

JBoss SOA Platform 4.3
with
JBoss Developer Studio 2.0
Workshop

Table of Contents

Introduction.....	3
Lab #1: Install and Configure SOA Platform.....	4
Lab #2: SOA Platform Quickstarts.....	5
Lab #3: Installation of JBDS.....	11
Lab #4: Configure SOA-P in JBDS.....	17
Lab #5: Creating First ESB Project.....	23
Lab #6: Adding a Custom ESB Action.....	30
Lab #7: Installing SoapUI For WS Testing.....	35
Lab #8: Create a JSR 181 Web Service.....	36
Lab #9: Proxy Web Service with ESB.....	43
Lab #10: Adding XSLT to WS Proxy on ESB.....	53
Lab #11: Adding CBR to WS Proxy on ESB.....	58
Lab #12: Using jBPM.....	62
Conclusion.....	77

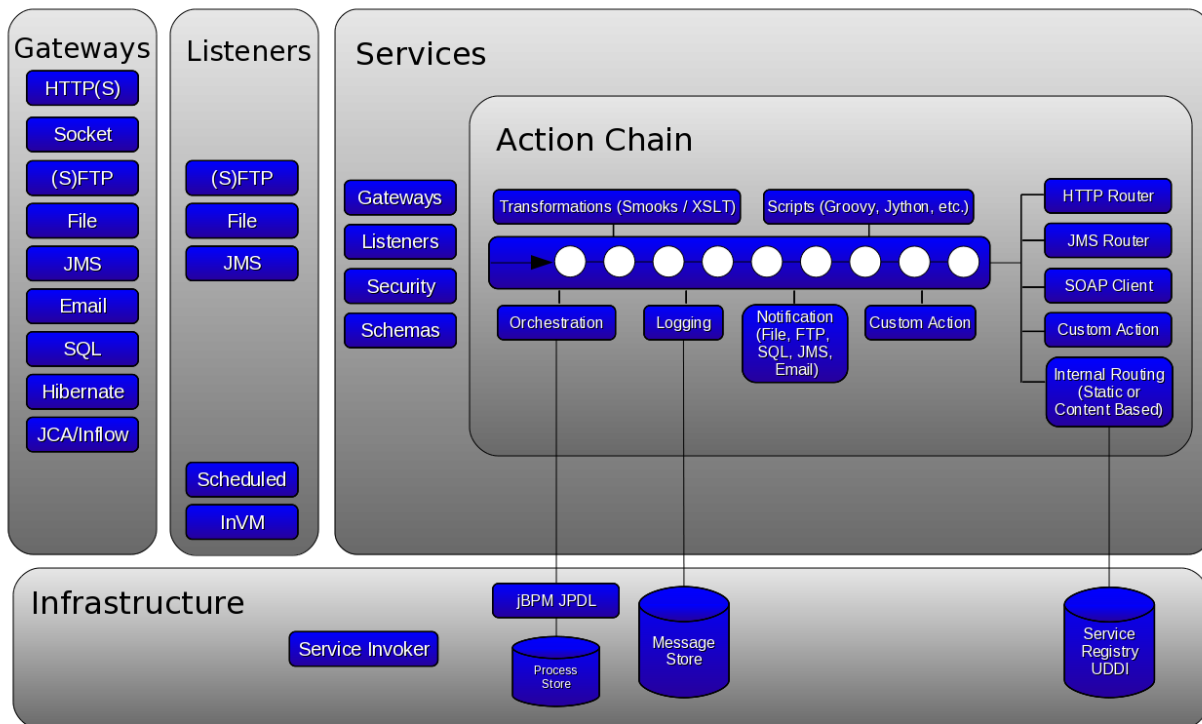
Introduction

Overview

JBoss SOA Platform is a collection of technologies designed to meet an organization's SOA needs. SOA-P includes an ESB, BPM engine (jBPM), Rules engine (JBoss Rules), UDDI Registry (jUDDI), as well as a full JEE application server. To cover each of these areas in depth is beyond the scope of this workshop. Instead, this workshop is designed to give you an overview of the SOA Platform as well as some experience using JBoss Developer Studio to create and deploy SOA-P applications.

SOA-P ESB Architecture

Understanding the SOA-P ESB architecture is important to really understanding what is happening in the following labs. Here is an architecture overview of the SOA-P ESB that we will discuss.



The typical scenario for ESB messages is that a message comes in from the left through a “Gateway”. A Gateway can be any of the protocols listed in the gateways box above or a custom gateway implementation. Any type of message can come into a gateway: text, binary, image, XML, SOAP, etc. The gateway's job is to get the message, wrap it in a JBoss ESB internal message structure and pass it to a “Listener”. The listeners in this context are expecting a JBoss ESB Message structure. They will take the message they are given and hand it to a service. Depending on the service, this may include security authorization or schema validation. Assuming both of those are ok, the message will travel to the action chain for the service. The action chain is simply a chain of Java actions that can perform processing on the message. The type of processing that can be done on the message is limited only by Java. Some Java actions may do transformations, some may execute scripts, some may log, some may call out to external systems like

web services, etc. In the end, the last action in the chain is invoked. This is definitely an over simplification of the process, but will hopefully give some context to the terminology that will be used throughout the workshop.

Included Files

Several files are included with this workshop. There is a copy of SOA-P 4.3 CP02 (platform agnostic). There are copies of JBoss Developer Studio for Windows, Mac, and Linux. There are also copies of SoapUI for Windows, Mac, and Linux.

System Expectations

It is expected that you have a Windows, Linux, or Mac notebook and you are comfortable working and running Java programs on it. It is expected you will have the environment PATH set to include a JDK 5.0 to use for these labs. It is also a good idea to have JAVA_HOME set to your JDK that you plan on using. Please make sure you do this before running any of the labs. You should be able to run "java -version" from a command line and have it list the version of Java that you have installed and configured. Ant is also required to complete lab #2, but the later labs do not need Ant.

What is Expected of You

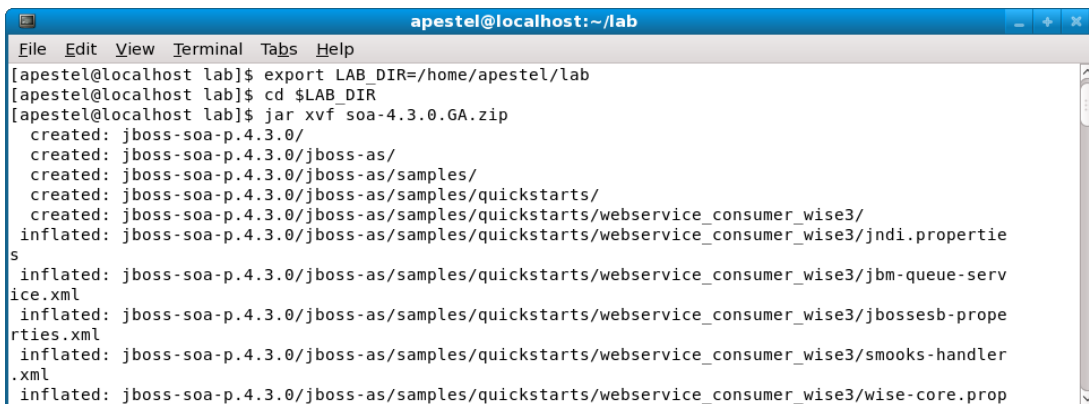
Please feel free to raise your hands with any questions that you have about the lab, why it is you are doing something, or if something does not seem right. Please know that all care was made in creating this user guide, but all screen shots and steps along the way might still be off by just a little so please be patient with any issues.

Lab #1: Install and Configure SOA Platform

Installing SOA-P is pretty straight forward. We just need to change directories into the directory we want SOA-P installed, and unzip it

```
export LAB_DIR=/home/apestel/lab
cd $LAB_DIR
jar xvf soa-4.3.0.GA.zip
```

Please run these commands in a command window as shown below:



```
apestel@localhost:~/lab
File Edit View Terminal Tabs Help
[apestel@localhost lab]$ export LAB_DIR=/home/apestel/lab
[apestel@localhost lab]$ cd $LAB_DIR
[apestel@localhost lab]$ jar xvf soa-4.3.0.GA.zip
created: jboss-soa-p-4.3.0/
created: jboss-soa-p-4.3.0/jboss-as/
created: jboss-soa-p-4.3.0/jboss-as/samples/
created: jboss-soa-p-4.3.0/jboss-as/samples/quickstarts/
created: jboss-soa-p-4.3.0/jboss-as/samples/quickstarts/webservice_consumer_wise3/
inflated: jboss-soa-p-4.3.0/jboss-as/samples/quickstarts/webservice_consumer_wise3/jndi.properties
inflated: jboss-soa-p-4.3.0/jboss-as/samples/quickstarts/webservice_consumer_wise3/jbm-queue-service.xml
inflated: jboss-soa-p-4.3.0/jboss-as/samples/quickstarts/webservice_consumer_wise3/jbossesb-properties.xml
inflated: jboss-soa-p-4.3.0/jboss-as/samples/quickstarts/webservice_consumer_wise3/smooks-handler.xml
inflated: jboss-soa-p-4.3.0/jboss-as/samples/quickstarts/webservice_consumer_wise3/wise-core.properties
```



```

apestel@localhost:~/lab
File Edit View Terminal Tabs Help
inflated: jboss-soa-p-4.3.0/jbpm-jpdl/lib/bsh.jar
inflated: jboss-soa-p-4.3.0/jbpm-jpdl/lib/cglib.jar
inflated: jboss-soa-p-4.3.0/jbpm-jpdl/lib/commons-collections.jar
inflated: jboss-soa-p-4.3.0/jbpm-jpdl/lib/commons-logging.jar
inflated: jboss-soa-p-4.3.0/jbpm-jpdl/lib/dom4j.jar
inflated: jboss-soa-p-4.3.0/jbpm-jpdl/lib/hibernate.jar
inflated: jboss-soa-p-4.3.0/jbpm-jpdl/lib/hsqldb.jar
inflated: jboss-soa-p-4.3.0/jbpm-jpdl/lib/jboss-j2ee.jar.license.txt
inflated: jboss-soa-p-4.3.0/jbpm-jpdl/lib/jbpm-identity.jar
inflated: jboss-soa-p-4.3.0/jbpm-jpdl/lib/jbpm-jpdl.jar
inflated: jboss-soa-p-4.3.0/jbpm-jpdl/lib/jsf-console.war
inflated: jboss-soa-p-4.3.0/jbpm-jpdl/lib/junit.jar
inflated: jboss-soa-p-4.3.0/jbpm-jpdl/lib/log4j.jar
inflated: jboss-soa-p-4.3.0/jbpm-jpdl/release.notes.html
inflated: jboss-soa-p-4.3.0/jbpm-jpdl/src/jbpm-enterprise-sources.jar
inflated: jboss-soa-p-4.3.0/jbpm-jpdl/src/jbpm-identity-sources.jar
inflated: jboss-soa-p-4.3.0/jbpm-jpdl/src/jbpm-jpdl-sources.jar
inflated: jboss-soa-p-4.3.0/jbpm-jpdl/src/resources/gpd/version.info.xml
[apestel@localhost lab]$

```

By default, SOA-P is shipped without an administration user configured. We want to enable that default user for this lab by editing `$LAB_DIR/jboss-soa-p.4.3.0/jboss-as/server/default/conf/props/soa-users.properties` and removing the “#” to uncomment the admin user.



```

apestel@localhost:~/lab/jboss-soa-p.4.3.0/jboss-as/server/d
File Edit View Terminal Tabs Help
admin=admin

```

NOTE: If using JDK 1.6: You must do one more thing. Copy the following files as shown below:

```

cp $LAB_DIR/jboss-soa-p.4.3.0/jboss-as/client/jboss-jax* $LAB_DIR/jboss-soa-p.4.3.0/jboss-as/lib/endorsed
cp $LAB_DIR/jboss-soa-p.4.3.0/jboss-as/client/jboss-saaj.jar $LAB_DIR/jboss-soa-p.4.3.0/jboss-as/lib/endorsed
cp $LAB_DIR/jboss-soa-p.4.3.0/jboss-as/client/jaxb-api.jar $LAB_DIR/jboss-soa-p.4.3.0/jboss-as/lib/endorsed

```

Lab #2: SOA Platform Quickstarts

Now you've got SOA-P installed, but where do you go from here? There are so many things people want to do with a SOA Platform – mediation, web services, jms, transformations, orchestration, security, routing, ftp, and a host of other things. Wouldn't it be nice if there were a single place that had a working example of pretty much every piece of functionality that could be done? SOA-P has that in the form of “quickstarts”. A quickstart is a complete working project (with complete source) that can be build with a single ant task, deployed with a single ant task, and run with a single ant task. You can

run the examples to see them work and then look at the code (or even modify it) to see what is really happening under the covers.

So, let's go look at the quickstarts with SOA-P. They are in `$LAB_DIR/jboss-soa-p.4.3.0/jboss-as/samples/quickstarts`.

```

apestel@localhost:~/lab/jboss-soa-p.4.3.0/jboss-as/samples/quickstarts
File Edit View Terminal Tabs Help
[apestel@localhost quickstarts]$ pwd
/home/apestel/lab/jboss-soa-p.4.3.0/jboss-as/samples/quickstarts
[apestel@localhost quickstarts]$ ls
aggregator                               helloworld_ejb3_ws                 native_client                       transform_XML2XML_date_manipulation
bpm_orchestration1                       helloworld_ejb3_ws_wss            notifications                       transform_XML2XML_simple
bpm_orchestration2                       helloworld_file_action            opensso                              transform_XML2XML_stream
bpm_orchestration3                       helloworld_file_notifier          publish_as_webservice               two_servers
bpm_orchestration4                       helloworld_ftp_action             readme.txt                          webservice_bpel
business_ruleservice_ruleAgent           helloworld_hibernate_action       recipient_list                      webservice_consumer1
business_ruleservice_stateful            helloworld_sql_action             scheduled_services                  webservice_consumer2
business_rules_service                   helloworld_topic_notifier         scripting_chain                      webservice_consumer_wise
business_service                         helloworld_tx_sql_action          scripting_groovy                    webservice_consumer_wise2
conf                                     https_2way_ssl                    security_basic                       webservice_consumer_wise3
custom_action                            huge-split-enrich-transform-route  simple_cbr                          webservice_consumer_wise4
deadletter                               invm_transport1                  smooks_file_splitter_router        webservice_mtom
dynamic_router                           JAAS_action                      spring_aop                          webservice_producer
ejbprocessor                             jms_router                       spring_helloworld                   webservice_proxy
exceptions_faults                       jms_secured                      spring_jpetstore                    webservice_wsaddressing
fun_cbr                                  jms_topic                        static_router                       webservice_wssecurity
groovy_gateway                           jms_transacted                   swift2xml_action                    wiretap
helloworld                               load_generator                   transform_CSV2XML                   wsmq_router
helloworld_action                        messagefilter                    transform_EDIXML_Groovy_XSLT       xml2swift_action
helloworld_bridge                       messagestore                     transform_XML2P030                 xsd_validation
helloworld_ejb3                         monitoring_action                 transform_XML2P0302
[apestel@localhost quickstarts]$

```

Before we run one of these quickstarts to see how they work, we need to start our SOA-P server. For this lab, we're going to start it from the command line. Later on, we'll show starting it from JBDS.

To start the server, you'll need to do this:

```

cd $LAB_DIR/jboss-soa-p.4.3.0/jboss-as/bin
chmod 777 run.sh
./run.sh -c default

```

```

apestel@localhost:~/lab/jboss-soa-p.4.3.0/jboss-as/bin
File Edit View Terminal Tabs Help
[apestel@localhost bin]$ cd /home/apestel/lab/jboss-soa-p.4.3.0/jboss-as/bin
[apestel@localhost bin]$ chmod 777 run.sh
[apestel@localhost bin]$
[apestel@localhost bin]$ ./run.sh -c default
=====
JBoss Bootstrap Environment

JBOSS_HOME: /home/apestel/lab/jboss-soa-p.4.3.0/jboss-as

JAVA: /usr/java/default/bin/java

JAVA_OPTS: -Dprogram.name=run.sh -server -Xms128m -Xmx728m -XX:PermSize=256m -XX:MaxPermSize=512m -Djava.awt.headless=true -Dsun.rmi.dgc.client.gcInterval=3600000 -Dsun.rmi.dgc.server.gcInterval=3600000 -Dsun.lang.ClassLoader.allowArraySyntax=true -Djava.net.preferIPv4Stack=true

CLASSPATH: /home/apestel/lab/jboss-soa-p.4.3.0/jboss-as/bin/run.jar:/usr/java/default/lib/tools.jar

=====
19:38:10,569 INFO [Server] Starting JBoss (MX MicroKernel)...
19:38:10,580 INFO [Server] Release ID: JBoss [SOA] 4.3.0.GA_CP02_SOA (build: SVNTag=4.3.0.GA_CP02_SOA date=200908051807)
19:38:10,581 INFO [Server] Home Dir: /home/apestel/lab/jboss-soa-p.4.3.0/jboss-as
19:38:10,581 INFO [Server] Home URL: file:/home/apestel/lab/jboss-soa-p.4.3.0/jboss-as/
19:38:10,582 INFO [Server] Patch URL: null
19:38:10,582 INFO [Server] Server Name: default
19:38:10,582 INFO [Server] Server Home Dir: /home/apestel/lab/jboss-soa-p.4.3.0/jboss-as/server/default

```

The “-c default” tells the start script to run the “default” server configuration. There is also an “all”,

“production”, and “minimal” configurations in \$LAB_DIR/jboss-soa-p.4.3.0/jboss-as/server. These different configurations have slightly different features enabled, logging levels set, etc.

You'll know the server is up when you see the line that is highlighted below. Leave the server running.

```

apestel@localhost:~/lab/jboss-soa-p.4.3.0/jboss-as/bin
File Edit View Terminal Tabs Help
19:39:02,178 INFO [TomcatDeployer] deploy, ctxPath=/jbossesb, warUrl=../tmp/deploy/tmp4814management.esb-contents/jbossesb-exp.war/
19:39:02,315 INFO [ContextConfig] WARNING: Security role name user used in an <auth-constraint> without being defined in a <security-rol
e>
19:39:02,456 INFO [JBoss4ESBDeployer] create esb service, slsb.esb
19:39:04,128 INFO [JBoss4ESBDeployer] create esb service, smooks.esb
19:39:04,157 INFO [TopicService] Topic[/topic/org.jboss.soa.esb.transformation.Update] started, fullSize=200000, pageSize=2000, downCach
eSize=2000
19:39:04,191 INFO [SmooksService] Centralized Smooks Instance (Console Based) instance not started. See the 'console.url' property in '
/smooks.esb.properties'
19:39:06,207 INFO [JBoss4ESBDeployer] create esb service, soap.esb
19:39:07,746 INFO [JBoss4ESBDeployer] create esb service, spring.esb
19:39:07,882 INFO [Http11Protocol] Starting Coyote HTTP/1.1 on http-127.0.0.1-8080
19:39:07,955 INFO [AjpProtocol] Starting Coyote AJP/1.3 on ajp-127.0.0.1-8009
19:39:07,968 INFO [Server] JBoss (MX MicroKernel) [4.3.0.GA_CP02_SOA (build: SVNtag=4.3.0.GA_CP02_SOA date=200908051807)] Started in 57s
:383ms
19:39:10,504 INFO [Configuration] configuring from resource: monitoring.cfg.xml
19:39:10,505 INFO [Configuration] Configuration resource: monitoring.cfg.xml
19:39:10,521 INFO [Configuration] Reading mappings from resource : org/jboss/soa/esb/monitoring/monitoring-mappings.hbm.xml
19:39:10,572 INFO [HbmBinder] Mapping class: org.jboss.soa.esb.monitoring.pojo.JMXPattern -> JMXPATTERN
19:39:10,572 INFO [HbmBinder] Mapping class: org.jboss.soa.esb.monitoring.pojo.JMXData -> JMXDATA
19:39:10,583 INFO [HbmBinder] Mapping class: org.jboss.soa.esb.monitoring.pojo.JMXOperationResult -> JMXOPERATIONRESULT
19:39:10,586 INFO [HbmBinder] Mapping class: org.jboss.soa.esb.monitoring.pojo.JMXAttribute -> JMXATTRIBUTE
19:39:10,586 INFO [HbmBinder] Mapping class: org.jboss.soa.esb.monitoring.pojo.JMXOperation -> JMXOPERATION
19:39:10,587 INFO [Configuration] Configured SessionFactory: null
19:39:10,588 INFO [NamingHelper] JNDI InitialContext properties:{}
19:39:10,588 INFO [DataSourceConnectionProvider] Using datasource: java:/ManagementDS
19:39:10,589 INFO [SettingsFactory] RDBMS: HSQL Database Engine, version: 1.8.0

```

Now that we have the server running, we can try running a quickstart. First, we need to setup the quickstart configuration. So, you'll need to do this in a new console window:

```

cd $LAB_DIR/jboss-soa-p.4.3.0/jboss-as/samples/quickstarts/conf
mv quickstarts.properties-example quickstarts.properties

```

Then, edit the new quickstarts.properties with your favorite file editor (VI, Emacs, Notepad, etc.).

```

apestel@localhost:~/lab/jboss-soa-p.4.3.0/jboss-as/samples/quickstarts/conf
File Edit View Terminal Tabs Help
#####
#
# Environment specific properties for executing the quickstarts.
#
#####
# Location of your JBoss Application Server installation.
# Will override the same property name from install/deployment.properties
org.jboss.esb.server.home=/home/apestel/lab/jboss-soa-p.4.3.0/jboss-as
# JBossAS server name. If not set defaults to 'default'
# Will override the same property name from install/deployment.properties
org.jboss.esb.server.config=default
# jBPM console security credentials (if org.jboss.esb.server.config=production)
jbpm.console.username=admin
jbpm.console.password=admin

```

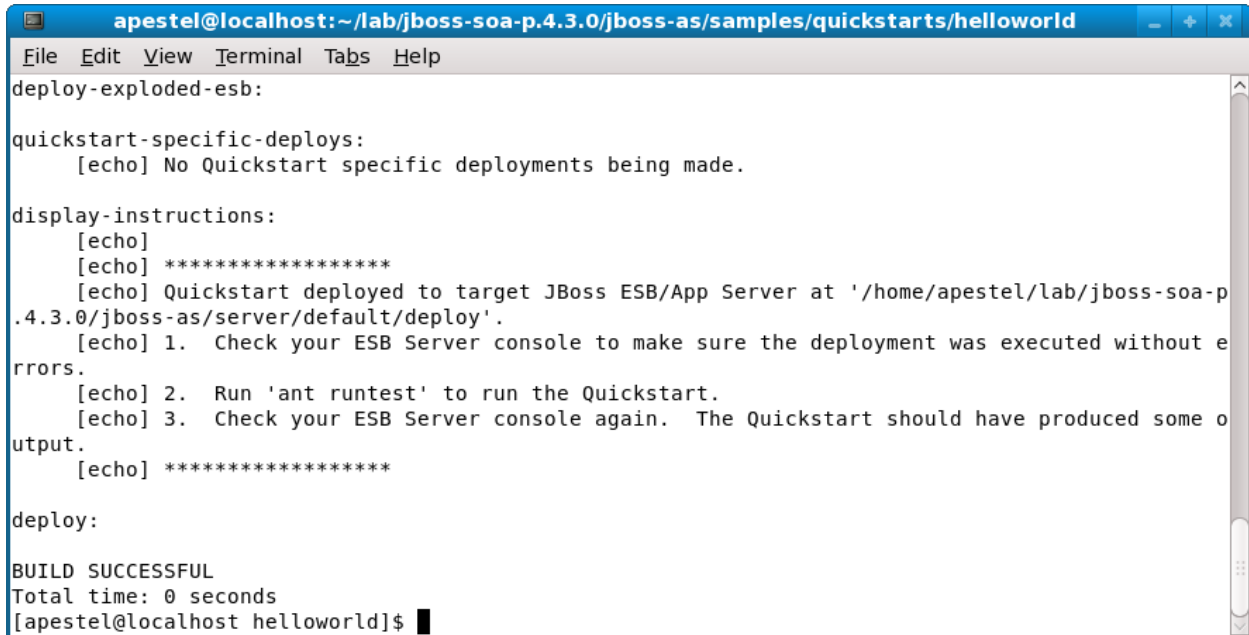
As shown above, there are four things you need to change in this file:

- 1.) Uncomment the org.jboss.esb.server.home line
- 2.) Set the correct home directory (note that it includes “jboss-as” at the end)
- 3.) Uncomment the org.jboss.esb.server.config line
- 4.) Uncomment the two jbpm.console.* lines.

Now that we've set the quickstart configurations, we can try running a quickstart. So, change directories to \$LAB_DIR/jboss-soa-p.4.3.0/jboss-as/samples/quickstarts/helloworld

Most all the quickstarts are setup similarly. All source code for the complete example is in this directory.

