldImpl.java

Once the interface is defined, a Java implementation class must be written.

```
@WebService(endpointInterface="com.jboss.example.req.Helloworld")
public class HelloWorldImpl implements Helloworld {

      public Greeter sayHello(ArrayList<Greeter> greeters)
      {
            Greeter server = new Greeter();
            String serverSaying = new String("Hello Back to: ");

            server.setName("John Doe");

            for (Greeter greeter:greeters)
            {
                  System.out.println(greeter.getName());
                  serverSaying += greeter.getName() + ", ";
            }

            serverSaying += ".";
            server.setSaying(serverSaying.replace(", .", "."));

            return server;
      }
}
```

@WebService is required with an endpointInterface reference to the interface. This class is a POJO, but a SLSB could also be exposed as a web service. The implementation is simplistic but it demonstrates the power of JAXB by looping through a JAXB populated ArrayList, appending data then returning a JAXB un-marshalled result to the client.

web.xml

Since a POJO is used instead of a SLSB, a web.xml servlet endpoint must be defined.

```
<servlet>
    <servlet-name>com.jboss.example.HelloWorldImpl</servlet-name>
    <servlet-class>com.jboss.example.HelloWorldImpl</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>com.jboss.example.HelloWorldImpl</servlet-name>
    <url-pattern>/HelloworldPort</url-pattern>
</servlet-mapping>
```

It may seem redundant, but both a servlet and servlet-mapping element must be defined. Also, the “servlet-name”, “servlet-class” elements should all point to the web service implementation class. The "url-pattern" defines the context root relative location of the web service. Since the WAR name is "helloworld.war" and the url-pattern is "HelloWorldPort", the web service WSDL location will be: http://127.0.0.1:8080/helloworld/HelloworldPort?wsdl.

package-info.java

This file is optional but it is a good idea to include it and define the web service namespace.

```
@javax.xml.bind.annotation.XmlSchema(namespace = "http://www.jboss.com/hws",
elementFormDefault = javax.xml.bind.annotation.XmlNsForm.QUALIFIED)
package com.jboss.example.req;
```

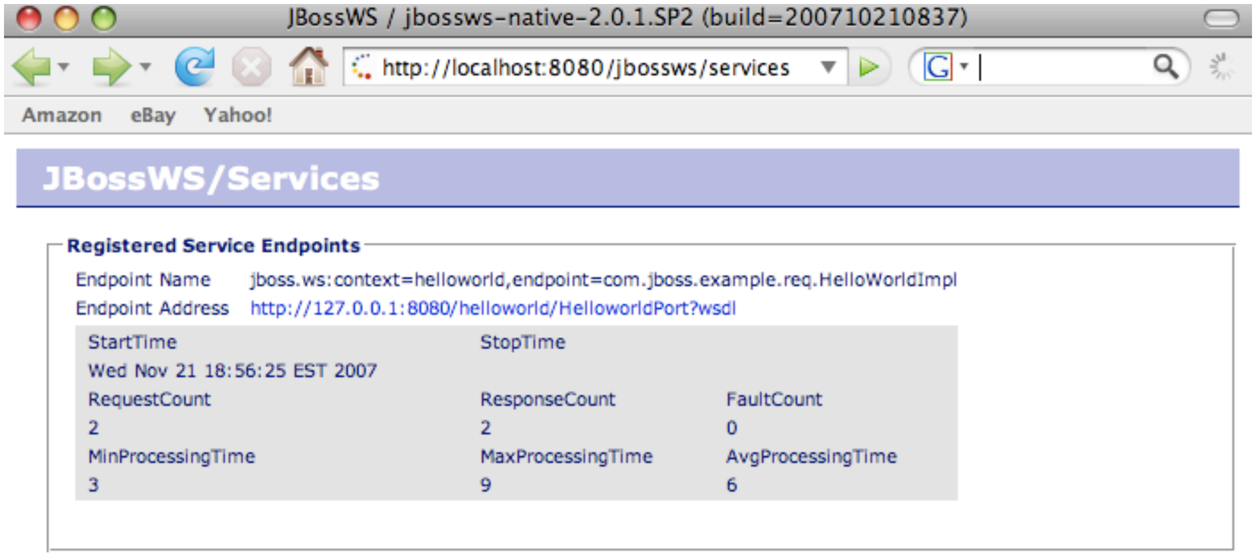
The package-info.java settings will force a SOAP request payload to look like:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
                  xmlns:req="http://req.example.jboss.com/"
                  xmlns:hws="http://www.jboss.com/hws" >