To build and deploy the example, we need to run “ant deploy” as shown below. If you don't have Ant installed, you can skip to the end of this lab, but you'll probably want to try it later as these quickstarts have a wealth of examples in them.



```

apestel@localhost:~/lab/jboss-soa-p.4.3.0/jboss-as/samples/quickstarts/helloworld
File Edit View Terminal Tabs Help
deploy-exploded-esb:

quickstart-specific-deploys:
  [echo] No Quickstart specific deployments being made.

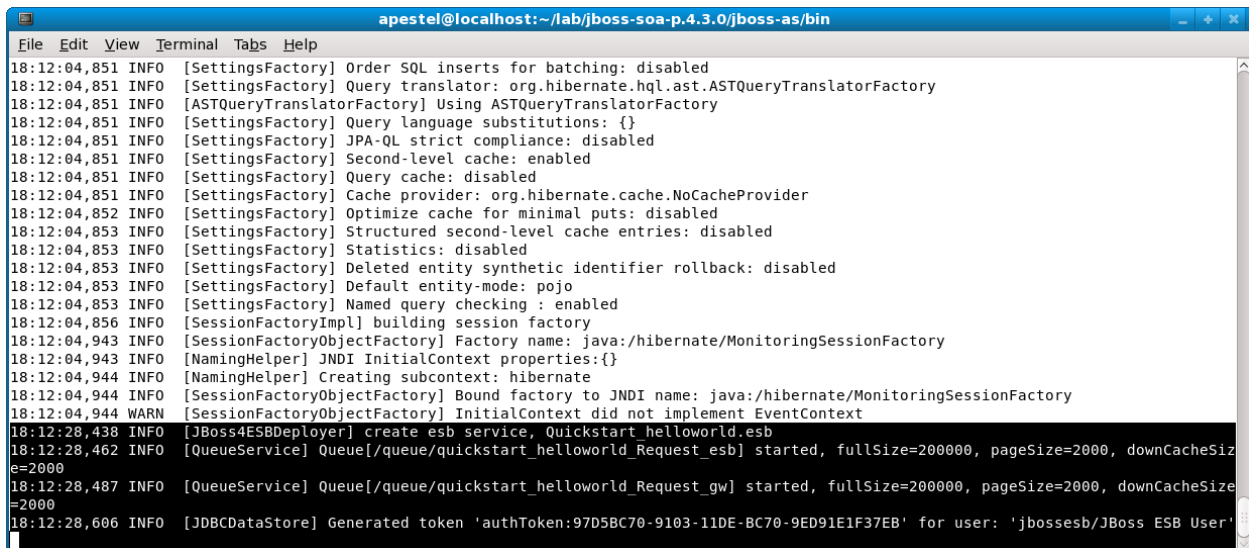
display-instructions:
  [echo]
  [echo] *****
  [echo] Quickstart deployed to target JBoss ESB/App Server at '/home/apestel/lab/jboss-soa-p
.4.3.0/jboss-as/server/default/deploy'.
  [echo] 1. Check your ESB Server console to make sure the deployment was executed without e
rrors.
  [echo] 2. Run 'ant runtest' to run the Quickstart.
  [echo] 3. Check your ESB Server console again. The Quickstart should have produced some o
utput.
  [echo] *****

deploy:

BUILD SUCCESSFUL
Total time: 0 seconds
[apestel@localhost helloworld]$

```

If you look in your server window, you should see the the example deployed:



```

apestel@localhost:~/lab/jboss-soa-p.4.3.0/jboss-as/bin
File Edit View Terminal Tabs Help
18:12:04,851 INFO [SettingsFactory] Order SQL inserts for batching: disabled
18:12:04,851 INFO [SettingsFactory] Query translator: org.hibernate.hql.ast.ASTQueryTranslatorFactory
18:12:04,851 INFO [ASTQueryTranslatorFactory] Using ASTQueryTranslatorFactory
18:12:04,851 INFO [SettingsFactory] Query language substitutions: {}
18:12:04,851 INFO [SettingsFactory] JPA-QL strict compliance: disabled
18:12:04,851 INFO [SettingsFactory] Second-level cache: enabled
18:12:04,851 INFO [SettingsFactory] Query cache: disabled
18:12:04,851 INFO [SettingsFactory] Cache provider: org.hibernate.cache.NoCacheProvider
18:12:04,852 INFO [SettingsFactory] Optimize cache for minimal puts: disabled
18:12:04,853 INFO [SettingsFactory] Structured second-level cache entries: disabled
18:12:04,853 INFO [SettingsFactory] Statistics: disabled
18:12:04,853 INFO [SettingsFactory] Deleted entity synthetic identifier rollback: disabled
18:12:04,853 INFO [SettingsFactory] Default entity-mode: pojo
18:12:04,853 INFO [SettingsFactory] Named query checking : enabled
18:12:04,856 INFO [SessionFactoryImpl] building session factory
18:12:04,943 INFO [SessionFactoryObjectFactory] Factory name: java:/hibernate/MonitoringSessionFactory
18:12:04,943 INFO [NamingHelper] JNDI InitialContext properties:{}
18:12:04,944 INFO [NamingHelper] Creating subcontext: hibernate
18:12:04,944 INFO [SessionFactoryObjectFactory] Bound factory to JNDI name: java:/hibernate/MonitoringSessionFactory
18:12:04,944 WARN [SessionFactoryObjectFactory] InitialContext did not implement EventContext
18:12:28,438 INFO [JBoss4ESBDeployer] create esb service, Quickstart helloworld.esb
18:12:28,462 INFO [QueueService] Queue[/queue/quickstart_helloworld_Request_esb] started, fullSize=200000, pageSize=2000, downCacheSize=2000
18:12:28,487 INFO [QueueService] Queue[/queue/quickstart_helloworld_Request_gw] started, fullSize=200000, pageSize=2000, downCacheSize=2000
18:12:28,606 INFO [JDBCDataStore] Generated token 'authToken:97D5BC70-9103-11DE-BC70-9ED91E1F37EB' for user: 'jbossesb/JBoss ESB User'

```

This particular example creates a very simple ESB service that has a JMS gateway and sends a JMS message with the text “Hello World”.

To run the client that publishes the message for the ESB to pick up, just run “ant runtest” from the quickstart window. Resulting output is shown below:



```

apestel@localhost:~/lab/jboss-soa-p.4.3.0/jboss-as/samples/quickstarts/helloworld
File Edit View Terminal Tabs Help

messaging-dependencies:

jbossmq-dependencies:

quickstart-specific-dependencies:

classpath-dependencies:

quickstart-specific-checks:

dependencies:

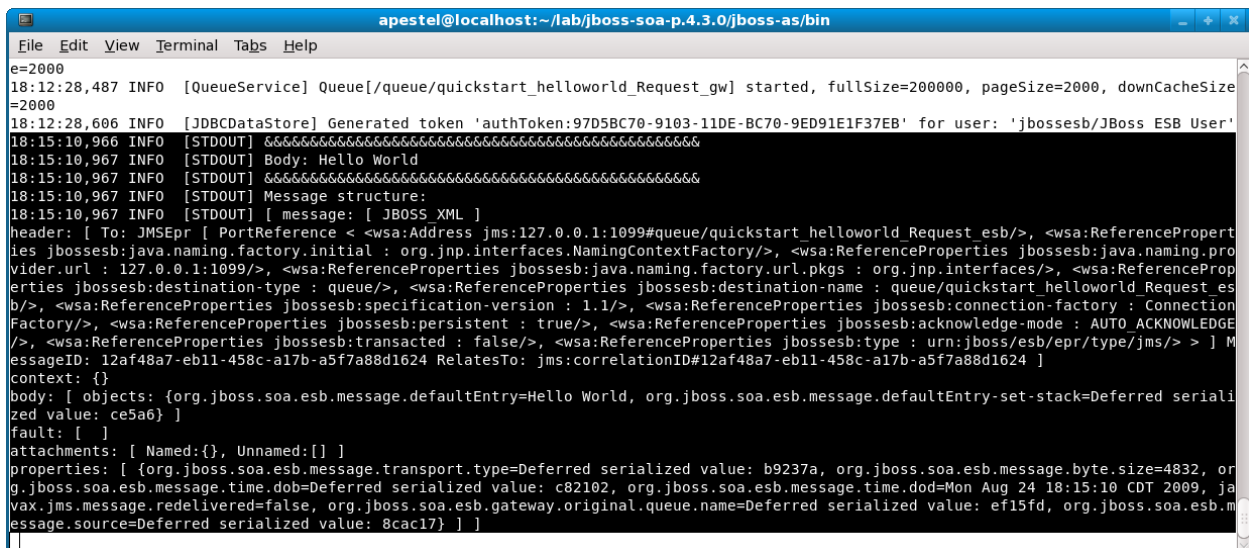
compile:

runtest:
  [echo] Runs Test JMS Sender
  [java] Connection Started

BUILD SUCCESSFUL
Total time: 3 seconds
[apestel@localhost helloworld]$

```

If you again look at your server console window, you'll see that the ESB received the message and printed out the contents (including the entire ESB message):



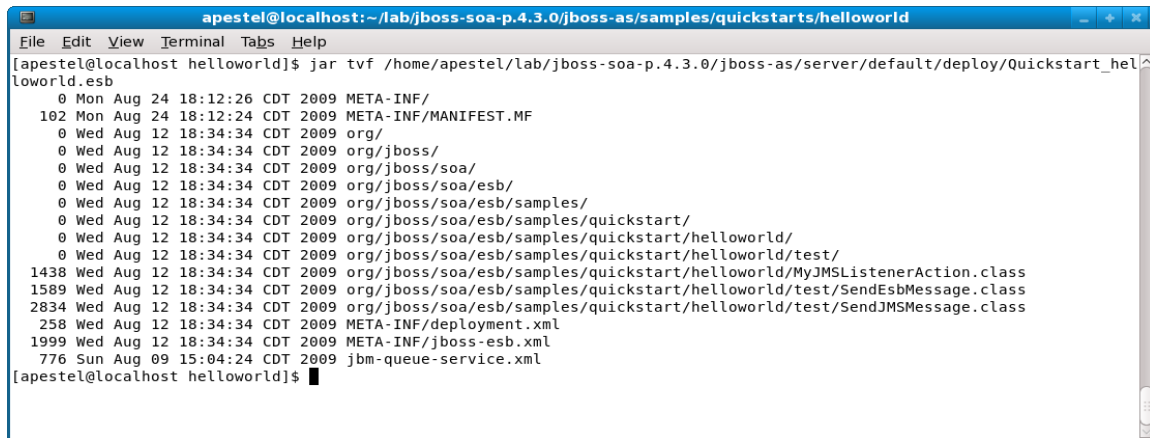
```

apestel@localhost:~/lab/jboss-soa-p.4.3.0/jboss-as/bin
File Edit View Terminal Tabs Help

e=2000
18:12:28,487 INFO [QueueService] Queue[/queue/quickstart_helloworld_Request_gw] started, fullSize=200000, pageSize=2000, downCacheSize=2000
18:12:28,606 INFO [JDBCDataStore] Generated token 'authToken:97D5BC70-9103-11DE-BC70-9ED91E1F37EB' for user: 'jbossesb/JBoss ESB User'
18:15:10,966 INFO [STDOUT] ~~~~~
18:15:10,967 INFO [STDOUT] Body: Hello World
18:15:10,967 INFO [STDOUT] ~~~~~
18:15:10,967 INFO [STDOUT] Message structure:
18:15:10,967 INFO [STDOUT] [ message: [ JBOSS_XML ]
header: [ To: JMSEpr [ PortReference < wsa:Address jms:127.0.0.1:1099#queue/quickstart_helloworld_Request_esb/>, < wsa:ReferenceProperties jbossesb:java.naming.factory.initial : org.jnp.interfaces.NamingContextFactory/>, < wsa:ReferenceProperties jbossesb:java.naming.provider.url : 127.0.0.1:1099/>, < wsa:ReferenceProperties jbossesb:java.naming.factory.url.pkgs : org.jnp.interfaces/>, < wsa:ReferenceProperties jbossesb:destination-type : queue/>, < wsa:ReferenceProperties jbossesb:destination-name : queue/quickstart_helloworld_Request_esb/>, < wsa:ReferenceProperties jbossesb:specification-version : 1.1/>, < wsa:ReferenceProperties jbossesb:connection-factory : ConnectionFactory/>, < wsa:ReferenceProperties jbossesb:persistent : true/>, < wsa:ReferenceProperties jbossesb:acknowledge-mode : AUTO ACKNOWLEDGE/>, < wsa:ReferenceProperties jbossesb:transacted : false/>, < wsa:ReferenceProperties jbossesb:type : urn:jboss/esb/epr/type/jms/> > ]
messageID: 12af48a7-eb11-458c-a17b-a5f7a88d1624 RelatesTo: jms:correlationID#12af48a7-eb11-458c-a17b-a5f7a88d1624 ]
context: {}
body: [ objects: { org.jboss.soa.esb.message.defaultEntry=Hello World, org.jboss.soa.esb.message.defaultEntry-set-stack=Deferred serialized value: ce5a6 } ]
fault: [ ]
attachments: [ Named:{}, Unnamed:[ ] ]
properties: [ {org.jboss.soa.esb.message.transport.type=Deferred serialized value: b9237a, org.jboss.soa.esb.message.byte.size=4832, org.jboss.soa.esb.message.time.dob=Deferred serialized value: c82102, org.jboss.soa.esb.message.time.dod=Mon Aug 24 18:15:10 CDT 2009, javax.jms.message.redelivered=false, org.jboss.soa.esb.gateway.original.queue.name=Deferred serialized value: ef15fd, org.jboss.soa.esb.message.source=Deferred serialized value: 8cac17 } ]

```

So, what was in that ESB deployment? There is not too much, actually. It's just a single “.esb” file that was deployed to the server's deploy directory. We can see what's in that file by running “jar tvf \$LAB_DIR/jboss-soa-p.4.3.0/jboss-as/server/default/deploy/Quickstart_helloworld.esb” as shown below.



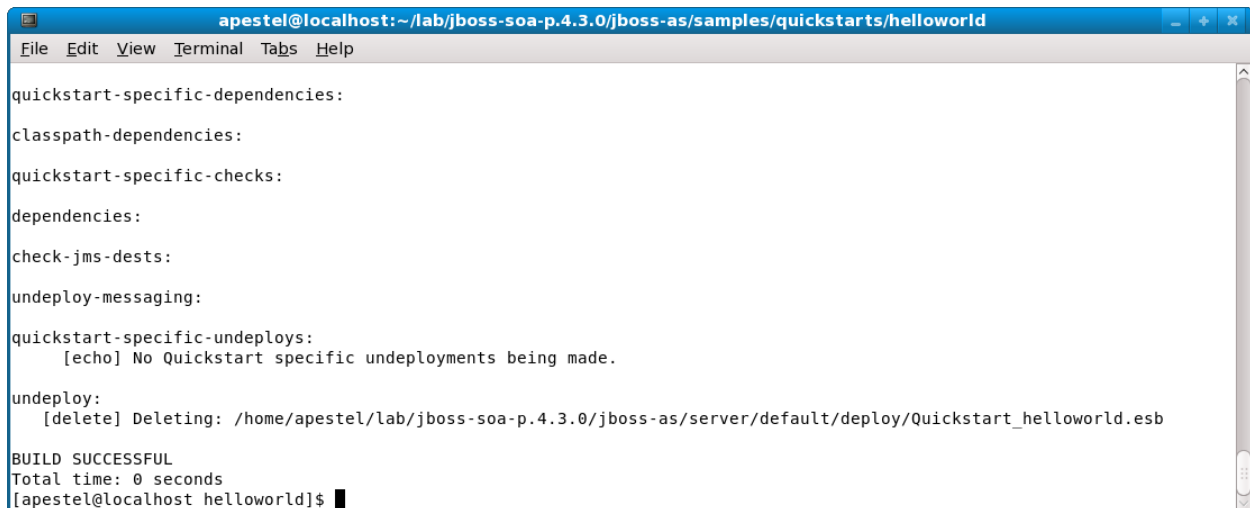
```

apestel@localhost:~/lab/jboss-soa-p.4.3.0/jboss-as/samples/quickstarts/helloworld
[apestel@localhost helloworld]$ jar tvf /home/apestel/lab/jboss-soa-p.4.3.0/jboss-as/server/default/deploy/Quickstart_helloworld.esb
 0 Mon Aug 24 18:12:26 CDT 2009 META-INF/
102 Mon Aug 24 18:12:24 CDT 2009 META-INF/MANIFEST.MF
 0 Wed Aug 12 18:34:34 CDT 2009 org/
 0 Wed Aug 12 18:34:34 CDT 2009 org/jboss/
 0 Wed Aug 12 18:34:34 CDT 2009 org/jboss/soa/
 0 Wed Aug 12 18:34:34 CDT 2009 org/jboss/soa/esb/
 0 Wed Aug 12 18:34:34 CDT 2009 org/jboss/soa/esb/samples/
 0 Wed Aug 12 18:34:34 CDT 2009 org/jboss/soa/esb/samples/quickstart/
 0 Wed Aug 12 18:34:34 CDT 2009 org/jboss/soa/esb/samples/quickstart/helloworld/
 0 Wed Aug 12 18:34:34 CDT 2009 org/jboss/soa/esb/samples/quickstart/helloworld/test/
1438 Wed Aug 12 18:34:34 CDT 2009 org/jboss/soa/esb/samples/quickstart/helloworld/MyJMSListenerAction.class
1589 Wed Aug 12 18:34:34 CDT 2009 org/jboss/soa/esb/samples/quickstart/helloworld/test/SendEsbMessage.class
2834 Wed Aug 12 18:34:34 CDT 2009 org/jboss/soa/esb/samples/quickstart/helloworld/test/SendJMSMessage.class
258 Wed Aug 12 18:34:34 CDT 2009 META-INF/deployment.xml
1999 Wed Aug 12 18:34:34 CDT 2009 META-INF/jboss-esb.xml
776 Sun Aug 09 15:04:24 CDT 2009 jbm-queue-service.xml
[apestel@localhost helloworld]$

```

So there are three Java classes. The two in the test directory are actually only client classes used to send the JMS message and not needed in the ESB deployment. The `MyJMSListenerAction` is a custom action that prints out what we saw in the server console. The `jbm-queue-service.xml` file defines the JMS queues that get deployed. `deployment.xml` declares that the ESB services depend on those queues, and `jboss-esb.xml` defines the services. That's it! Please feel free to look over this example in more detail. You can even change the source code in the quickstart directory and “ant deploy” will re-compile and re-deploy.

To undeploy the example, just run “ant undeploy” from the quickstart directory as shown below:



```

apestel@localhost:~/lab/jboss-soa-p.4.3.0/jboss-as/samples/quickstarts/helloworld
File Edit View Terminal Tabs Help
quickstart-specific-dependencies:
classpath-dependencies:
quickstart-specific-checks:
dependencies:
check-jms-dests:
undeploy-messaging:
quickstart-specific-undeploys:
 [echo] No Quickstart specific undeployments being made.
undeploy:
 [delete] Deleting: /home/apestel/lab/jboss-soa-p.4.3.0/jboss-as/server/default/deploy/Quickstart_helloworld.esb
BUILD SUCCESSFUL
Total time: 0 seconds
[apestel@localhost helloworld]$

```

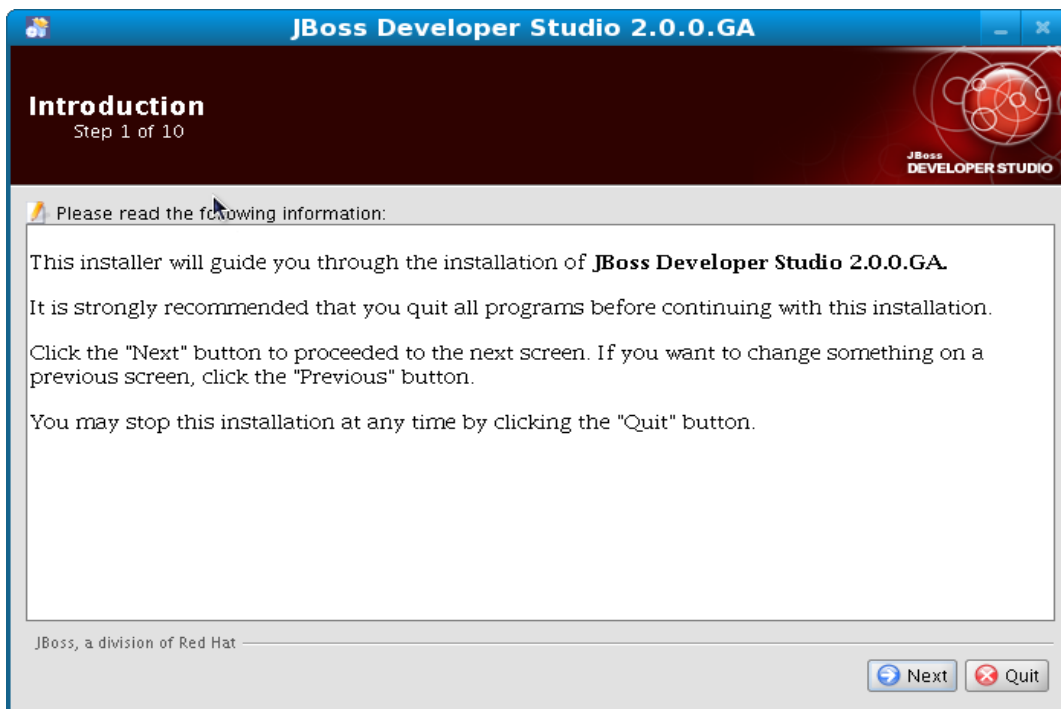
Look at the server console window should show that the ESB deployment was undeployed:


```

apestel@localhost:~/lab
File Edit View Terminal Tabs Help
[apestel@localhost lab]$ java -jar jbdevstudio-linux-gtk-2.0.0.GA-R200903141247-H4.jar
PackageListener, install.log=/tmp/jbdevstudio-install-41190.log
Locking assertion failure. Backtrace:
#0 /usr/lib/libxcb-xlib.so.0 [0x864767]
#1 /usr/lib/libxcb-xlib.so.0(xcb_xlib_unlock+0x31) [0x864831]
#2 /usr/lib/libX11.so.6(_XReply+0x244) [0x694bf64]
#3 /usr/java/jdk1.5.0_16/jre/lib/i386/xawt/libmawt.so [0x62220dce]
#4 /usr/java/jdk1.5.0_16/jre/lib/i386/xawt/libmawt.so [0x6220ad77]
#5 /usr/java/jdk1.5.0_16/jre/lib/i386/xawt/libmawt.so [0x6220aef3]
#6 /usr/java/jdk1.5.0_16/jre/lib/i386/xawt/libmawt.so(Java_sun_awt_X11GraphicsEnvironment_init
Display+0x26) [0x6220b136]
#7 [0xb1686008]
#8 [0xb167fb6b]
#9 [0xb167fb6b]
#10 [0xb167d236]
#11 /usr/java/jdk1.5.0_16/jre/lib/i386/server/libjvm.so [0xb78beeec]
#12 /usr/java/jdk1.5.0_16/jre/lib/i386/server/libjvm.so [0xb7a8eae8]
#13 /usr/java/jdk1.5.0_16/jre/lib/i386/server/libjvm.so [0xb78bed1f]
#14 /usr/java/jdk1.5.0_16/jre/lib/i386/server/libjvm.so(JVM_DoPrivileged+0x32d) [0xb791c82d]
#15 /usr/java/jdk1.5.0_16/jre/lib/i386/libjava.so(Java_java_security_AccessController_doPrivil
eged_Ljava_security_PrivilegedAction_2+0x3d) [0xb761c30d]
#16 [0xb1685898]

```

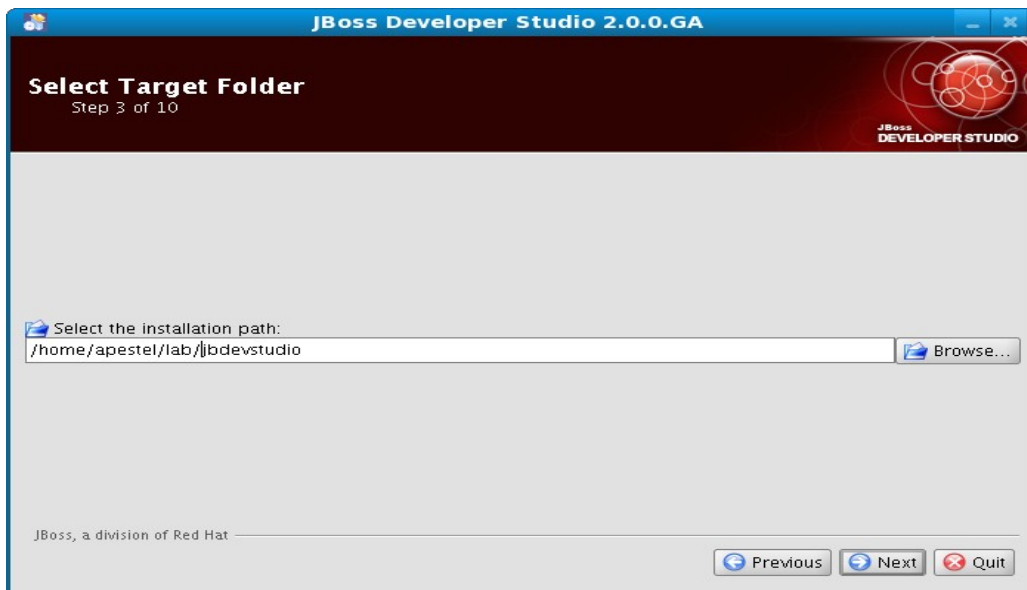
This should open a GUI installer. We'll now walk through the steps of that installer.



Select the Next Button and this will get you to this screen:



Select the "I accept..." radio button and click the "Next" button.



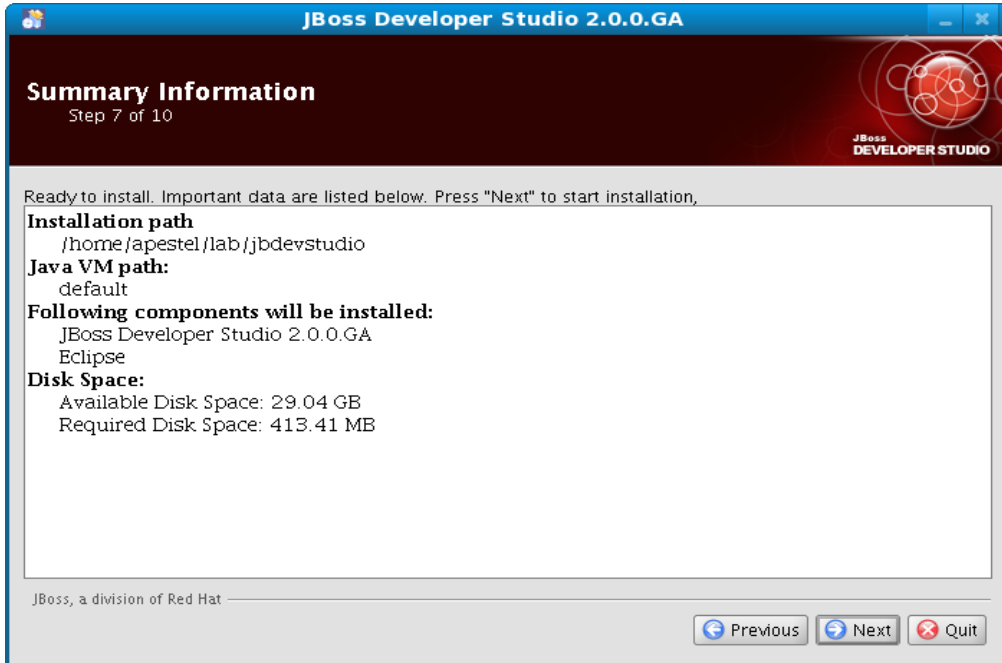
Select the folder you want JBDS installed in and click "Next"



Select "Next".



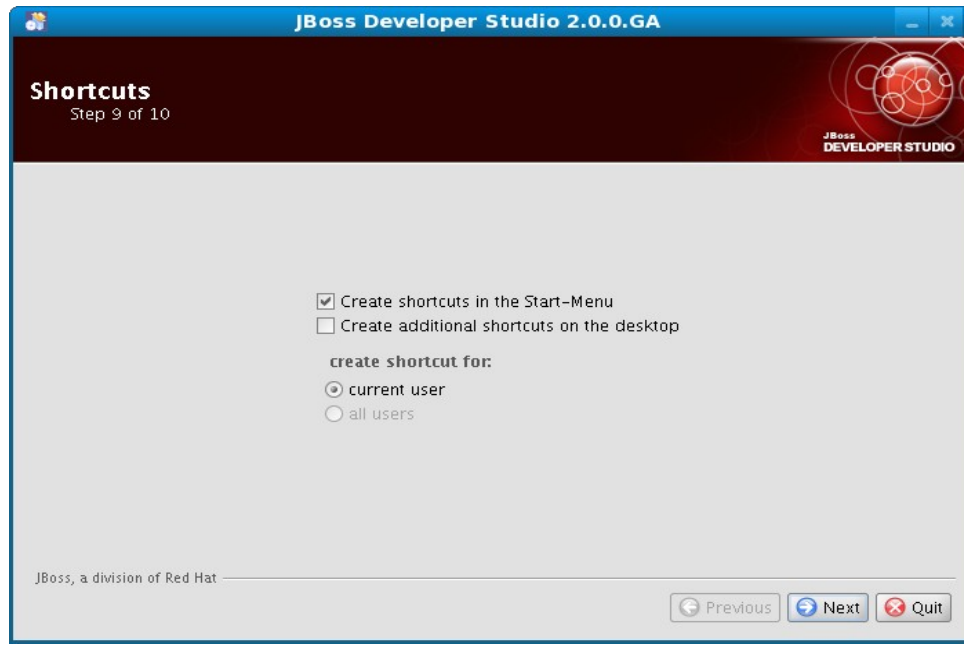
Click "Next" (EAP may be used for the Seam lab, so we'll include it in case you do that lab as well).



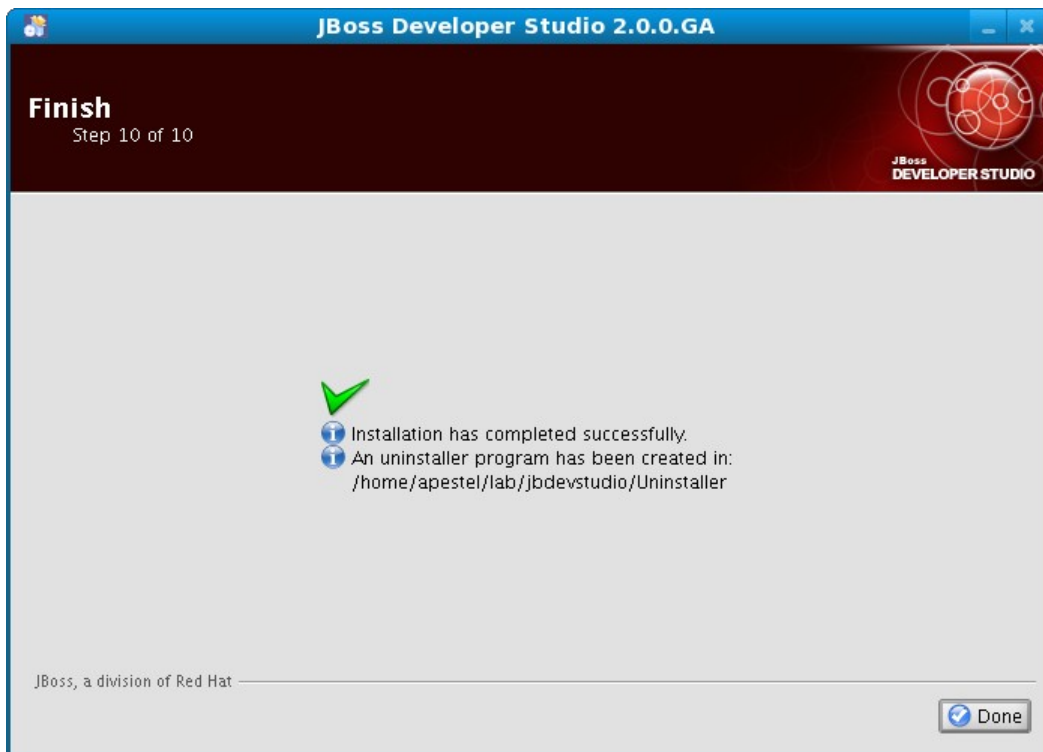
Click Next



Your screen will likely say 3/3 instead of 2/2 as in this older screenshot. Click "Next".



Click "Next".

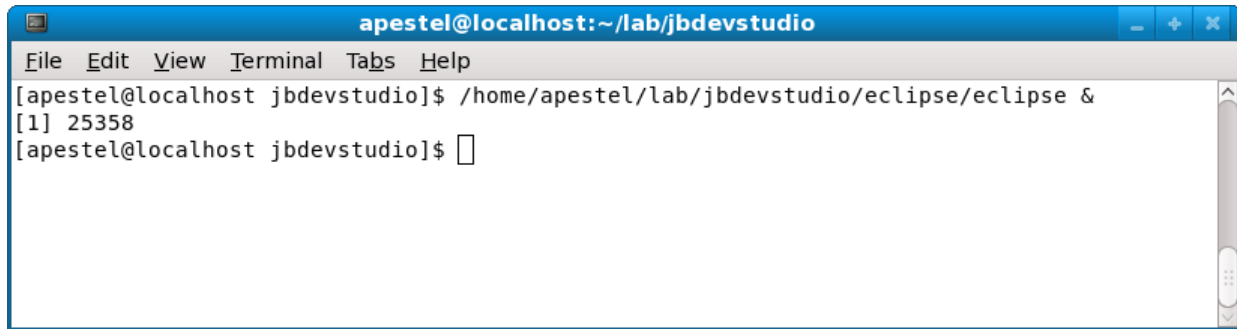


Click "Done". Congratulations, JBDS is now installed!

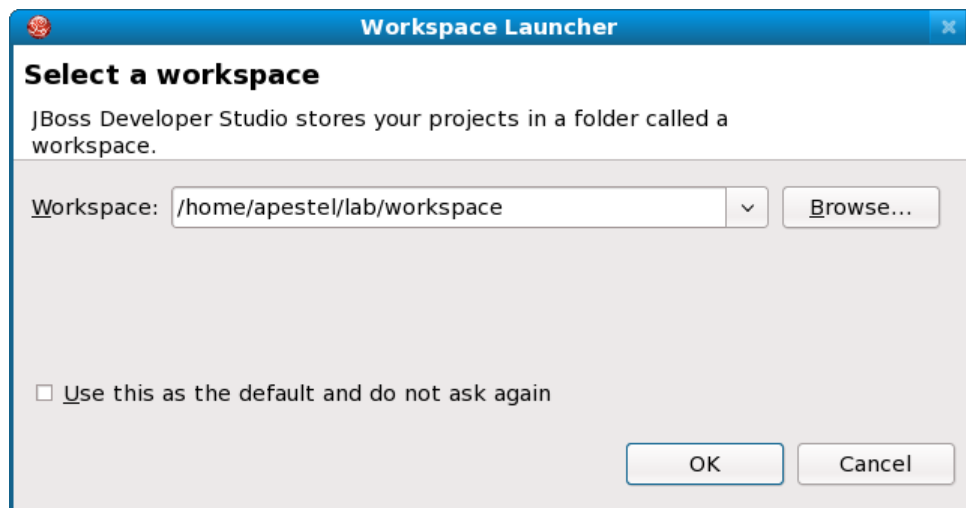
Lab #4: Configure SOA-P in JBDS

The SOA Platform can be configured as a “server” in Eclipse just like any other JEE server. In this way, we can deploy JEE applications to the server and/or ESB applications to the server. To set this up, we first need to start JBDS.

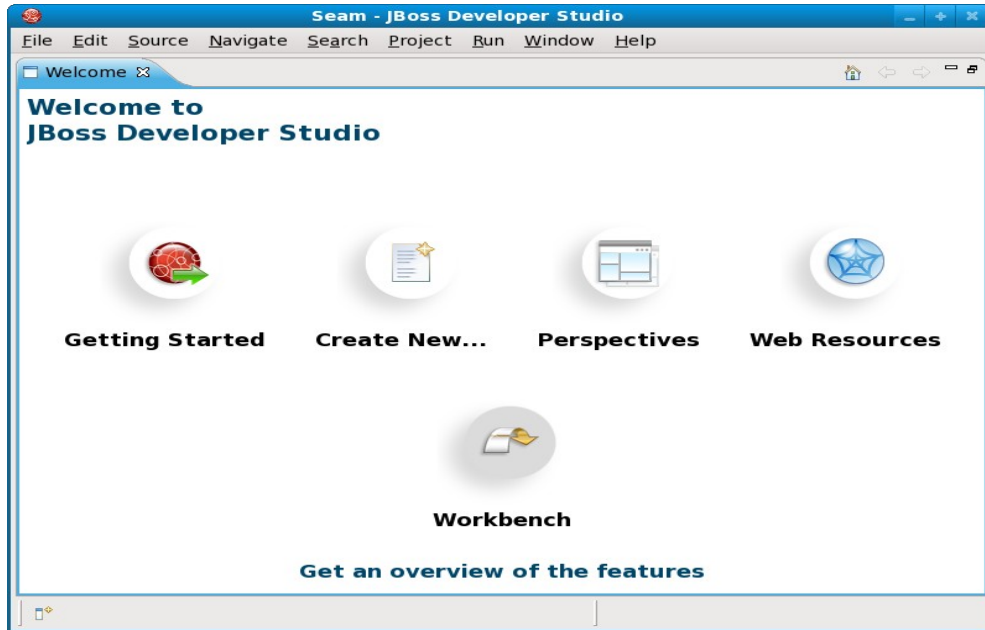
This can be done via the shortcuts created during installation or via the command line. We'll show starting via command line below:



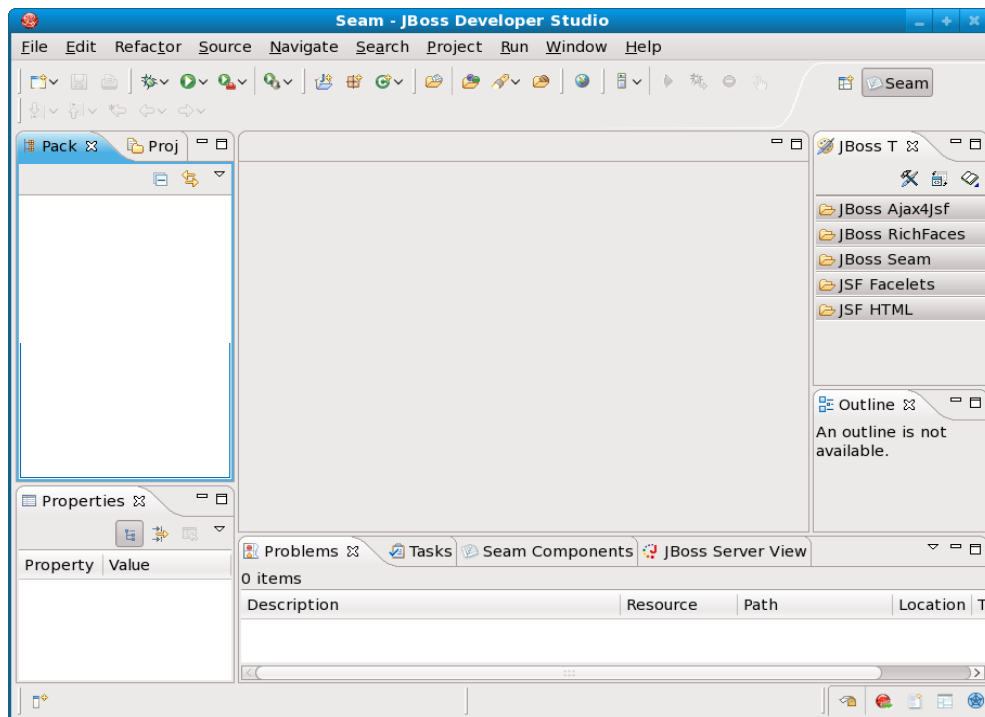
```
apestel@localhost:~/lab/jbdevstudio
File Edit View Terminal Tabs Help
[apestel@localhost jbdevstudio]$ /home/apestel/lab/jbdevstudio/eclipse/eclipse &
[1] 25358
[apestel@localhost jbdevstudio]$
```



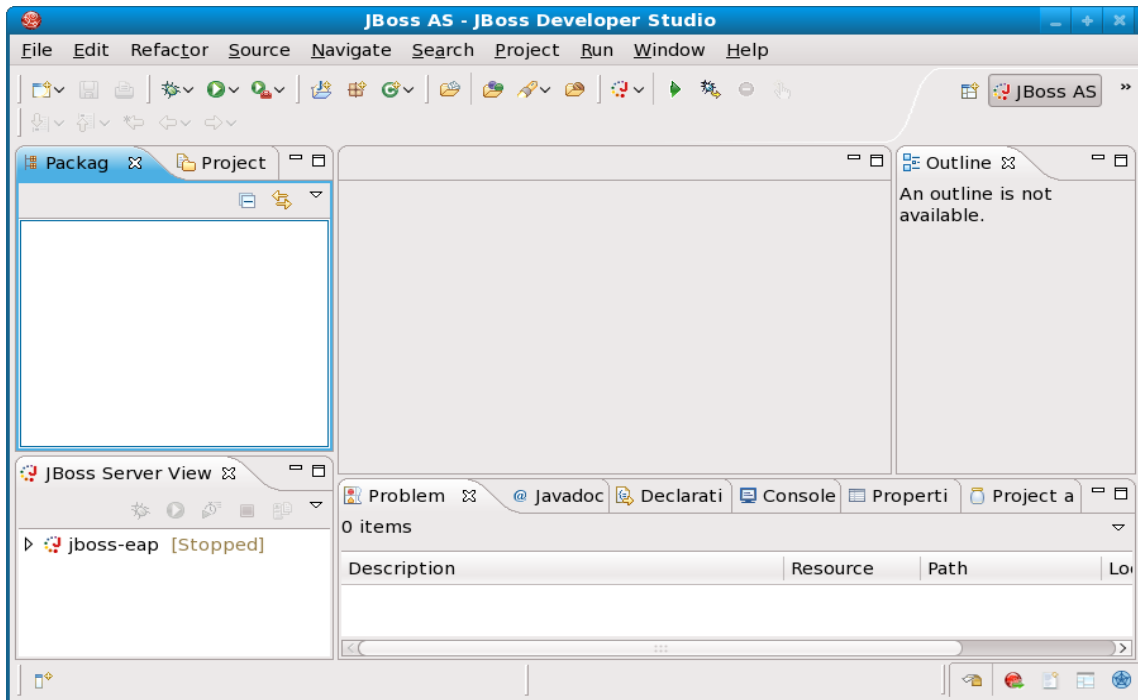
We need to select a workspace where our projects will be stored. Select your desired workspace location and click “OK”.



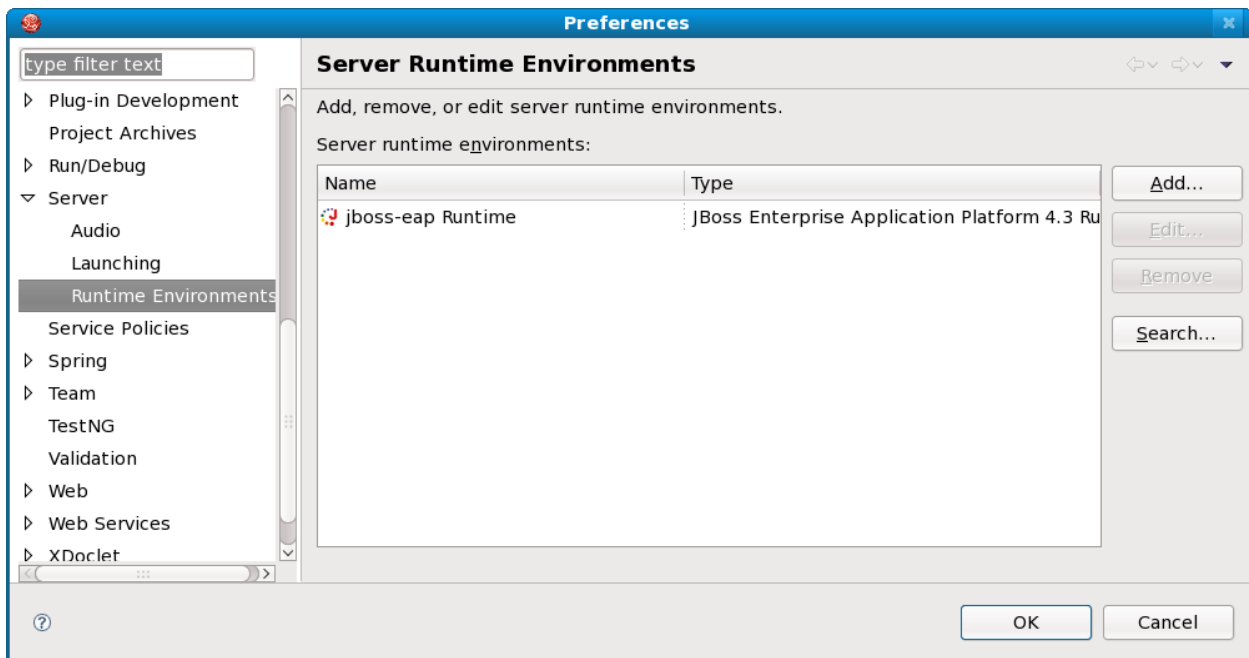
This is the default screen for a new workspace. In this lab, we want to go straight to the Workbench by clicking “Workbench”.



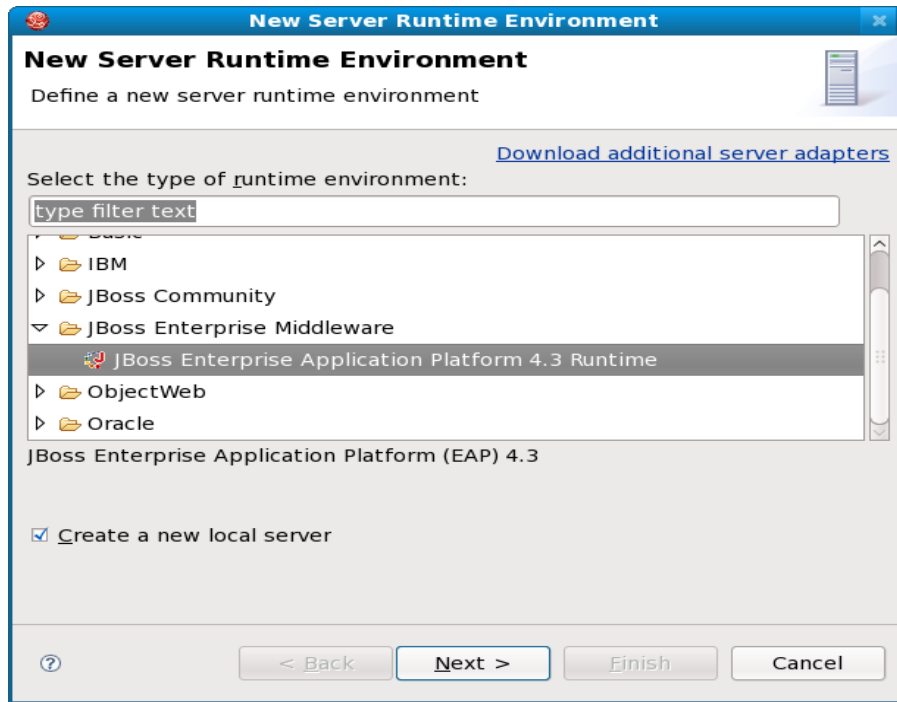
This is an empty workspace. The first thing we want to do is change to the “JBoss AS”. To do this, select “Window | Open Perspective | Other... | JBoss AS” and click “OK”. Now the JBDS window should look like this below:



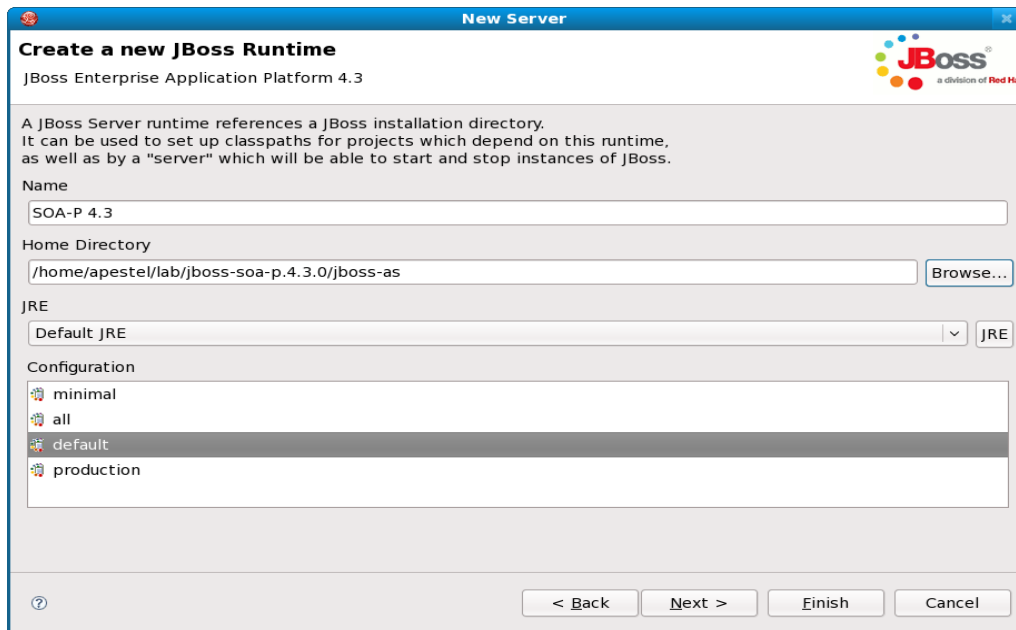
We need to create a new server for SOA-P. The first thing we want to do is create a runtime for that server. We could just right-click in the JBoss Server View and create a new server. However, the screenshots for that will differ depending on if you already have an EAP server configured shown above or if you do not. So, we're going to do it a different way - which will help you learn about server runtimes anyhow. So, click "Window | Preferences" and select the "Server | Runtime Environments" as shown below.



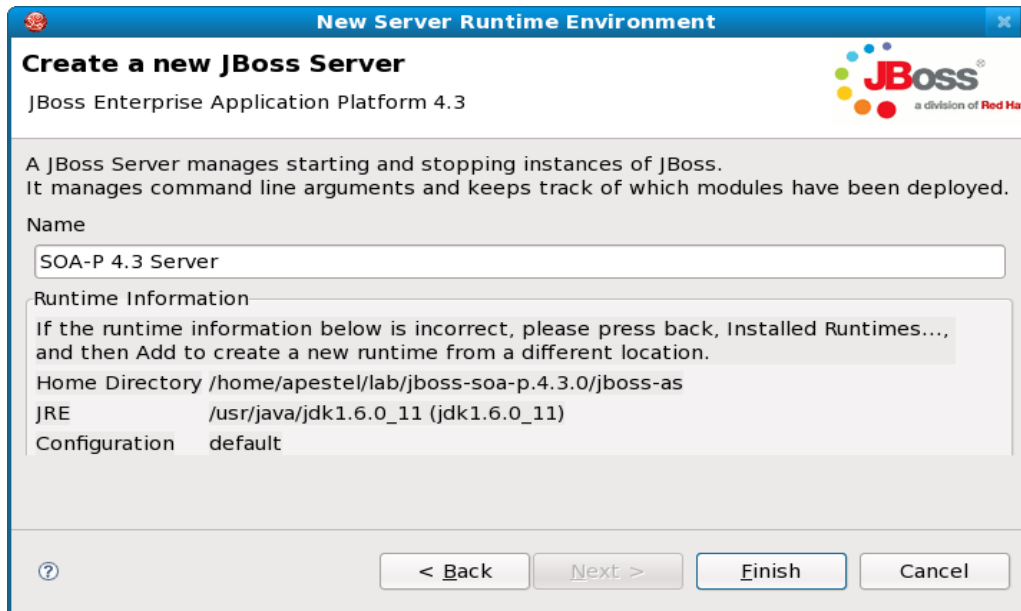
We need to add a runtime for our SOA-P server. So, click the “Add” button. Make sure to select the “JBoss Enterprise Middleware | JBoss Enterprise Application Platform 4.3 Runtime” AND select the “Create a new local server” checkbox as shown below.