<soapenv:Header/>

   <soapenv:Body>

      <req:sayHello>

         <greeting>

            <hws:name>Jill</hws:name>

            <hws:saying>Hello!</hws:saying>

         </greeting>

       <greeting>

            <hws:name>Lisa</hws:name>

            <hws:saying>Whazz Up!</hws:saying>

         </greeting>

      </req:sayHello>

   </soapenv:Body>

</soapenv:Envelope>
```

Of course a web service client will know that the correct request structure because the JBossWS generated WSDL will include the namespace definitions.

Test the Helloworld Web Service in soapUI

Once deployed, the web service should be accessible using a web service client like soapUI. JBossWS registers and exposes all JAXR registered web services to this URL: http://localhost:8080/jbossws/services.

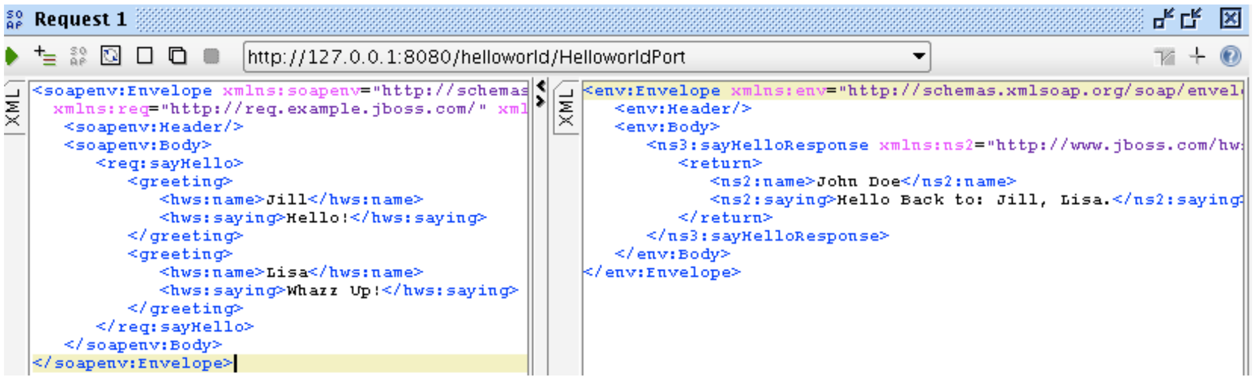

JBossWS / jbossws-native-2.0.1.SP2 (build=200710210837)
http://localhost:8080/jbossws/services
Amazon
eBay
Yahoo!

JBossWS/Services

Registered Service Endpoints

Endpoint Name	jboss.ws:context=helloworld,endpoint=com.jboss.example.req.HelloWorldImpl	
Endpoint Address	http://127.0.0.1:8080/helloworld/HelloworldPort?wsdl	
StartTime	StopTime	
Wed Nov 21 18:56:25 EST 2007		
RequestCount	ResponseCount	FaultCount
2	2	0
MinProcessingTime	MaxProcessingTime	AvgProcessingTime
3	9	6

The soapUI tool can use a JBossWS "Endpoint Address" for a "New WSDL Project". Once the WSDL project is created, requests to the new web service can be simulated. A sample execution soapUI screen looks like:



```xml
<soapenv:Envelope xmlns:soapenv="http://schemas
  xmlns:req="http://req.example.jboss.com/" xml
   <soapenv:Header/>
   <soapenv:Body>
      <req:sayHello>
         <greeting>
            <hws:name>Jill</hws:name>
            <hws:saying>Hello!</hws:saying>
         </greeting>
         <greeting>
            <hws:name>Lisa</hws:name>
            <hws:saying>Whazz Up!</hws:saying>
         </greeting>
      </req:sayHello>
   </soapenv:Body>
</soapenv:Envelope>
```

```xml
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envel
   <env:Header/>
   <env:Body>
      <ns3:sayHelloResponse xmlns:ns2="http://www.jboss.com/hw
         <return>
            <ns2:name>John Doe</ns2:name>
            <ns2:saying>Hello Back to: Jill, Lisa.</ns2:saying
         </return>
      </ns3:sayHelloResponse>
   </env:Body>
</env:Envelope>
```

This soapUI request passes a list of Greeter elements, then executes the "sayHello" web-service. JBossWS will marshall the SOAP payload "req" element into a Java ArrayList of Greeting objects using JAXB. The web service will return a Greeting POJO that JBossWS un-marshalls back to XML using JAXB.

Helloworld using JBossWS-CXF

Build JBossWS-CXF

JBossWS-Native, the default web service implementation can be swapped out with a SAR wrapped Apache-CXF implementation. While swapping out JBossWS-Native is not recommended for production use today, it is technically possible to use this web services implementation as a JBossWS-Native replacement. The most likely use-case for this alternative web services implementation is to port an XFire implementation to JBoss without modifying code. The helloworld project discussed in this paper should deploy properly to JBossWS-CXF because the project doesn't leverage any JBossWS-Native proprietary features. It is important to note that JBossWS-CXF must be built from subversion. There are no pre-built binaries for it yet. The build process is as follows:

- Checkout JBossWS-CXF from subversion
- Modify the build settings in *ant.properties* and *version.properties*
- Run the *"ant deploy-jboss42"* target
- When JBossAS starts up there should be "CXFServerConfig" output on the console and no errors.

Please refer to:
http://jbws.dyndns.org/mediawiki/index.php?title=Building_From_Source#Checkout_and_build_JBossWS-CXF for detailed JBossWS-CXF build instructions.

Helloworld using JBossWS-Metro

Build JBossWS-Metro

JBossWS-Native, the default web service implementation can be swapped out with a SAR wrapped Sun-Metro implementation. While swapping out JBossWS-Native is not recommended for production use today, it is technically possible to use this web services implementation as a JBossWS-Native replacement. The most likely use-case for this alternative web services implementation is to port a Glassfish implementation to JBoss without modifying code. The helloworld project discussed in this paper should deploy properly to JBossWS-Metro because the project doesn't leverage any JBossWS-Native proprietary features. It is important to note that

JBossWS-Metro must be built from subversion. There are no pre-built binaries for it yet. The build process is as follows:

- Checkout JBossWS-Metro from subversion
- Modify the build settings in *ant.properties* and *version.properties*
- Run the *"ant deploy-jboss42"* target
- When JBossAS starts up there should be "MetroServerConfig" output on the console and no errors.

Please refer to:
http://jbws.dyndns.org/mediawiki/index.php?title=Building_From_Source#Checkout_and_build_JBossWS-Metro for detailed JBossWS-Metro build instructions.

RESOURCES

http://en.wikipedia.org/wiki/Web_service

http://jbws.dyndns.org/mediawiki/index.php?title=JAX-WS_User_Guide