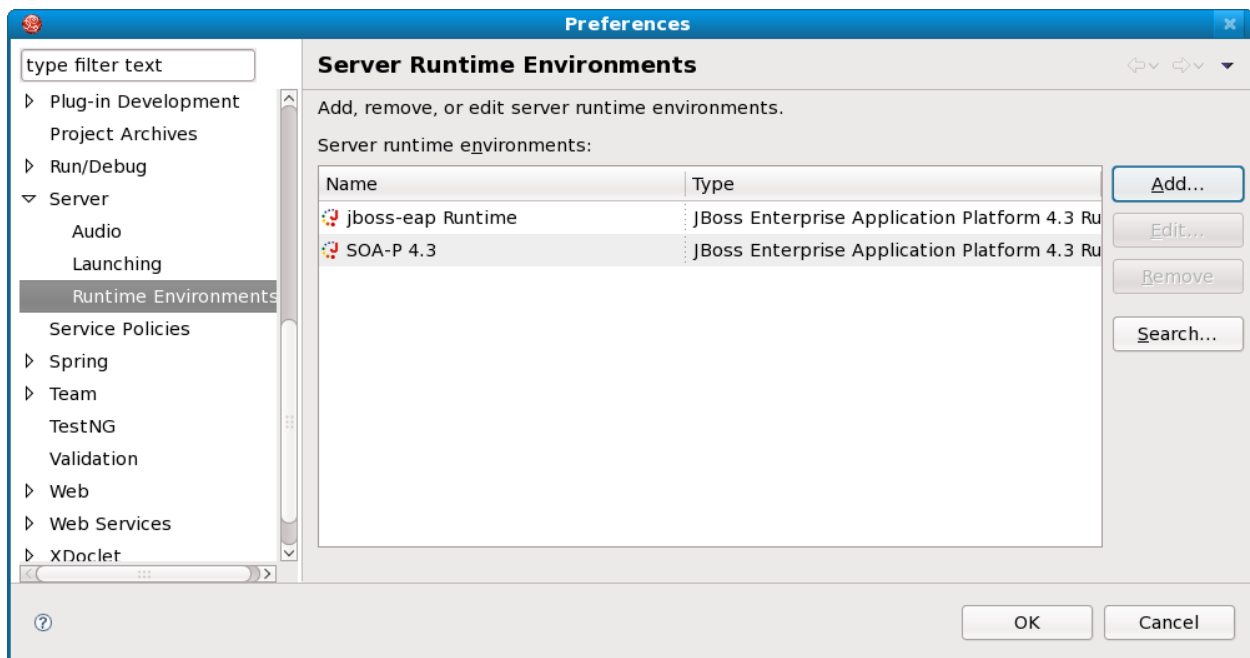
Click “Next” and fill out the dialog as shown below.



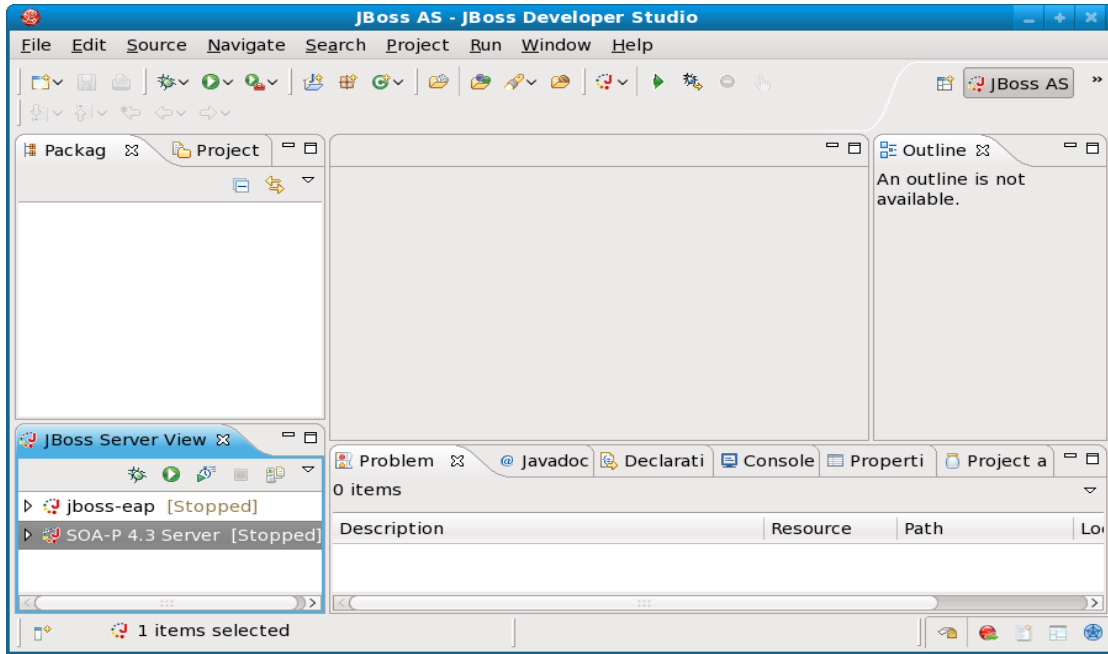
Click “Next”. Now, give the local server we are creating for this runtime a name as shown below.



Click "Finish".

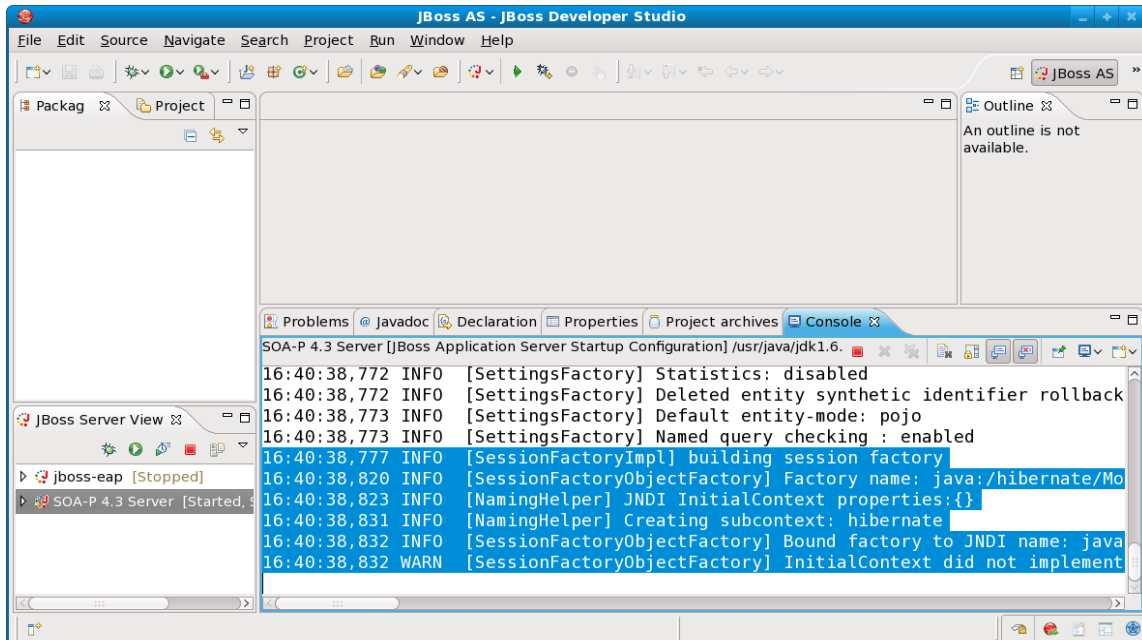


Click "OK" to complete the creation of the SOA-P 4.3 runtime. Now, we should have the SOA-P 4.3 Server in the JBoss Server View as shown below.



Before starting this server, please make sure you have stopped the server that was started in Quick-start lab #2. If we try to start it again while it is already running, we'll get port conflicts. You can kill the other server by pressing CTRL-C in servers command window.

To start this server from JBDS, we need to right-click on it and choose "Start". Sometimes the console tab does not show the output of the server. If it does not, click on the "Console" tab, select the dropdown icon in the top right corner of the console tab and choose "Java Stack Trace Console". Then, you should see more icons to the left of the dropdown you selected. Select the next dropdown to the left and choose "SOA-P 4.3 Server". We will want to be able to see the server output in the console as shown below.

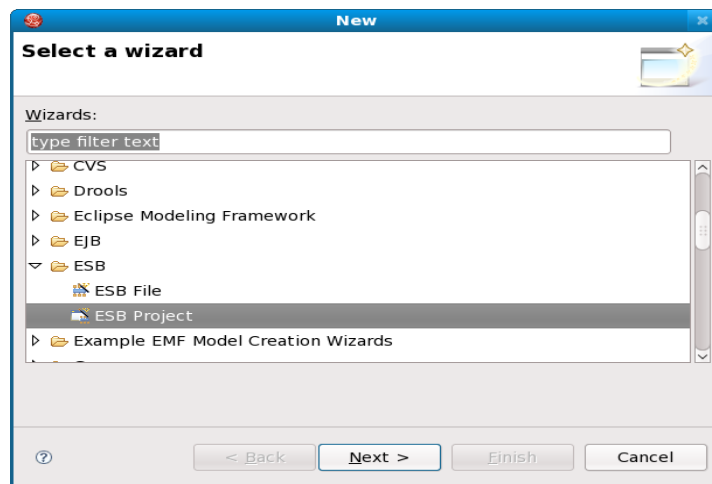


Please raise your hand if you do not see the console output. Congratulations, you have successfully started the SOA-P server from JBDS. Now, we can create our first ESB project.

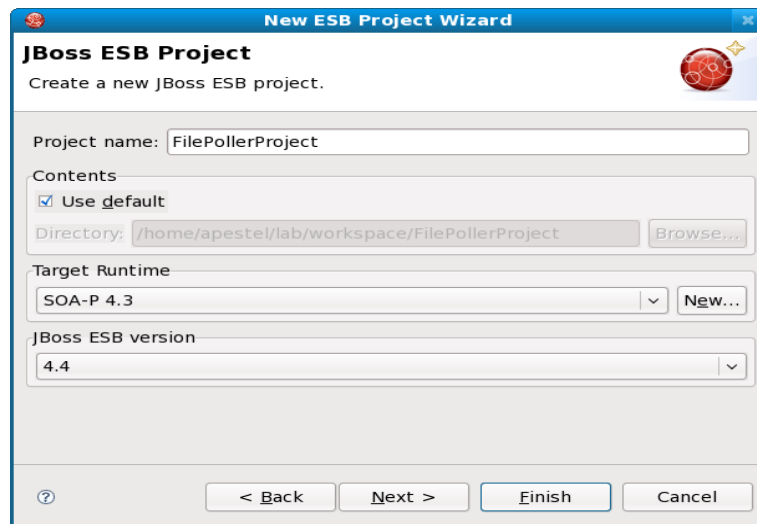
Lab #5: Creating First ESB Project

The first ESB project we are going to create is going to be a simple project with one ESB service. This ESB service will poll a directory looking for a file with a given suffix. When a matching file is placed in the poll directory, it will read the contents of the file and send them as a message on the ESB.

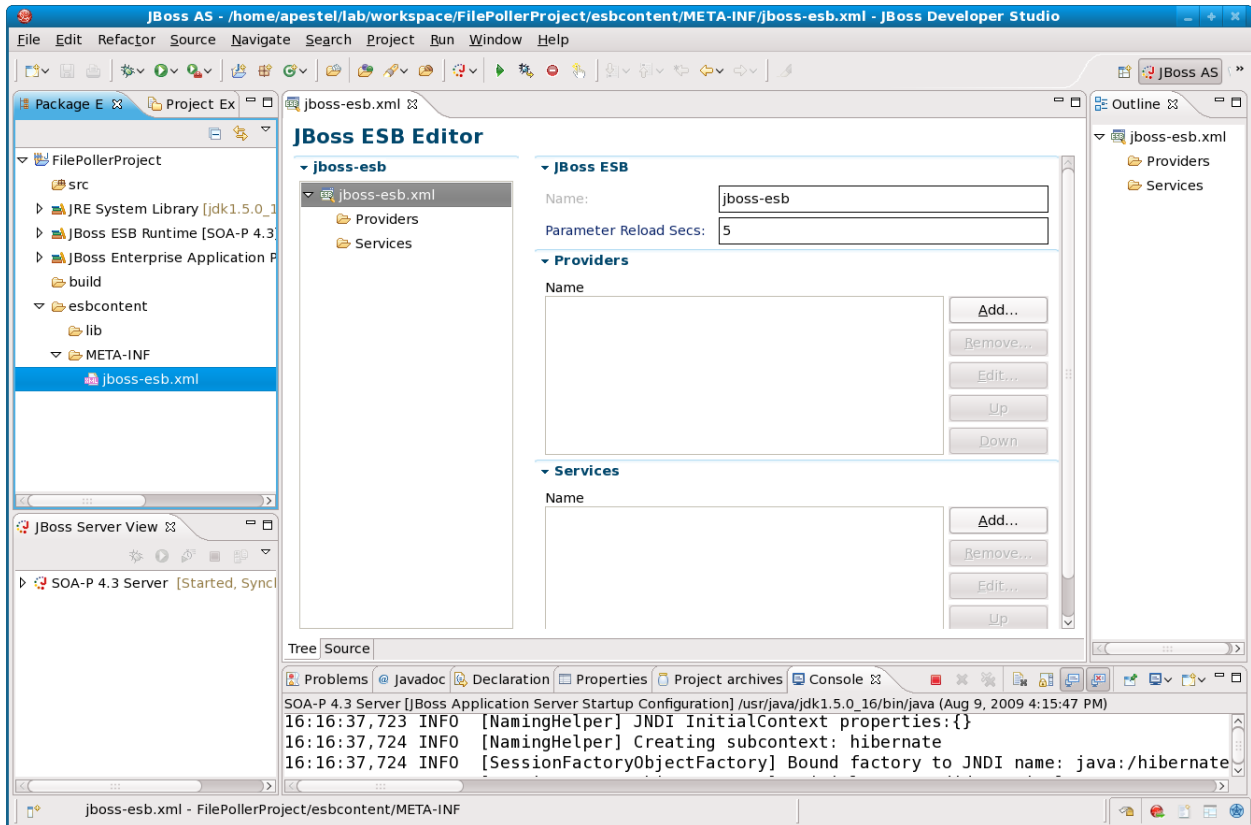
To create our ESB project, we need to select “File | New | Other”. This will give us the dialog below.



As shown above, navigate to the “ESB” folder and select “ESB Project”. Click “Next”.

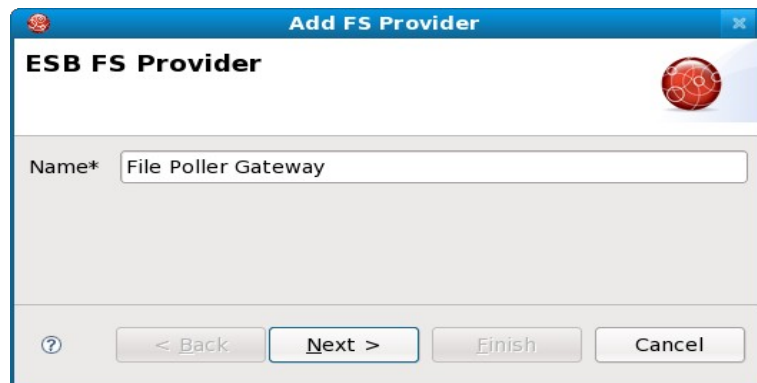


There are three things we need to specify as shown above. 1.) Provide a name “FilePollerProject”. 2.) Select the Target Runtime from the dropdown (“SOA-P 4.3” that we created earlier), 3.) Choose the ESB version which is 4.4 for this lab. After specifying these options, click “Finish”.

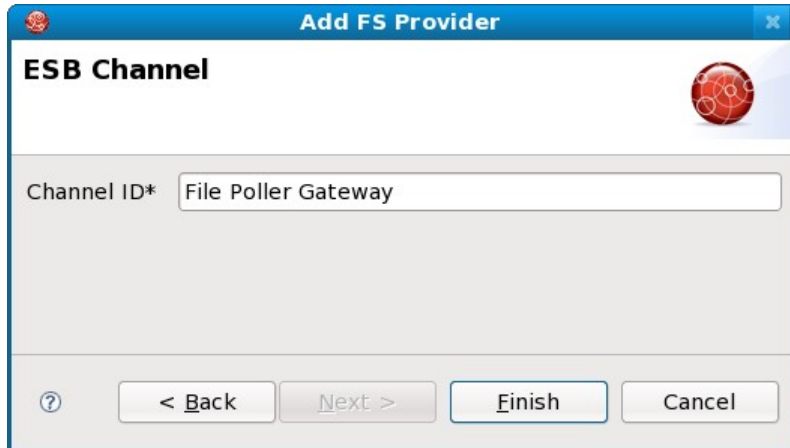


You should see that the project was created and the ESB configuration editor was automatically opened. This is actually the editor for the jboss-esb.xml file which you can find in the project by navigating to “esbcontent | META-INF | jboss-esb.xml” in the project explorer on the left of the JBDS window. It is highlighted in the screenshot above.

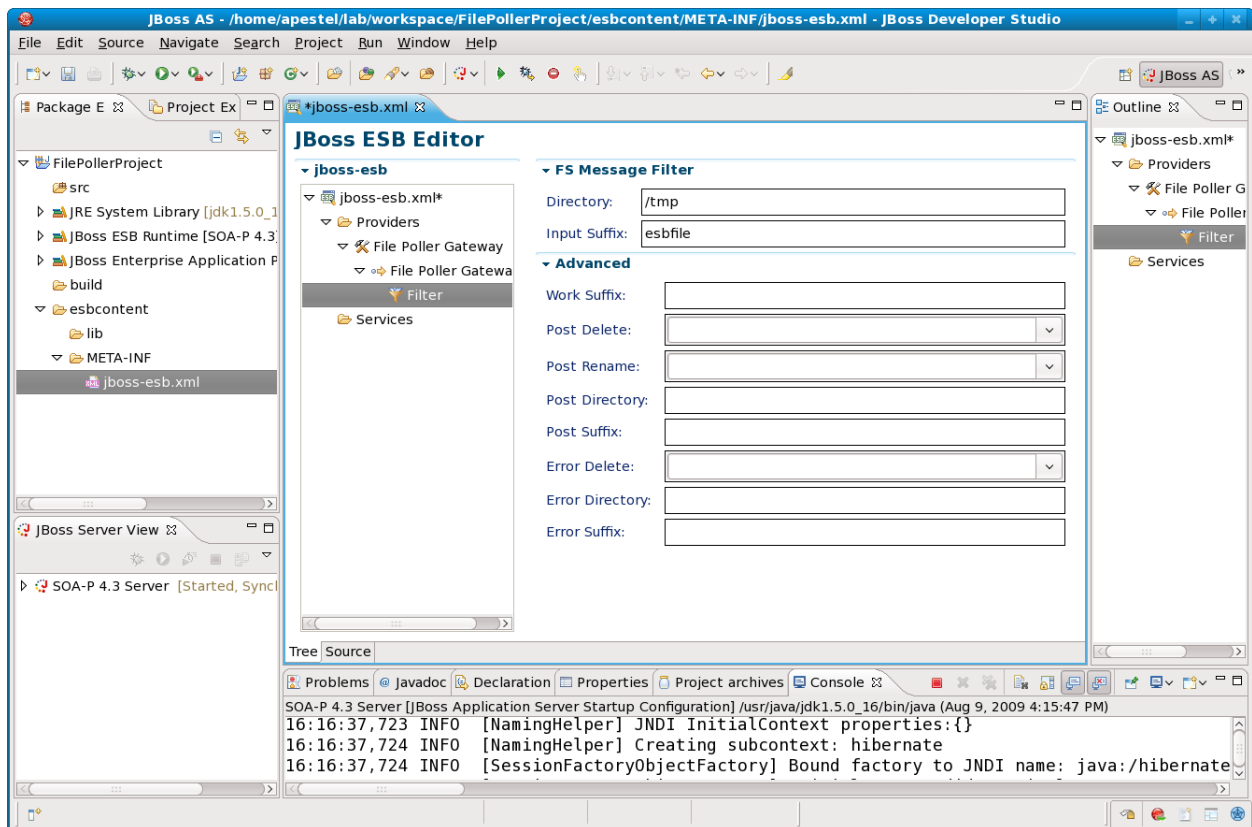
Now we need to create a gateway “provider”. This is how we get messages onto the ESB. There are providers for HTTP, JMS, FTP, File, Email, JCA, Database, and a host of others – including the ability to create your own custom provider. We are going to create a fs-provider (File System) gateway. So, right-click on the “Providers” folder in the JBoss ESb Editor and select “New | FS Provider...”. You should see a dialog like this.



Give the provider a name “File Provider Gateway” as shown above and select “Next”.



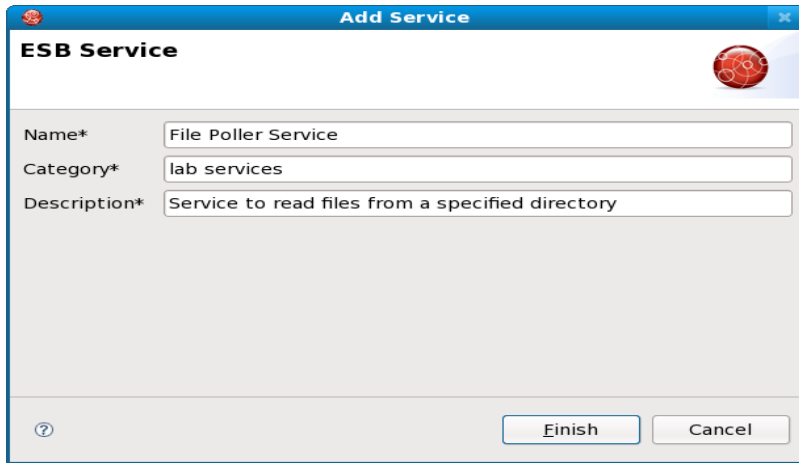
The Channel ID is the identifier that will be used later when a service declares that it wants to get messages from this gateway. I named the Channel ID the same as the provider name “File Poller Gateway”. Click “Finish”.



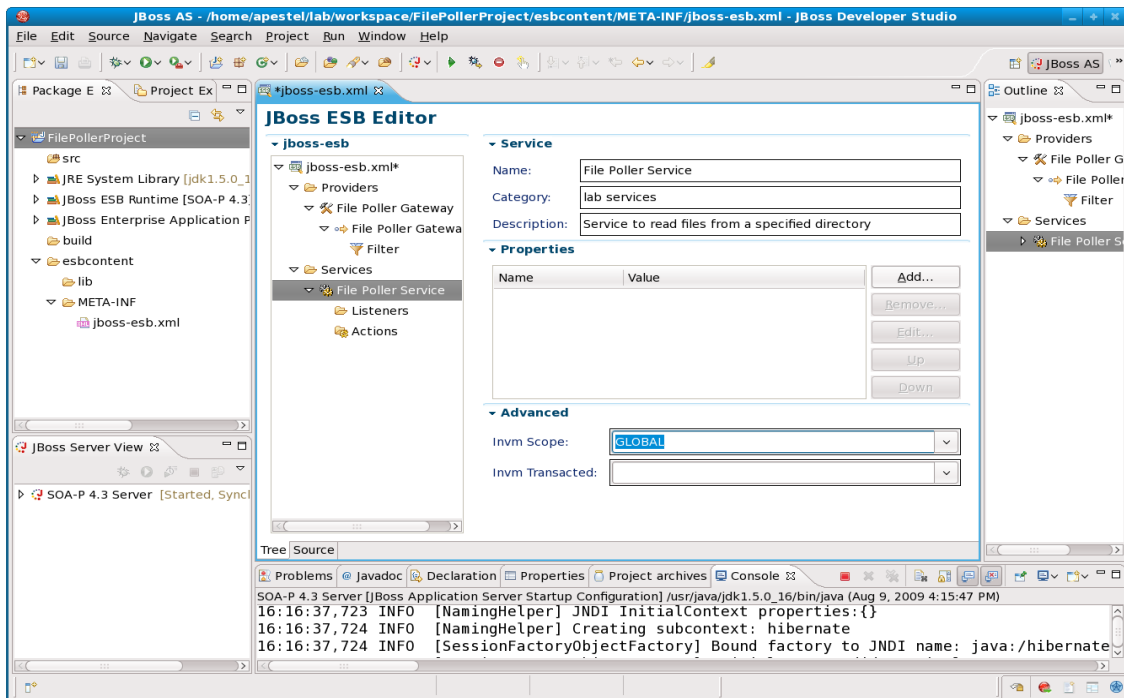
Now the file gateway provider has been added to our ESB configuration. However, we need to specify a filter so that the file gateway will know what files to look for. So, expand the File Poller Gateway tree as shown above and select the “Filter” node. This will allow us to configure a host of options. In this case, we only want to specify the directory

to be polled and the suffix of the files we will be looking for. I am polling “/tmp” and looking for files that have an “esbfile” suffix.

Now that we have a gateway provider, we need to create an ESB service to consume messages from our provider. Right-click on the “Services” folder of the ESB Editor and select “Add Service...”.



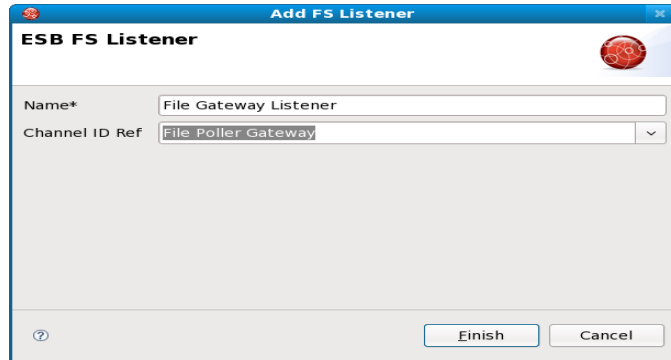
Enter the information as shown above and click “Finish”.



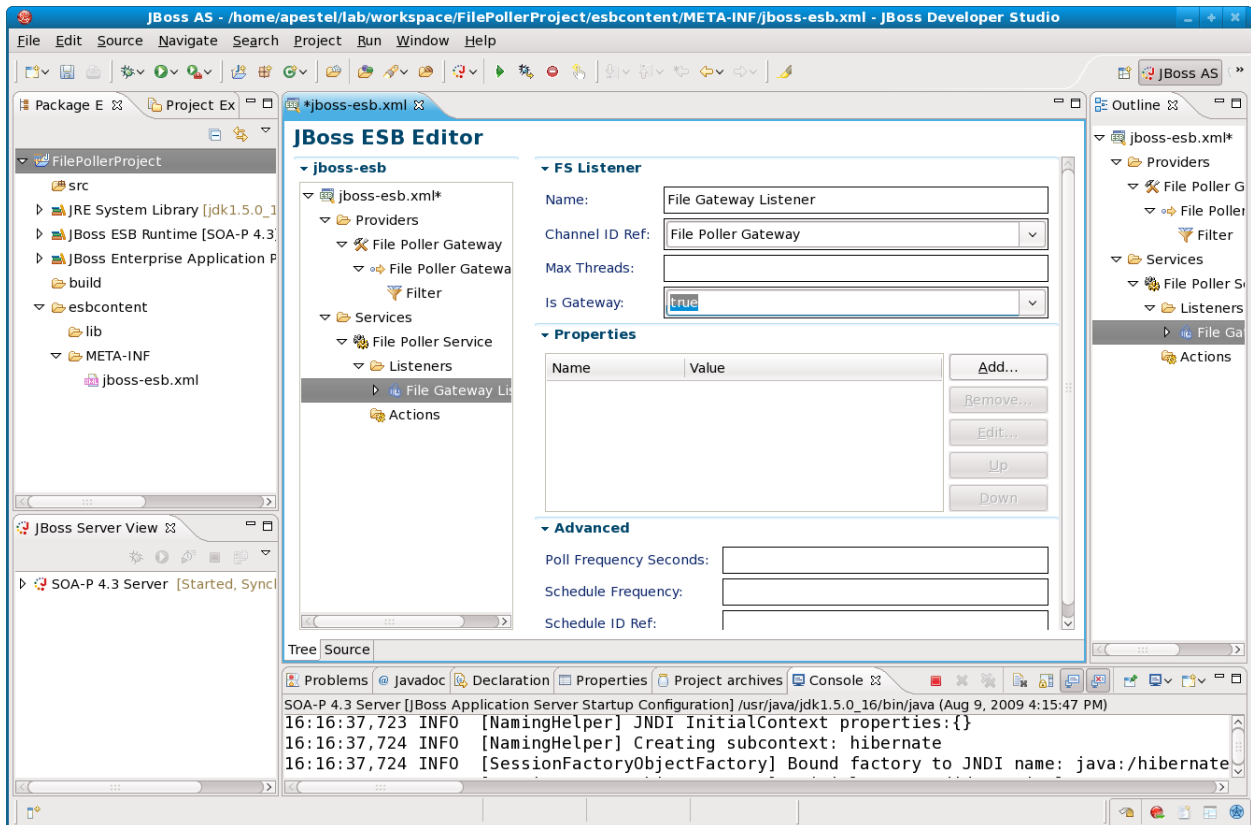
We should see the service selected as above. The only thing we need to do here is select “GLOBAL” for the InVM scope (highlighted in the above screenshot). Basically, the InVM scope is an optimization that allows the gateway provider we created to send messages directly to the service without have to go through an additional non-gateway provider.

We now have our provider setup and we have a service created. It's now time to tell our service that we want it to receive any messages that come into the provider we created. This might initially seem an unnecessary step. But, not all services have just one provider. For example, we might have a service that accepted a SOAP document from a file system, FTP, JMS, and HTTP. In this case, we would still only have to create one service but could link it to four different gateway providers.

To link this service to the gateway provider we created, right-click on the "listeners" folder under the service we created and choose "New | FS Listener".



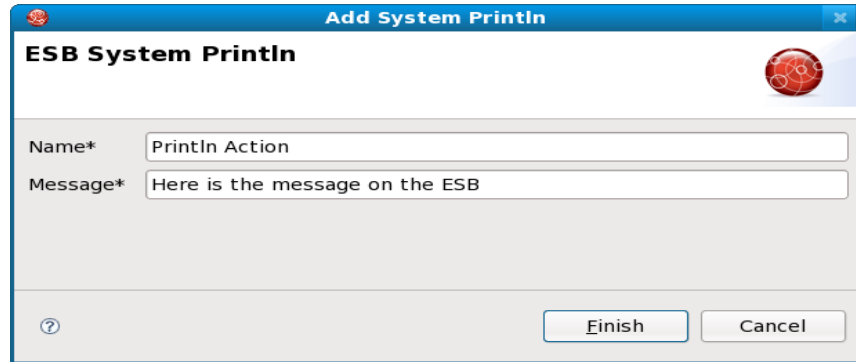
We need to name the listener and select the "File Poller Gateway" that we previously created by choosing it from the dropdown. Click "Finish".



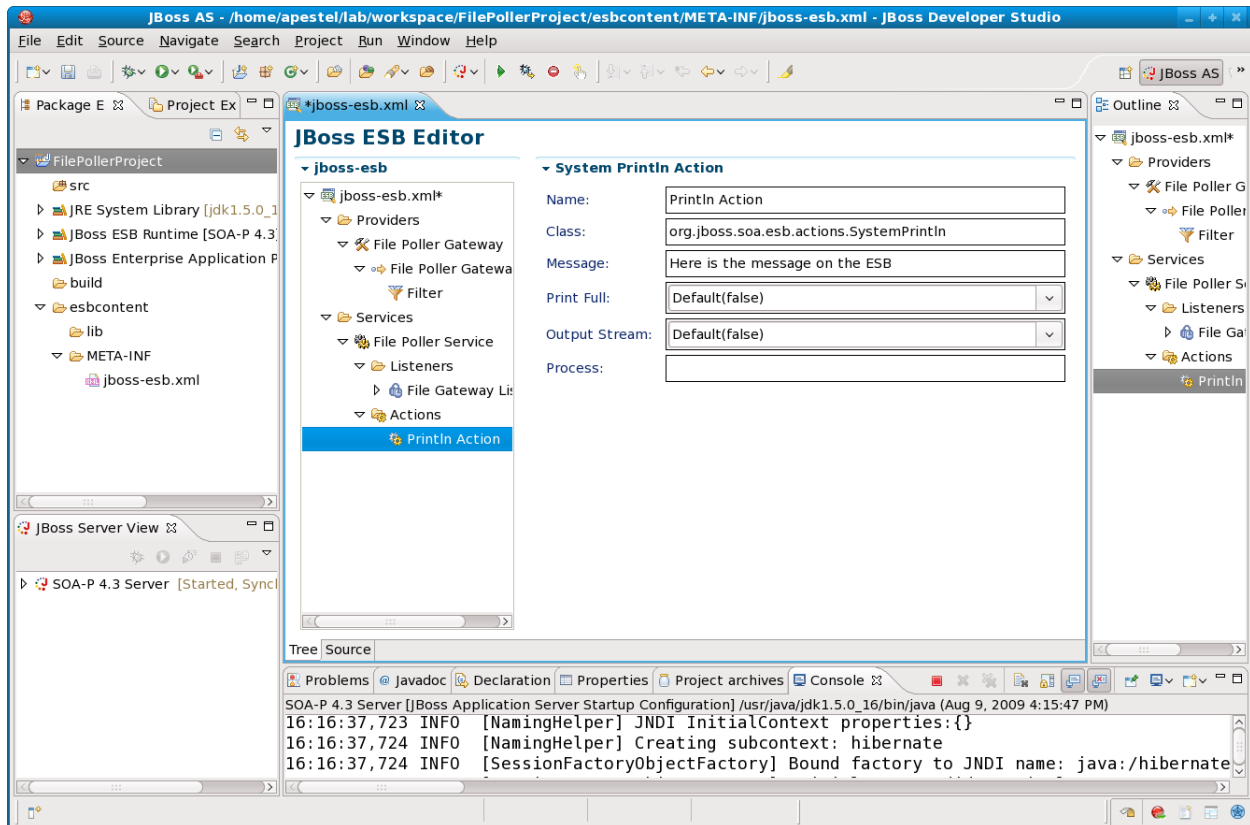
Now we have the service pointing to our gateway provider. Make sure that you set the "Is Gateway" to "true" as

highlighted above. This means that the listener is a “Gateway” and can therefore accept messages of any type (binary, ASCII, XML, etc.). If we did not specify that this was a gateway, it could only accept JBoss ESB specific messages.

The ESB service is actually created now and would work. But, since we don't have any actions specified on the service, we wouldn't see much when the service was invoked. So, we're going to add a “Print Ln” action to our service that will print the contents of whatever message we send to this service. To do this, right-click on the “Actions” folder under our service and select “New | System Println...”

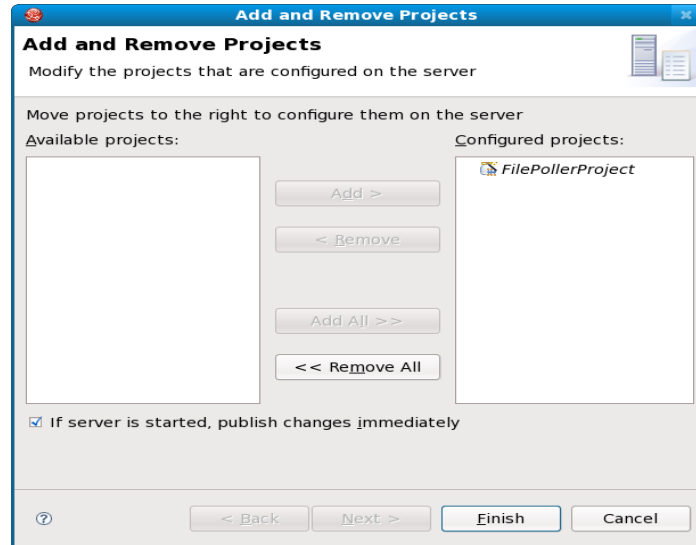


We need to give the action a unique name and specify the label we will print in front of the actual message that is on the ESB. Click “Finish”.

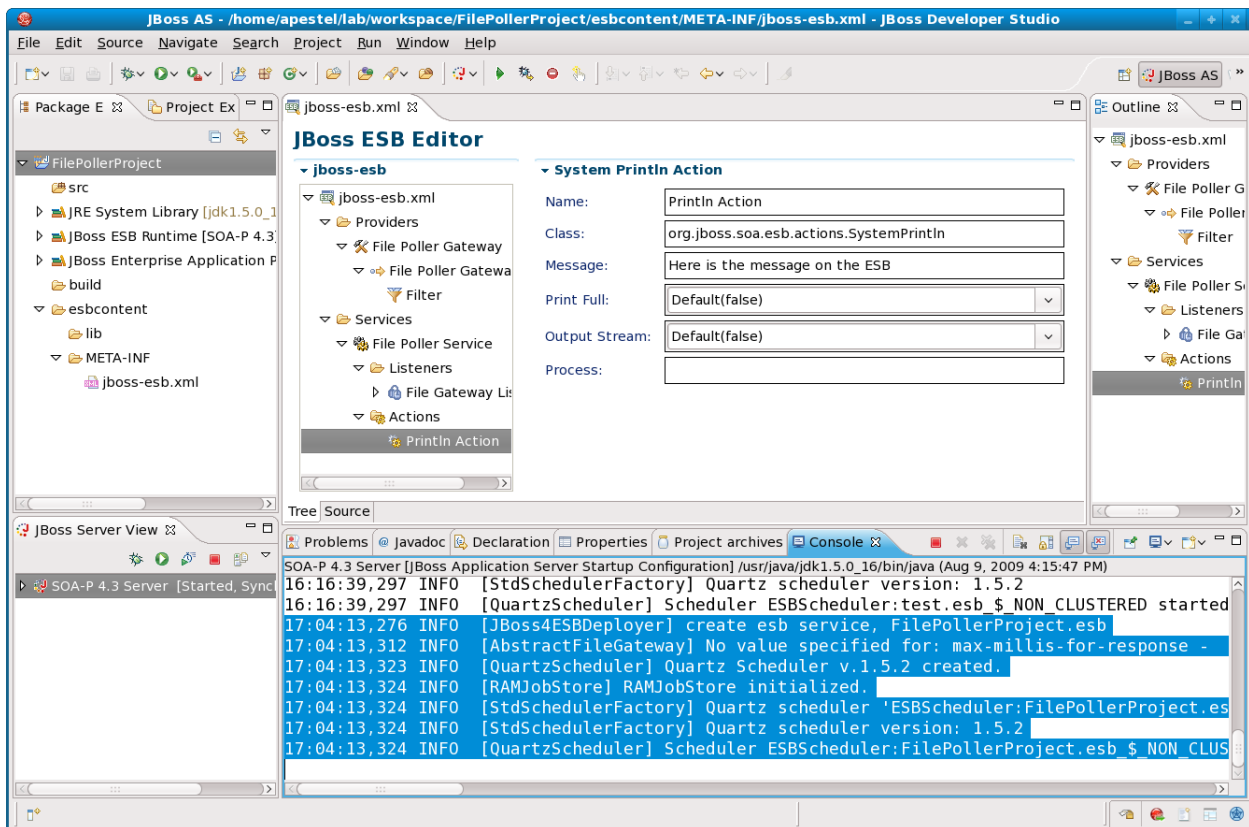


Now we are ready to deploy this ESB project to our server, just like we would any other JBDS (Eclipse) project. Let's save the project first: “File | Save”.

Now, right-click on our “SOA-P 4.3 Server in the lower left JBoss Server View and select “Add and Remove Projects...”.



Click the “Add All >>” button to move the FilePollerProject to the list of configured projects. Click “Finish”.



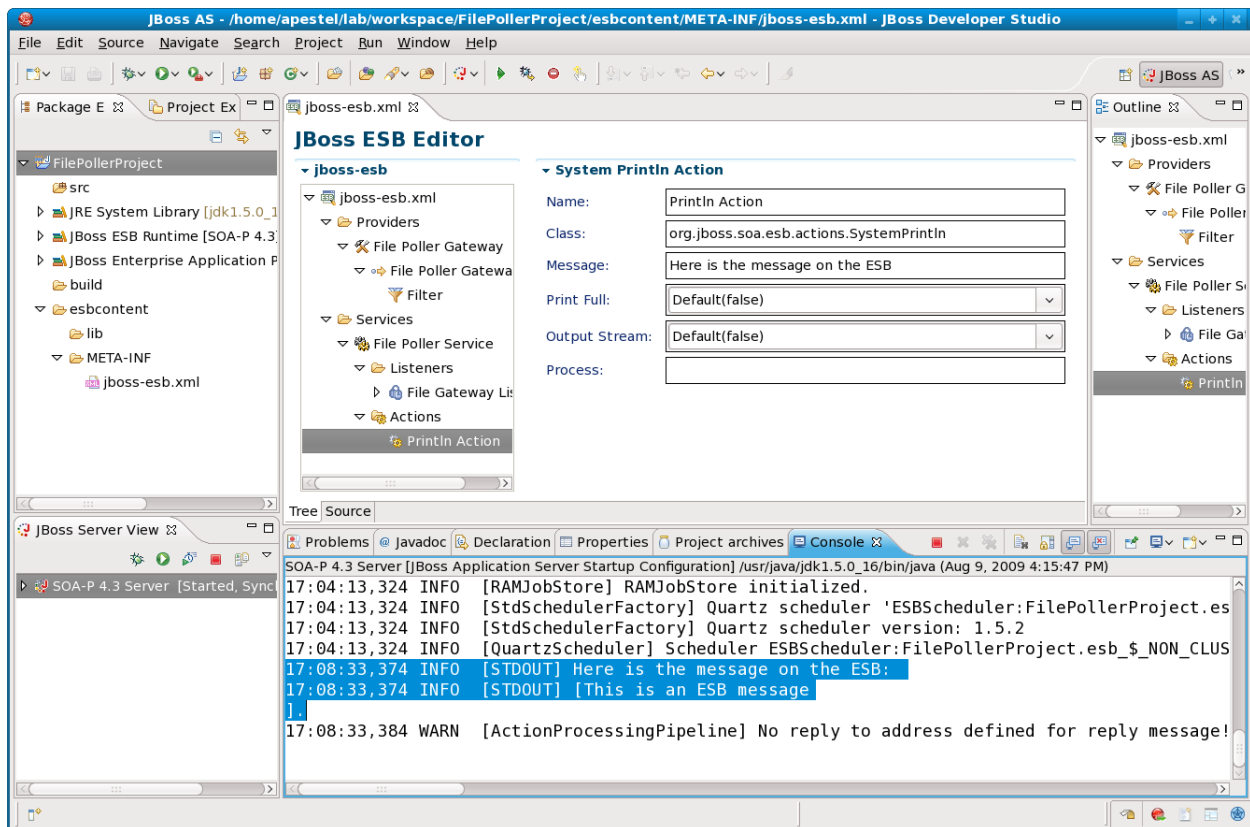
You'll see in the console that when we added our project to the server, JBDS created a “.esb” archive (similar to a .war archive) and deployed it to our SOA-P server. Now it's time to test it!

So, we need to create a file with the suffix we specified in the directory we specified. You can do this with any editor like VI, Notepad, Emacs, etc. I'm going to just "echo" some text to a file below like this:

```

apestel@localhost:~
File Edit View Terminal Tabs Help
[apestel@localhost ~]$ echo "This is an ESB message" > /tmp/filename.esbfile
[apestel@localhost ~]$
    
```

Now if we go back to JBDS and look in the console view, we should see that the ESB received our message.



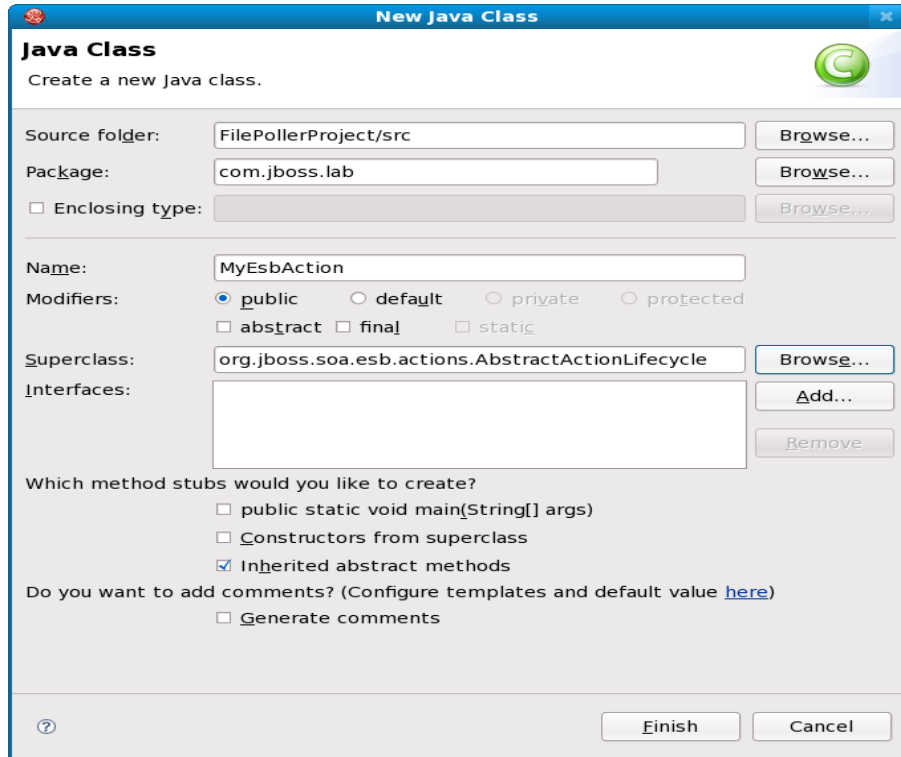
Sure enough, there is our message with the label we specified in our System PrintIn action!

If you do not see this message printed out in your console, please raise your hand.

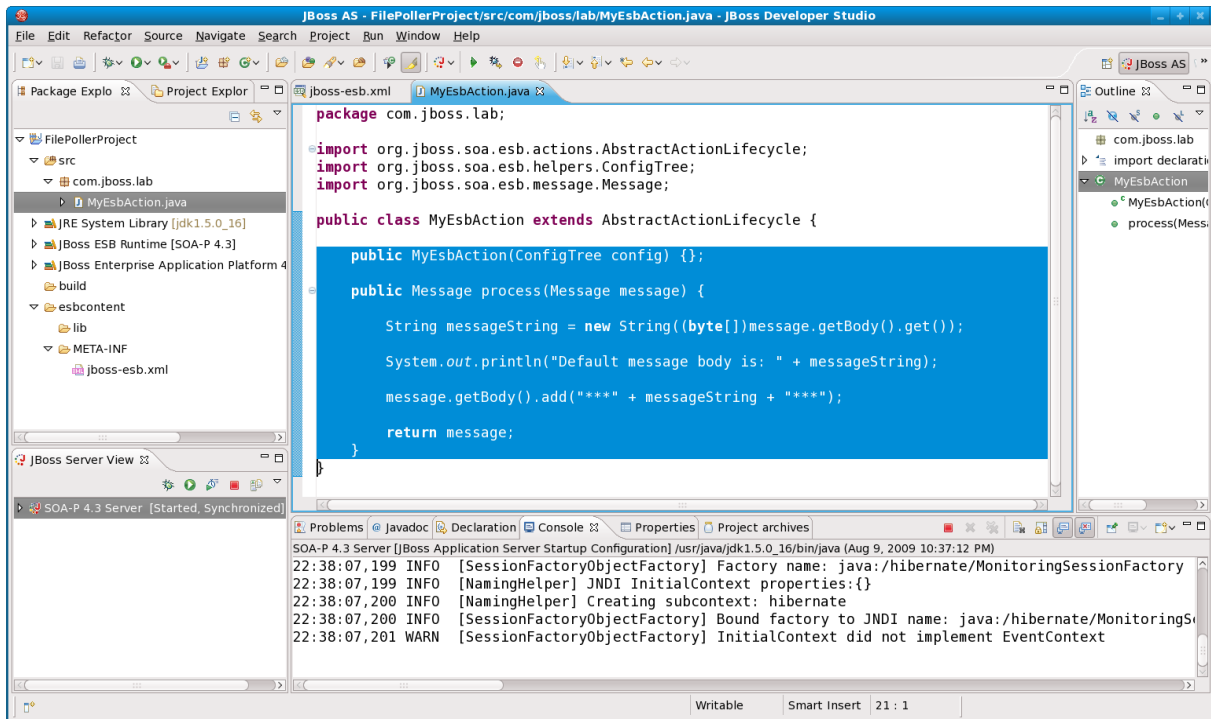
Lab #6: Adding a Custom ESB Action

ESB services have listeners and actions. Thus far, we have created a service with a file poller listener and a single System Println action. Now, we're going to add a custom Java action. An ESB action has full access to the ESB message. It can read the message, modify the message, forward to other services, call out to external systems, anything that can be done from a Java class.

To create a custom Java action, we first need to create the Java class. So, right-click on the "src" folder under the FilePollerProject at the left of the JBDS window and select "New | Class".



Here we've specified that the Package will be "com.jboss.lab", the Classname will be MyEsbAction, and most importantly, it will be a subclass of org.jboss.soa.esb.actions.AbstractActionLifecycle. Click "Finish".



By default, the class is created with no methods. We want to add a constructor and a “process” method that will be invoked when the ESB service gets to this action. We could name the method something other than “process” and then specify the method name when we reference the action in the ESB’s configuration. But by default, the ESB will look for a “process” method that accepts a “Message” parameter and returns a “Message” parameter.

Here is the full listing of the class:

```

-----
package com.jboss.lab;

import org.jboss.soa.esb.actions.AbstractActionLifecycle;
import org.jboss.soa.esb.helpers.ConfigTree;
import org.jboss.soa.esb.message.Message;

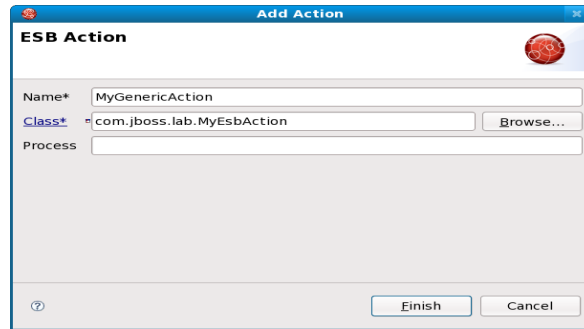
public class MyEsbAction extends AbstractActionLifecycle {
    public MyEsbAction(ConfigTree config) {};
    public Message process(Message message) {
        String messageString = new String((byte[])message.getBody().get());
        System.out.println("Default message body is: " + messageString);
        message.getBody().add("****" + messageString + "****");
        return message;
    }
}
-----

```

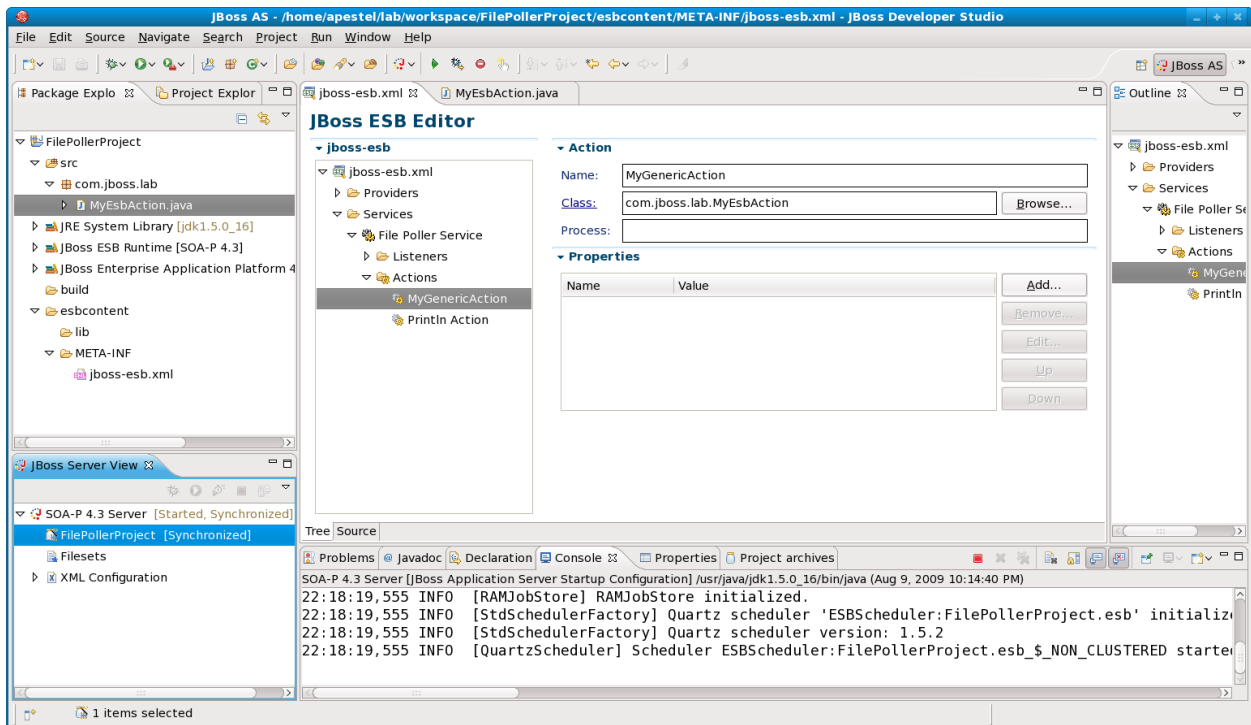
Note that this action is printing out the current contents of the message and then modifying

the message. We are casting the default message body to a byte[] and storing in a String so that it will print correctly. Without that, it would just print the address of the byte array (default toString() implementation).

Now that we've created the custom action, we need to reference it from the ESB service configuration. In JBDS, click back on the jboss-esb.xml editor. Right-click on the "Actions" folder under our ESB service and select "New | Generic Action".



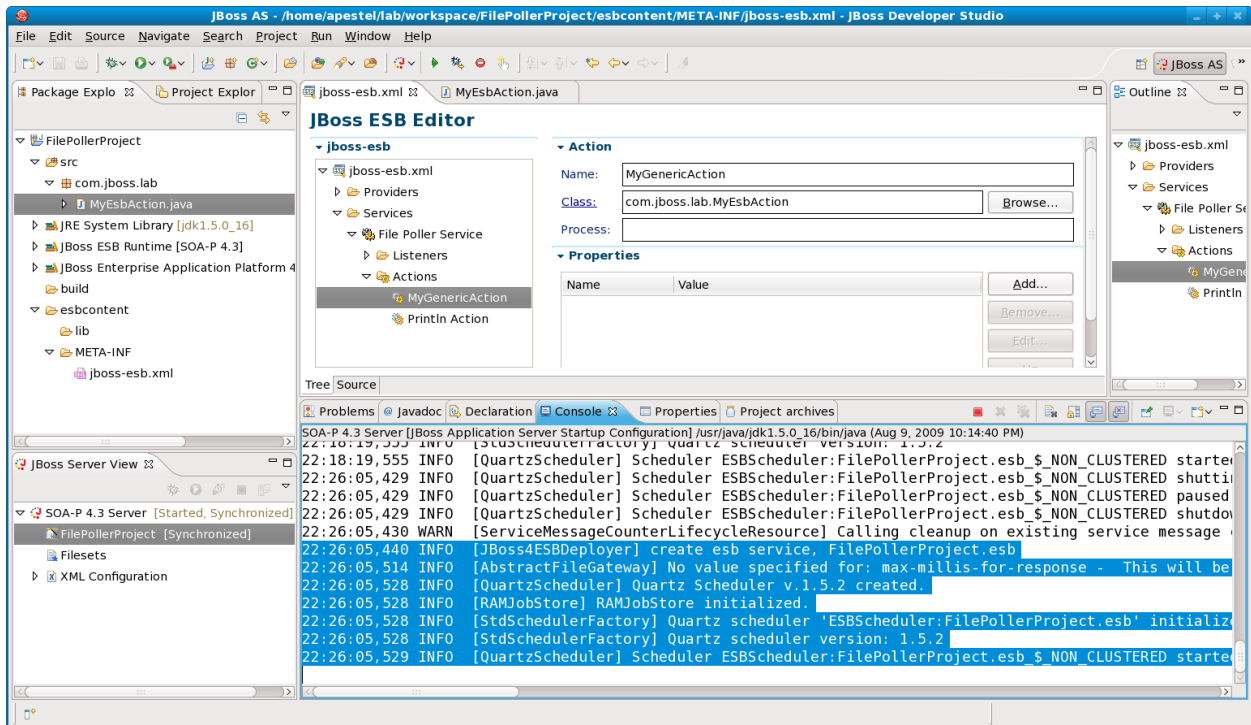
The Process field is where we would specify a different method name if we used something other than "process" in our Java class. Click "Finish".



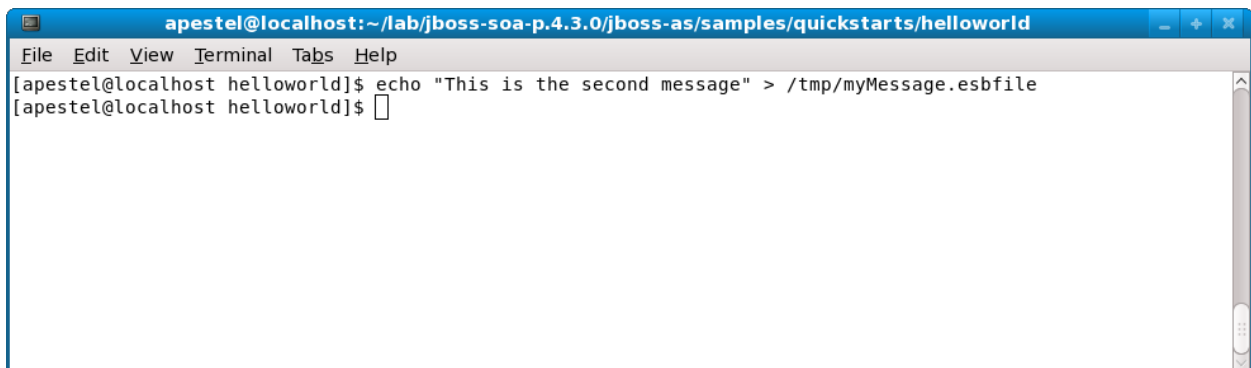
By default, our new action will be added below the PrintIn Action. So, click on the new custom action and drag it above the PrintIn Action so that it will be executed before the PrintIn Action. In this way, we should be able to see if our custom action really modified the ESB message.

Now we just need to save the project "File | Save All" and redeploy. To redeploy, we need to expand the "SOA-P 4.3 Server" in the JBoss Server View (lower left panel of JBDS). Inside there, we right-click

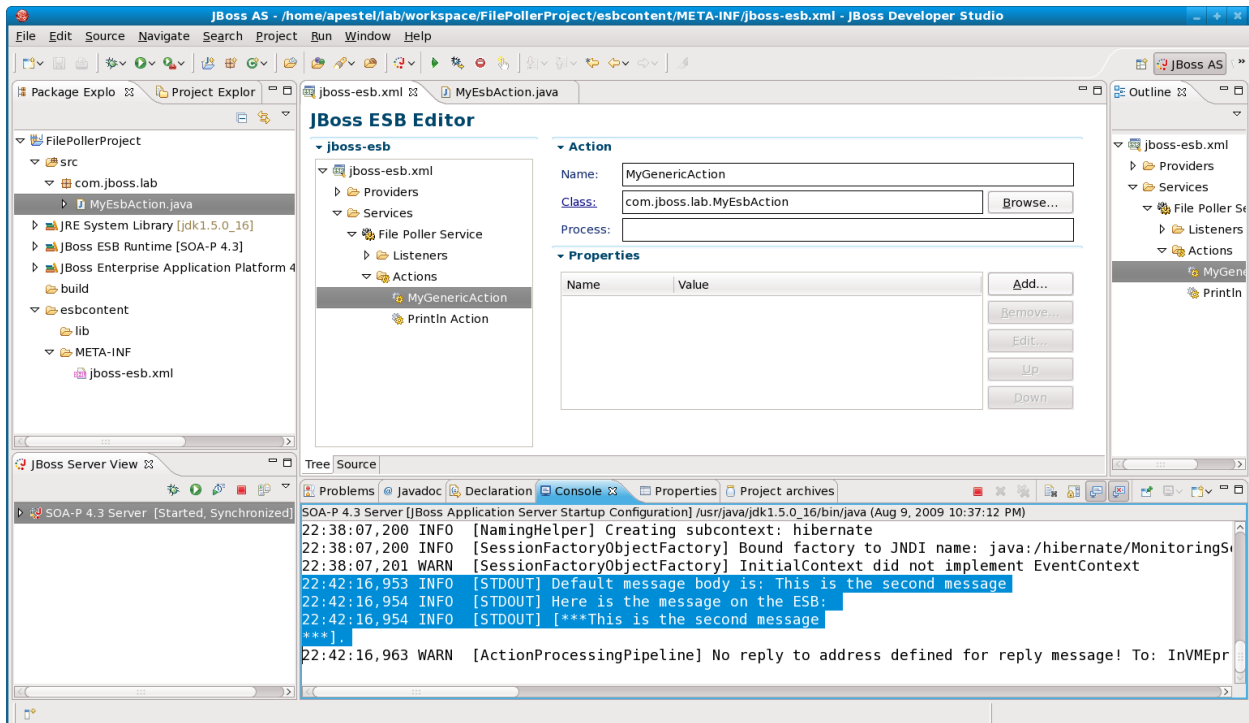
on FilePollerProject and select “Full Publish”. This will ensure that changes to Java source files get build and re-deployed to the server.



The console should show that the FilePollerProject was redeployed, as shown above. Now we just need to create another message (placing file in poller directory) to kick off the ESB service.



Now we need to take a look at the console panel in JBDS to see if the message made it through our two actions in our ESB service.



Sure enough, we see the "Default message body is" message from our custom action. Then, we see that our second action (System Println) printed out a message that is surrounded by "***" - which was done by our custom action.

Congratulations, you've now created a custom action and added it to your ESB service! You could modify this action to do anything that Java can do – which gives you a lot of flexibility.

Lab #7: Installing SoapUI For WS Testing

SoapUI is a wonderful tool for working with web services. In the simplest case, it provides a GUI to parse a WSDL, generate an editable test client, and invoke the web service. In more complex cases, SoapUI can be used to create elaborate and multi-threaded load test scenarios. For this lab we will only be using it for the most simple case.

First, we need to install SoapUI. Please find the correct zip for your platform (Linux, Mac, or Windows) and extract it as shown below.

```

apestel@localhost:~/lab
File Edit View Terminal Tabs Help
[apestel@localhost lab]$ cd /home/apestel/lab
[apestel@localhost lab]$ jar xvf soapui-3.0.1-linux-bin.zip
created: soapui-3.0.1/
created: soapui-3.0.1/bin/
created: soapui-3.0.1/bin/ext/
created: soapui-3.0.1/lib/
created: soapui-3.0.1/licenses/
created: soapui-3.0.1/Tutorials/
created: soapui-3.0.1/Tutorials/restexample/
created: soapui-3.0.1/Tutorials/restexample/NewsSearchService/
created: soapui-3.0.1/Tutorials/restexample/NewsSearchService/V1/
created: soapui-3.0.1/Tutorials/restexample/NewsSearchService/V1/search_files/
created: soapui-3.0.1/Tutorials/WSDL-WADL/

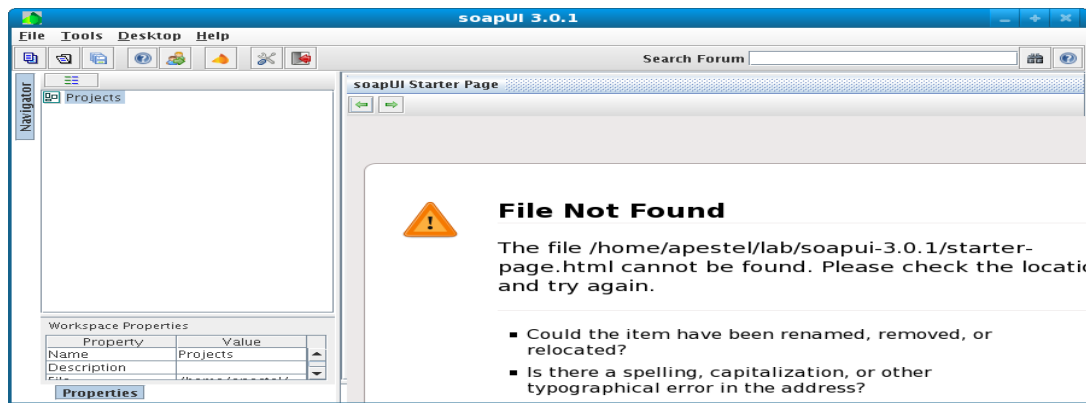
```

We should be able to start SoapUI by running the start script as shown below. On non Windows platforms, you may need to change the permissions on the start script so that it is executable.

```

apestel@localhost:~/lab
File Edit View Terminal Tabs Help
[apestel@localhost lab]$ chmod 777 soapui-3.0.1/bin/soapui.sh
[apestel@localhost lab]$ soapui-3.0.1/bin/soapui.sh
=====
= SOAPUI_HOME = /home/apestel/lab/soapui-3.0.1
=====
Locking assertion failure. Backtrace:
#0 /usr/lib/libxcb-xlib.so.0 [0x864767]
#1 /usr/lib/libxcb-xlib.so.0(xcb_xlib_unlock+0x31) [0x864831]
#2 /usr/lib/libX11.so.6(_XReply+0x244) [0x694bf64]
#3 /usr/java/jdk1.5.0_16/jre/lib/i386/xawt/libmawt.so [0x907f8dce]
#4 /usr/java/jdk1.5.0_16/jre/lib/i386/xawt/libmawt.so [0x907e2d77]
#5 /usr/java/jdk1.5.0_16/jre/lib/i386/xawt/libmawt.so [0x907e2ef3]
#6 /usr/java/jdk1.5.0_16/jre/lib/i386/xawt/libmawt.so(Java_sun_awt_X11GraphicsEnvironment_initDisplay+0x26)
[0x907e3136]
#7 [0xb16a0908]
#8 [0xb1699b6b]
#9 [0xb1699b6b]
#10 [0xb1697236]
#11 /usr/java/jdk1.5.0_16/jre/lib/i386/server/libjvm.so [0xb78d8eec]
#12 /usr/java/jdk1.5.0_16/jre/lib/i386/server/libjvm.so [0xb7aa8ae8]

```



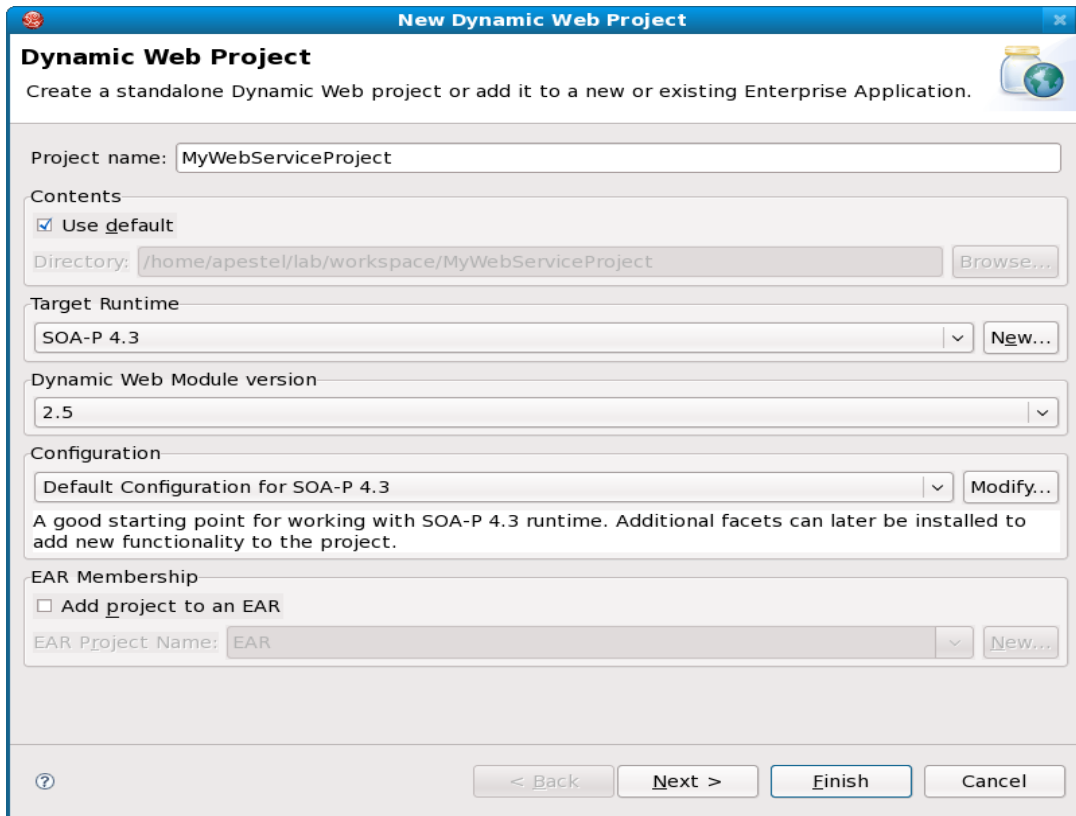
Don't worry about the file not found error if you are not online. As long as SoapUI has started as above, we're good to go. Please close the Starter Page panel though as other panels sometimes seem to get hidden behind it.

Lab #8: Create a JSR 181 Web Service

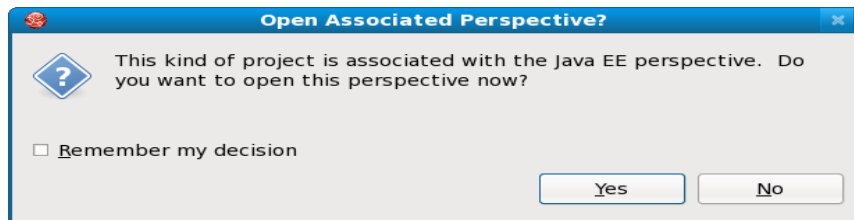
One of the most common use cases for an ESB is to mediate web services. The client sends a request to the ESB, the ESB may do specialized security, transformations, routing, or a host of other actions, and then the ESB routes to a backend web service implementation.

In this lab, we will create a simple JSR 181 web service and show invoking it from SoapUI. In the next lab, we'll proxy this web service with the ESB and show invoking the proxy via SoapUI.

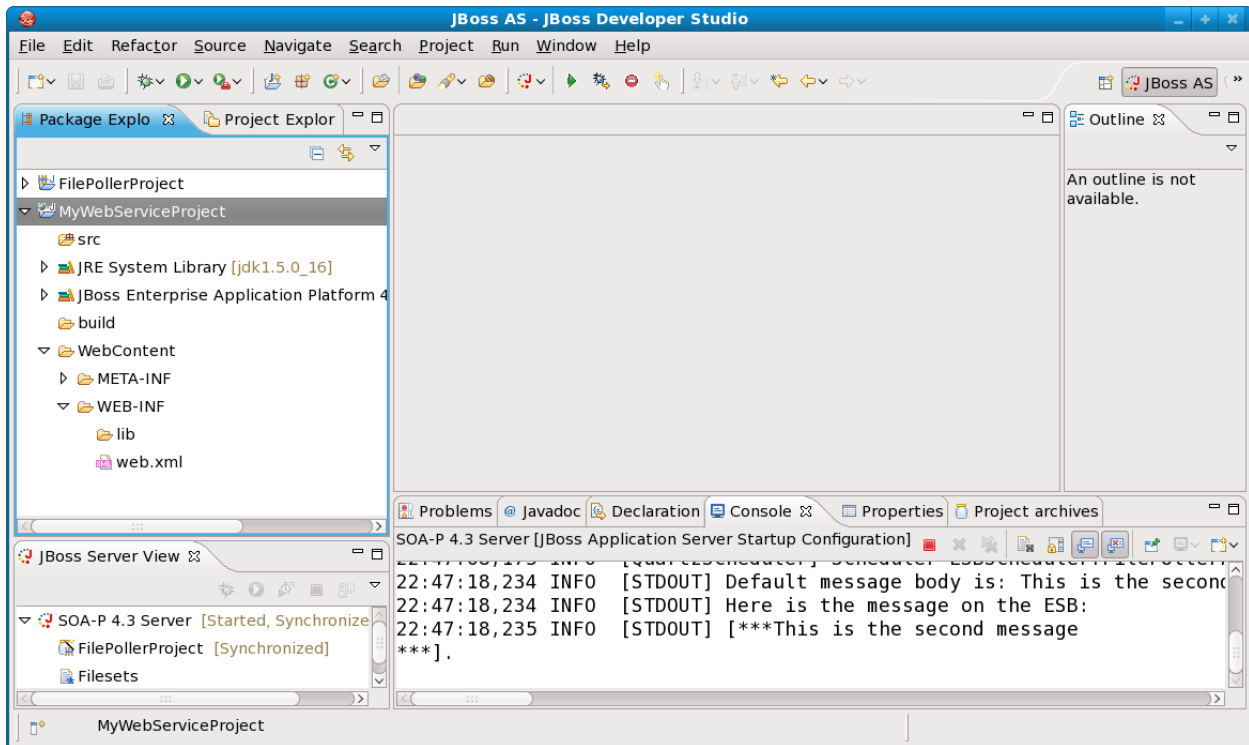
Creating a JSR 181 web service is quite easy with JBDS. First, we'll create a new Dynamic Web Project "File | New | Dynamic Web Project".



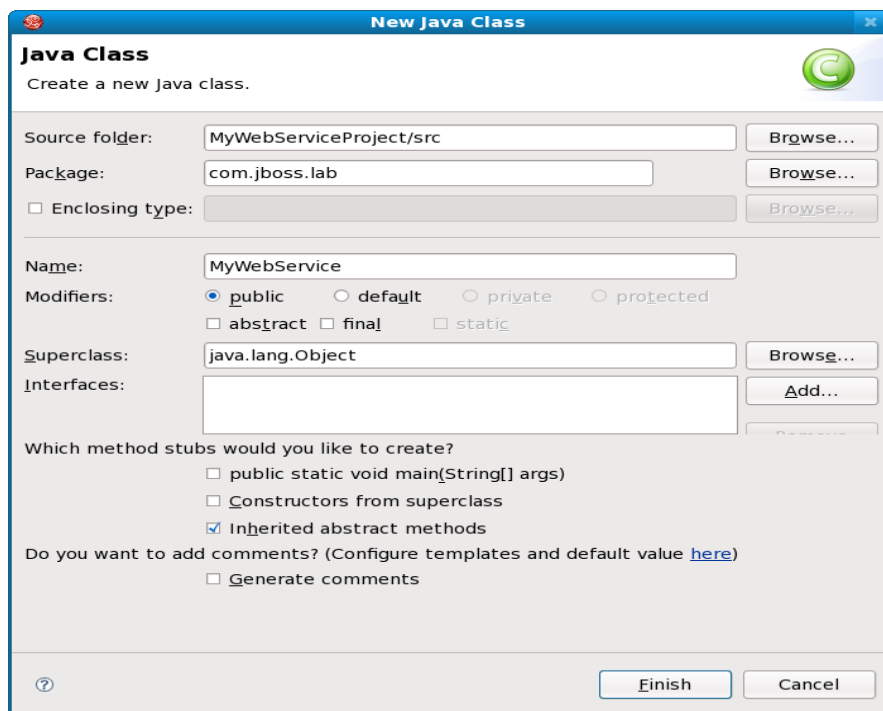
Name the project “MyWebServiceProject” and click “Finish”.



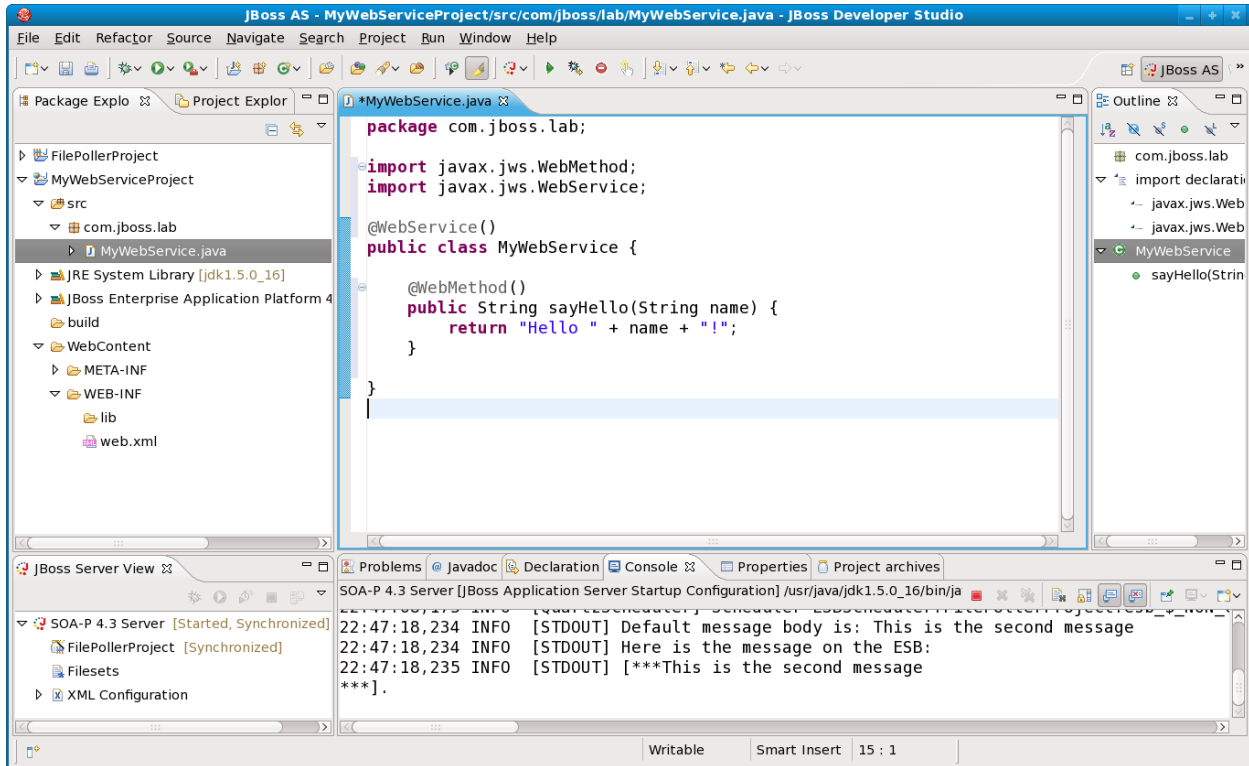
You can switch to the Java EE perspective if desired, but I prefer to stay in the JBoss AS perspective. So, click “No”.



Now we have our Web Service Project. But, we need to add our web service Java class. To do this, right-click on the “src” folder in our MyWebServiceProject and choose “New | Class”.



We need to give this class a package “com.jboss.lab” and a name “MyWebService” as shown above, then click “Finish”.



JSR 181 makes it pretty easy to create a web service – add a method with some annotations and you're done! Here is the listing for the class that is shown above.

```

package com.jboss.lab;

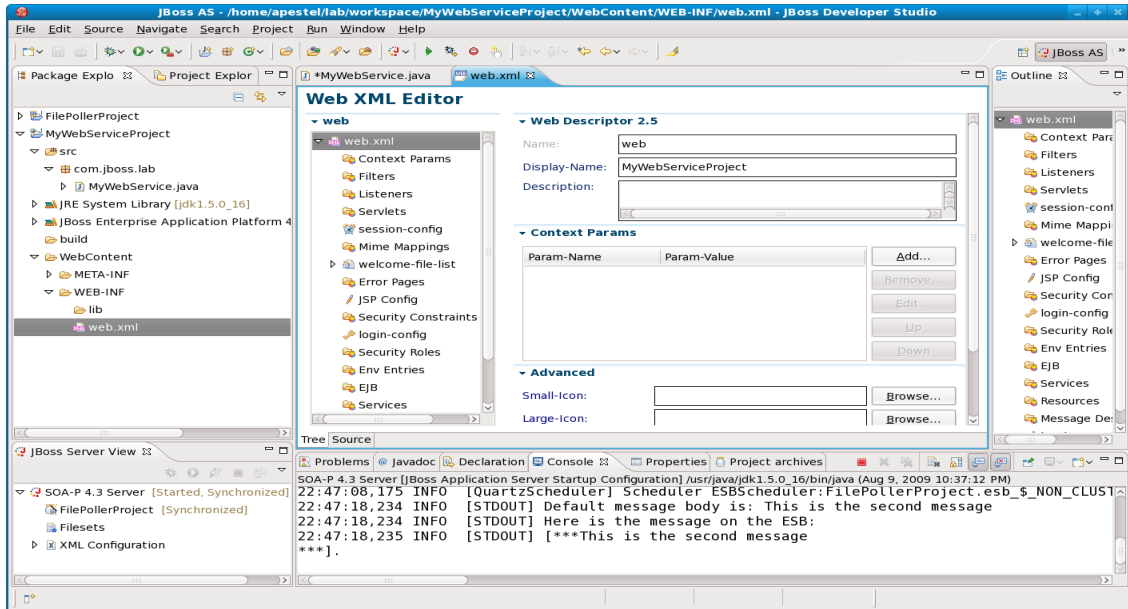
import javax.jws.WebMethod;
import javax.jws.WebService;

@WebService()
public class MyWebService {
    @WebMethod()
    public String sayHello(String name) {
        return "Hello " + name + "!";
    }
}

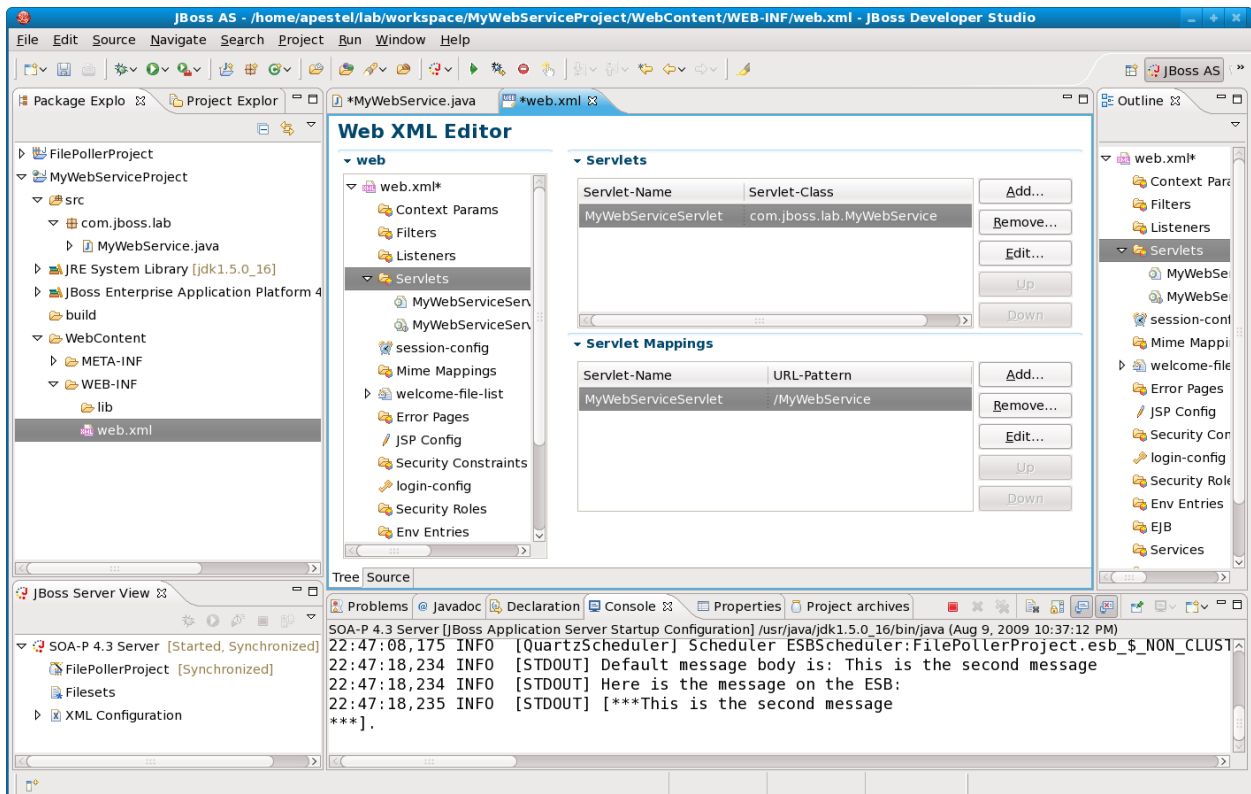
```

That's all the Java we have to write to create the web service. Now, we just need to list this as a servlet in our web.xml. To do this, go to the project explorer on the left side of JBDS and open the “Web Content / WEB-INF” folders and double click on web.xml.

JBoss SOA Platform with JBoss Developer Studio Workshop

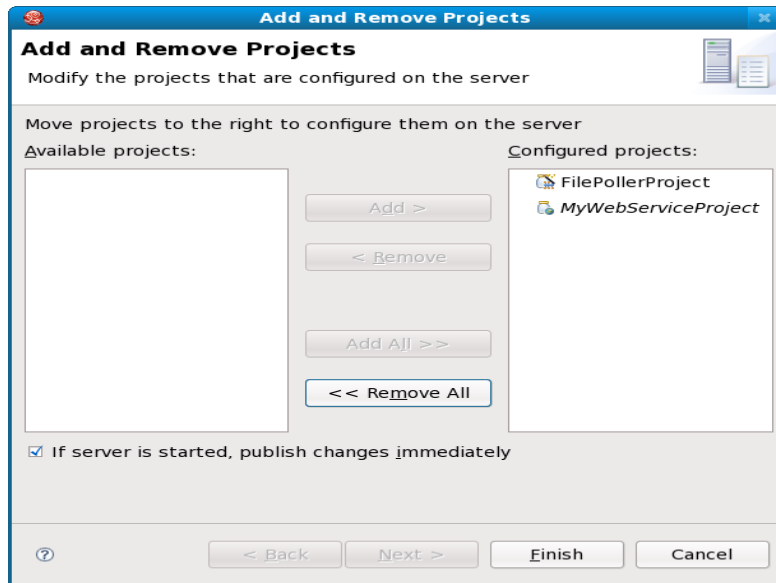


We need to add a servlet, so click the “Servlets” folder in the Web XML Editor and add information for our servlet as shown below. You'll have to click the “Add” button to add the servlet and “Add” button to add the servlet mapping.



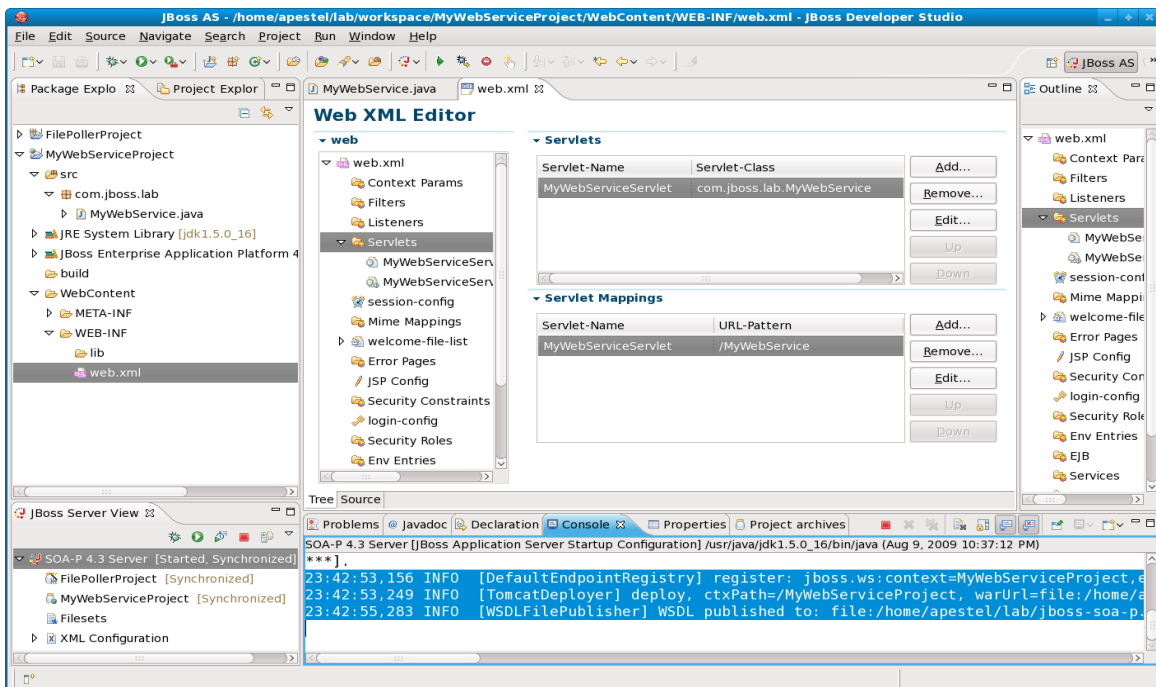
Note that we created a servlet with Servlet-Name “MyWebServiceServlet” and Servlet-Class “com.jboss.lab.MyWebService”. We also created a Servlet Mapping with Servlet-Name “MyWebServiceServlet” and URL-Pattern “/MyWebService”.

Now it is time to deploy this web service. So, save the project “File | Save All”. Then, right-click on the “SOA-P 4.3 Server” in the JBoss Server View (lower left corner of JBDS) and select “Add and Remove Projects...”.



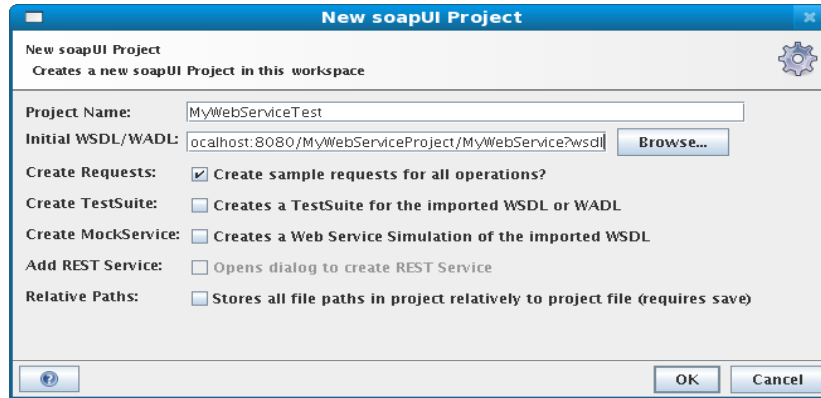
As shown above, we'll want to add the MyWebServiceProject to our list of Configured projects. Click “Finish”.

By looking in the JBDS console view, we should see that our web service was deployed and a WSDL was created in the filesystem – highlighted below.

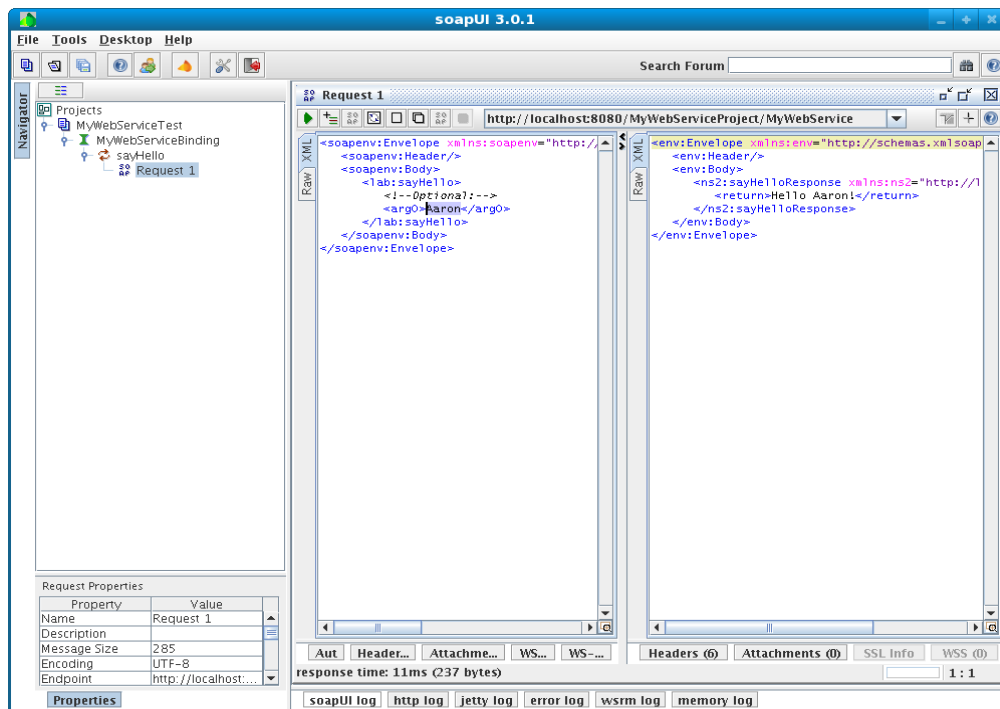


The WSDL is also available via a browser. By pointing your browser at <http://localhost:8080/jbossws/services>, logging in as “admin/admin”, and selecting “View a list of deployed services”, you can see all the web services deployed on the server (in our case there is just one).

The last thing we need to do is test this web service by invoking it via SoapUI. So, start SoapUI if not already started and choose “File | New soapUI project”.



For the WSDL location, we entered “<http://localhost:8080/MyWebServiceProject/MyWebService?wsdl>”. Click “OK”.



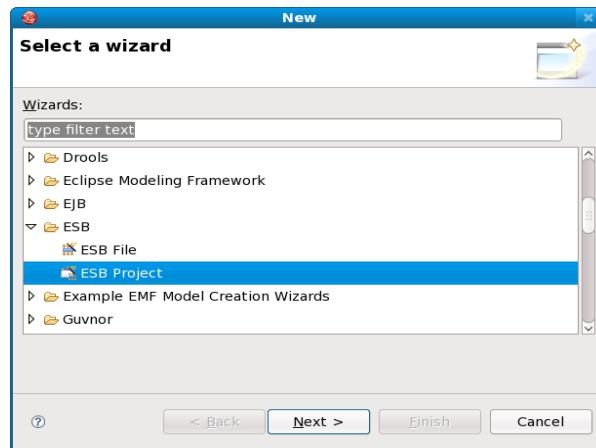
As shown above, navigate to “Request 1” in the tree on the left and double click on “Request 1”. This should open the Request 1 frame shown on the right. Enter your name instead of “?” for arg0 of the request and click the green triangle button above the request. You should see the “Hello <name>!” response in the panel on the right. Congratulations, you invoked a web service from SoapUI! Now, we need to proxy this service with the ESB.

Lab #9: Proxy Web Service with ESB

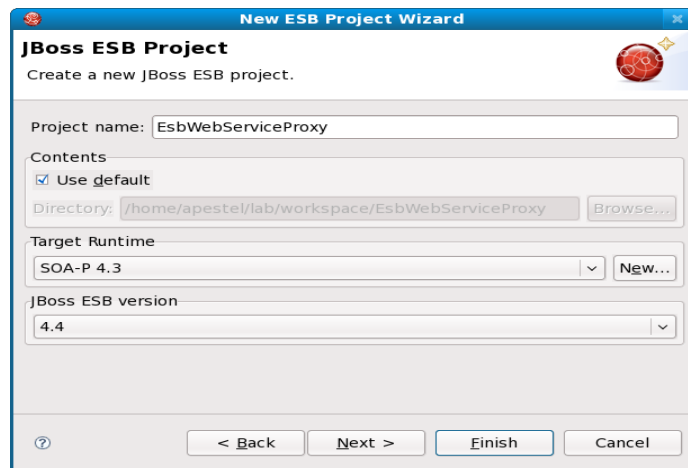
Now that we have our web service working, we want to create an ESB service that proxies this web service. In this way we can provide different security, versioning, transformation, routing, etc. for external clients coming in through the ESB than we do for internal clients accessing the web service directly.

At a high level, all we need to do here is create an HTTP gateway provider, create a service to use that provider, and add an HttpRouter action to that service to forward the web service request on to our JSR 181 web service implementation. In our case, we will also add a System Println action to our service just to see that the web service is passing through the ESB.

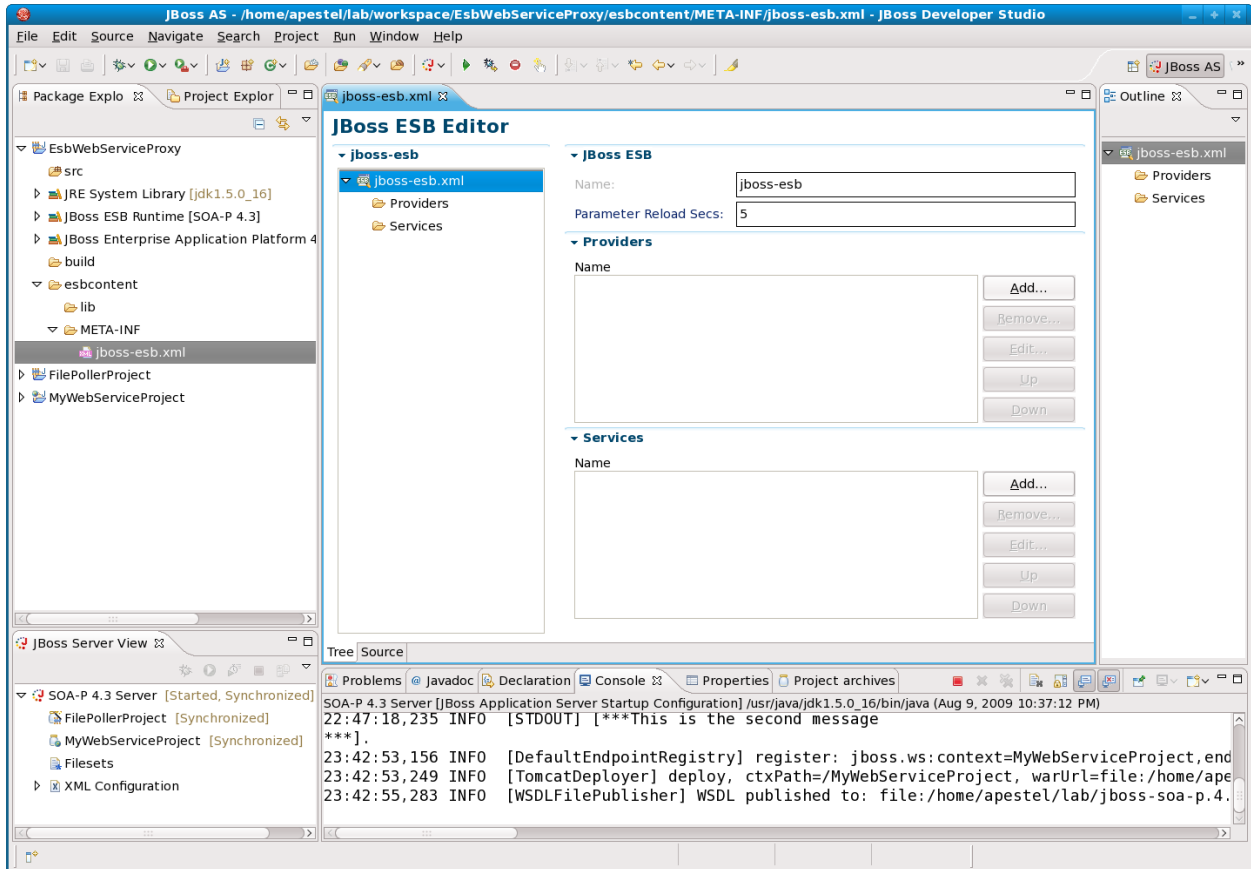
First, we are going to create a new ESB project. We could use the existing ESB project and just add an additional service to it, but I'd like to keep these labs as independent as possible. So in JBDS, select "File | New | Other...".



We'll select an "ESB Project" as shown above and click "Next".



Make sure to give the project a name, select the SOA-P 4.3 runtime, and select ESB version 4.4 as shown above. Click "Finish".

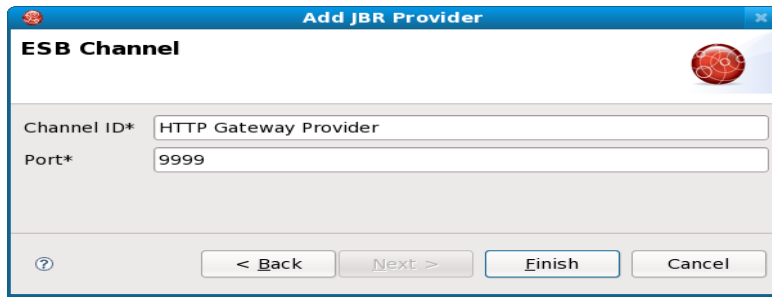


This should look familiar. We have our EsbWebServiceProxy in the explorer on the left and have our jboss-esb.xml file opened on the right with our JBoss ESb Editor.

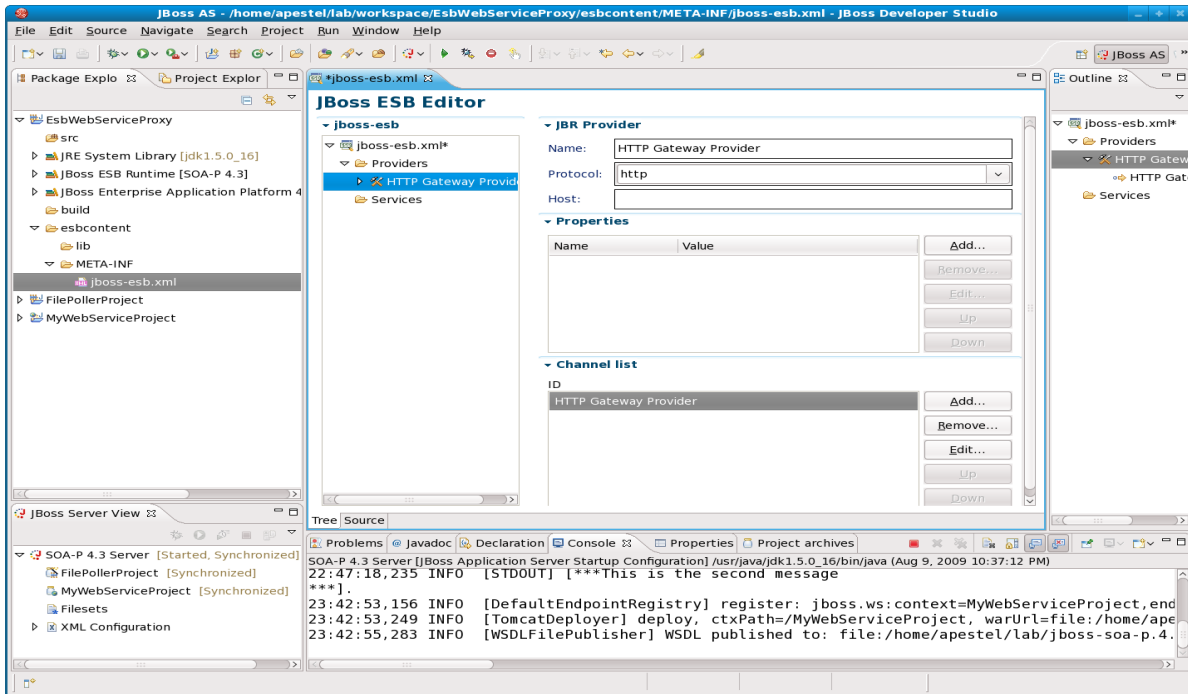
The first thing we need to do is create an HTTP provider gateway that the client will send web service requests to. Right-click on “Providers” in the JBoss ESb Editor and select “New | JBR Provider...”. JBR stands for JBoss Remoting and is one of two ways to create an HTTP gateway. EBWS (ESB Based Web Service) is the other way and will not be shown in this lab.



Provide a name as shown above and click “Next”.



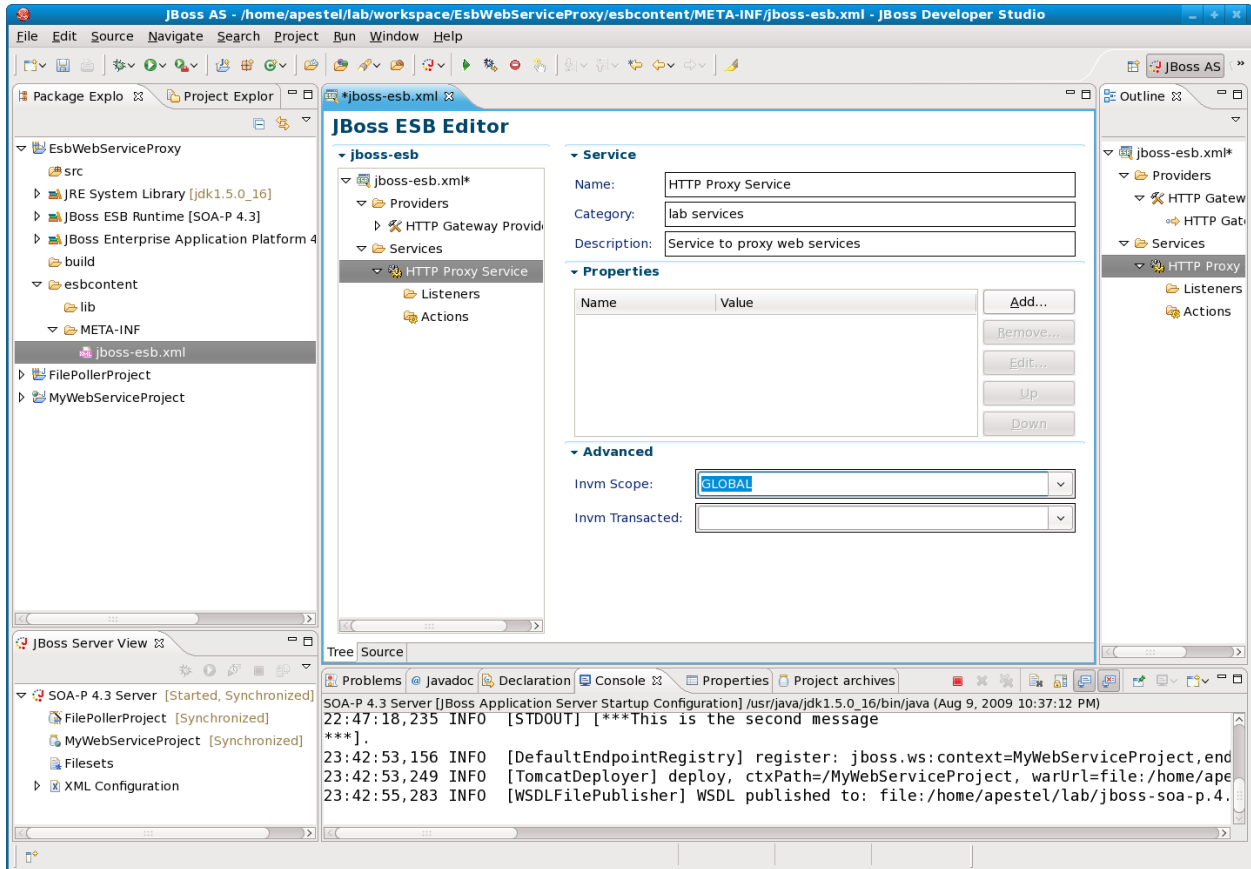
Provide the Channel ID (I use provider name again) and a port that this HTTP gateway will listen on. Click “Finish”.



Now we have our provider and need to create a service to use it. So, right-click on “Services” and select “Add Service...”.

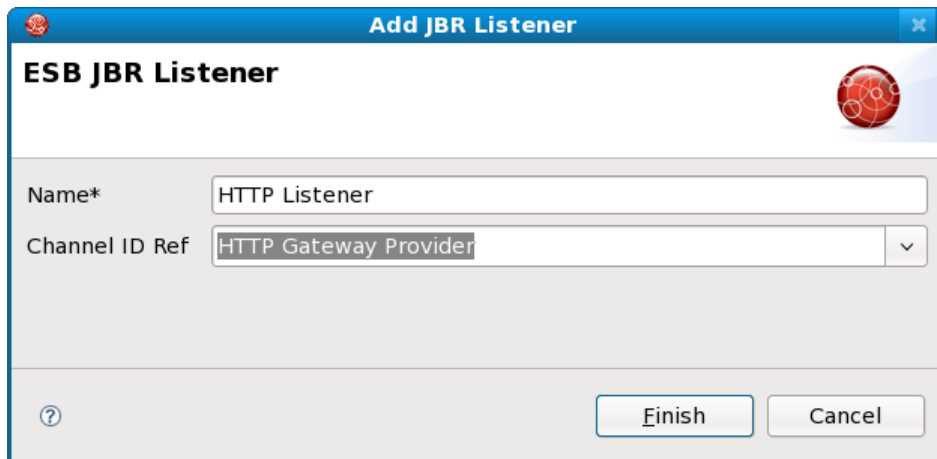


Enter information as shown above and click “Finish”.

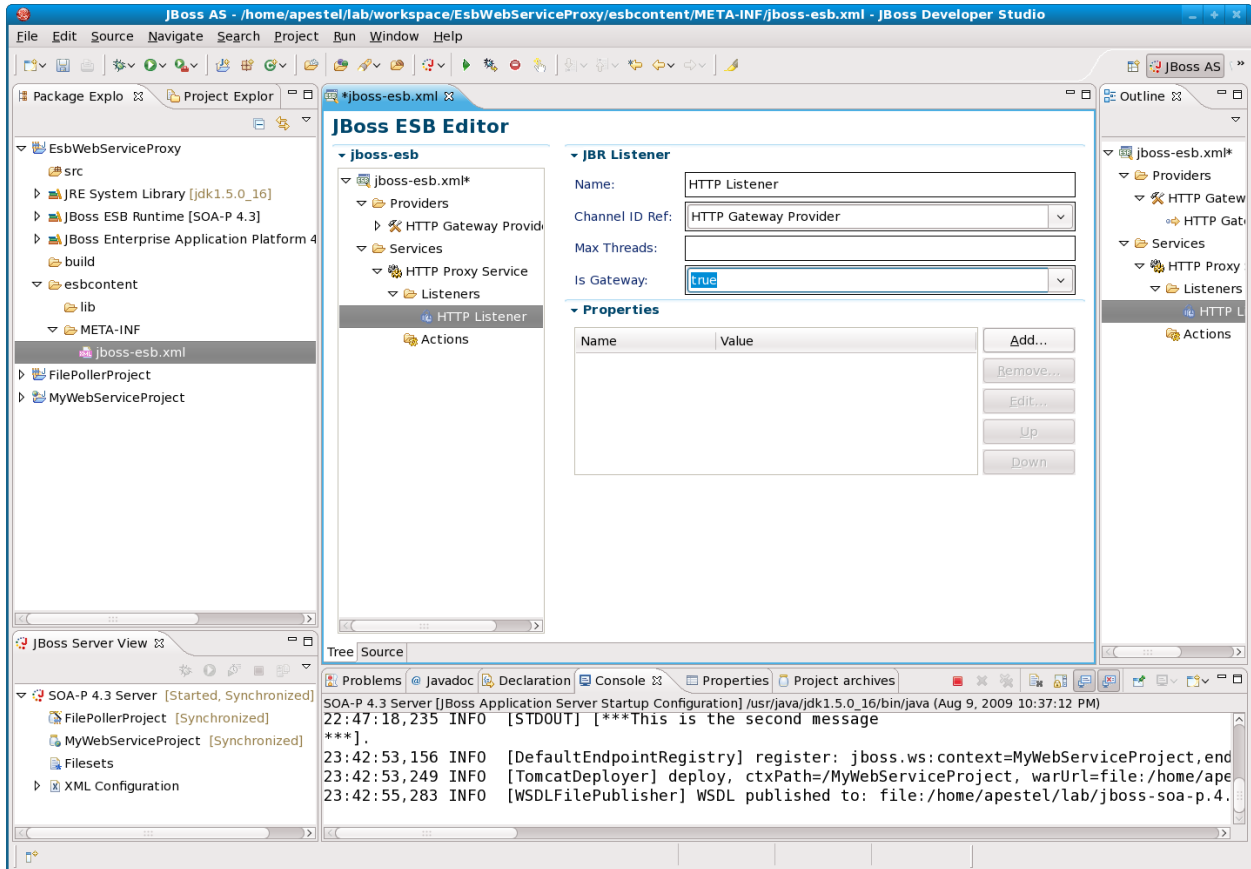


Now we have our service shown above. Make sure to set InVM Scope to “GLOBAL” as shown above or we would need to create another non-gateway provider in order to use this service.

We now need to add our HTTP Gateway Provider as a listener for this service. So, right-click on “Listeners” under our service and select “New | JBR Listener...”.



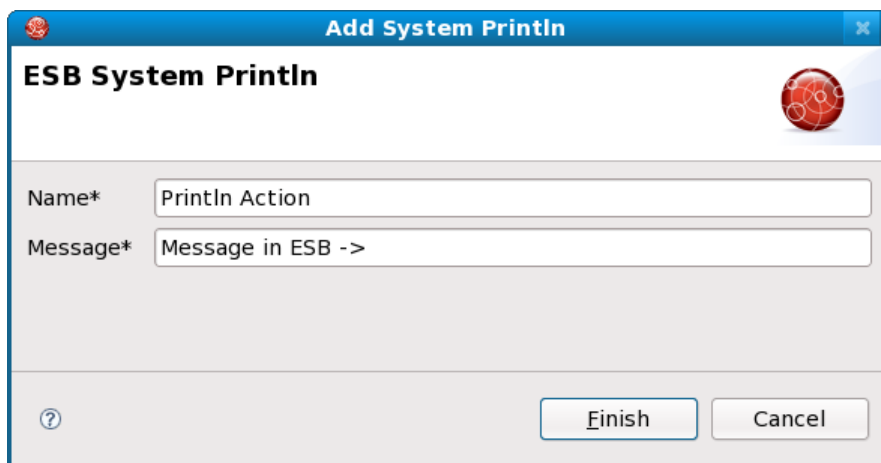
Name the listener and select our provider from the drop down box as shown above and click “Finish”.



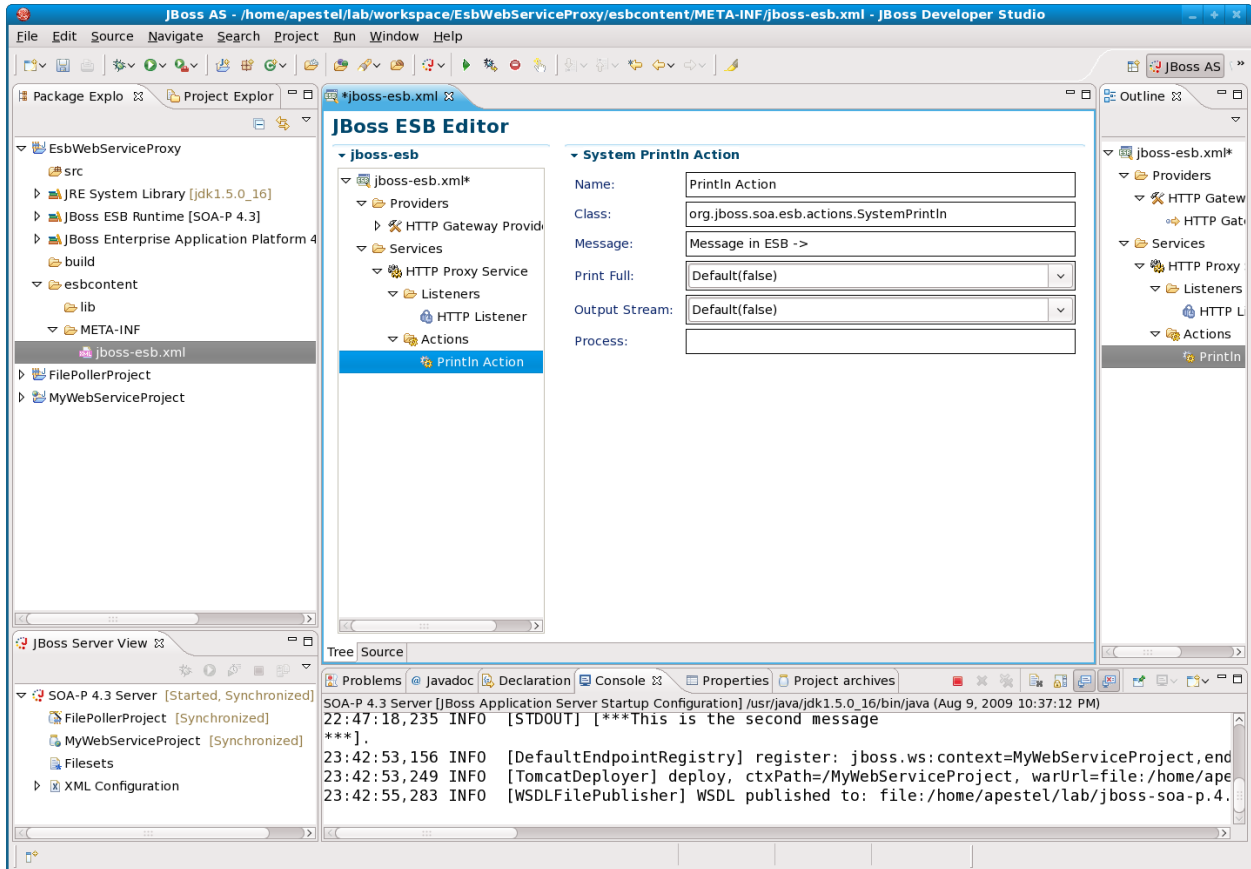
Now we have our listener setup, but make sure to set Is Gateway to “true” as shown above.

Next, we need to add some actions to print out the message (proving we're in the ESB) and to route to the JSR 181 web service we created earlier.

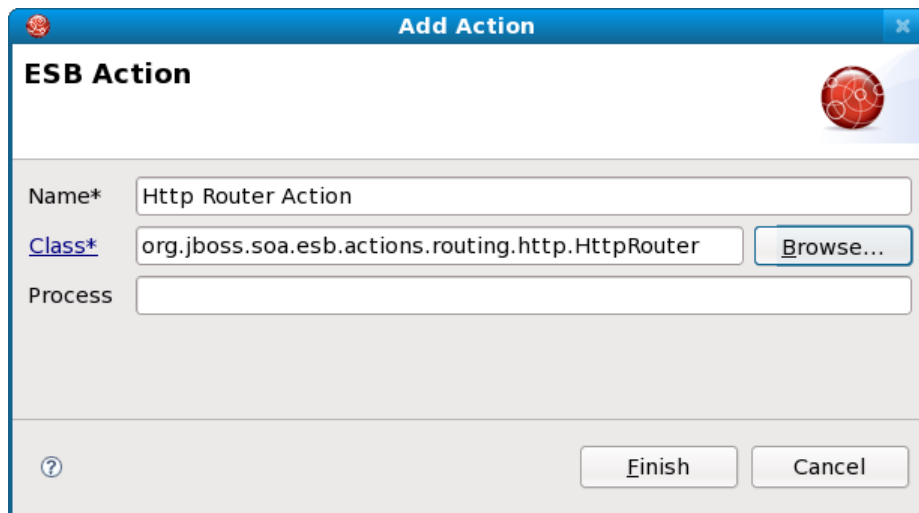
Right-click on “Actions” and select “New | System PrintIn...”.



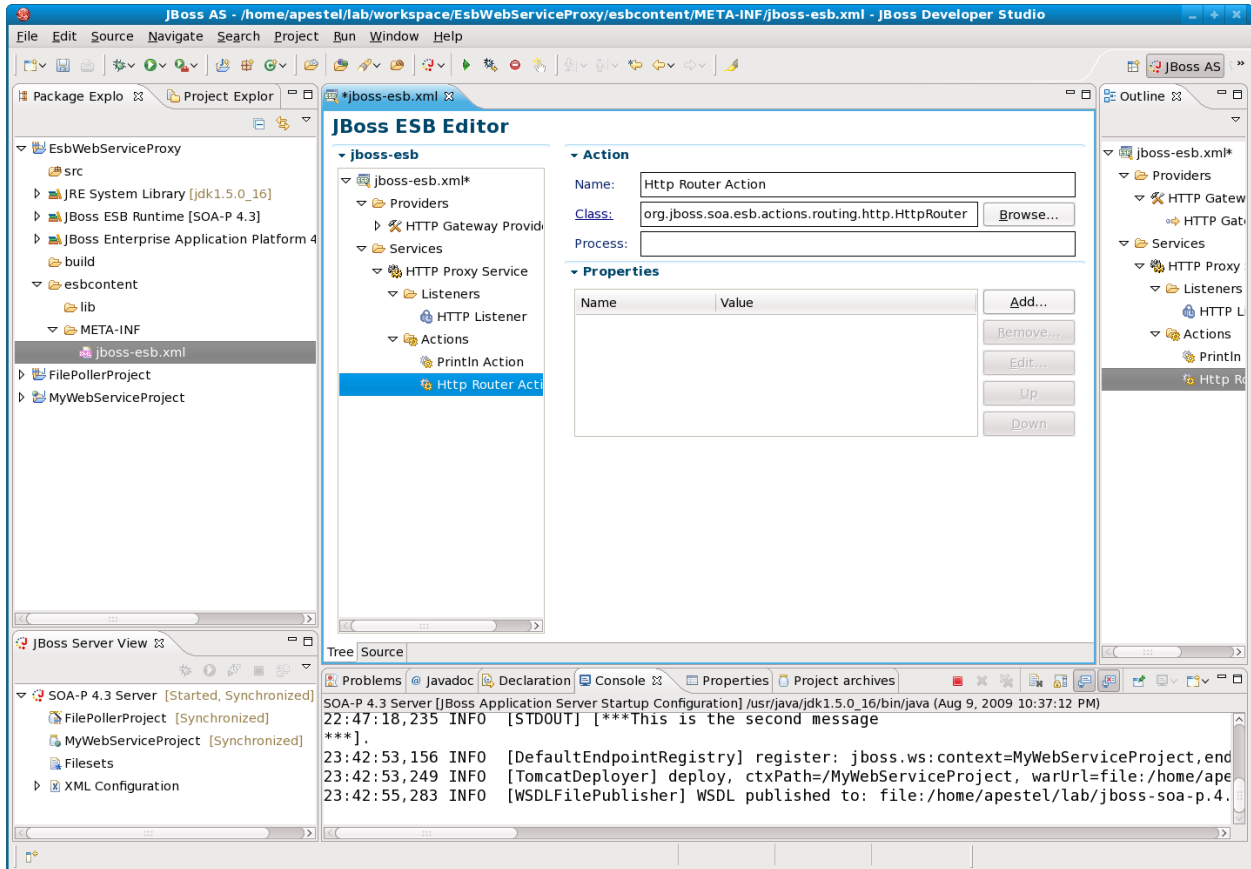
Name the action and specify the label that will be printed prior to the message contents. Click “Finish”.



Lastly, we need to add an action that will route the ESB message to our actual web service implementation. Right-click on Actions and choose “New | Generic Action...”.



You can type the class name in by hand or click “Browse” and start typing “HttpRouter”. It will show you the full class name that you can select. Once the dialog box is filled in as above, click “Finish”.

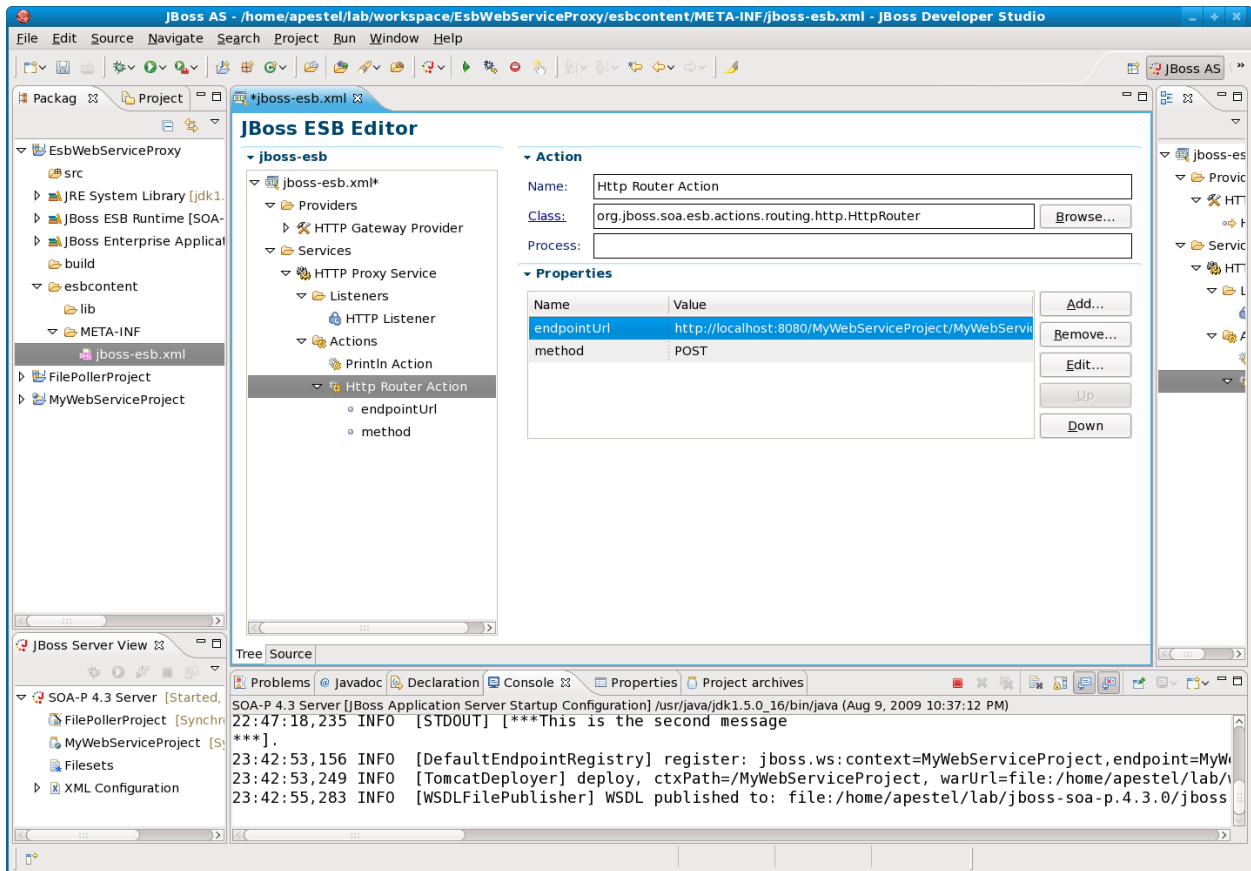


This action we just created needs some configuration by specifying properties. Other actions (including custom actions) can take advantage of having properties passed in as well. For the HttpRouter action, we need to specify the following properties:

```

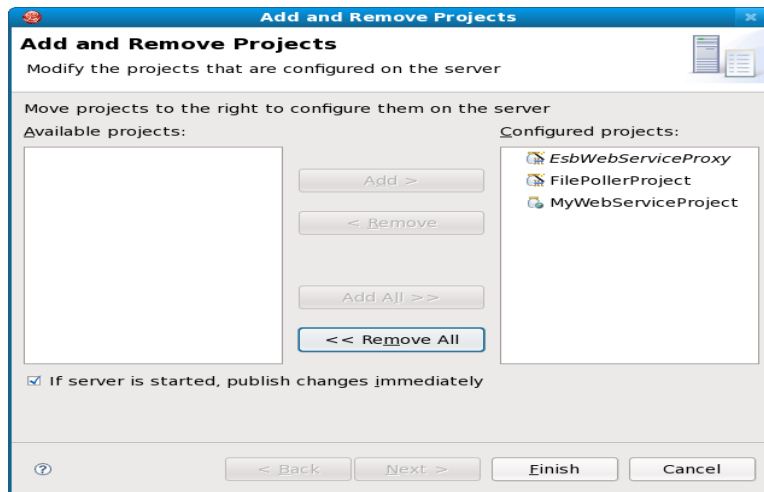
endpointUrl=http://localhost:8080/MyWebServiceProject/MyWebService
method=POST
    
```

To do this, click the “Add” button in the properties section of the Http Router Action. When done, the action configuration should look like this:

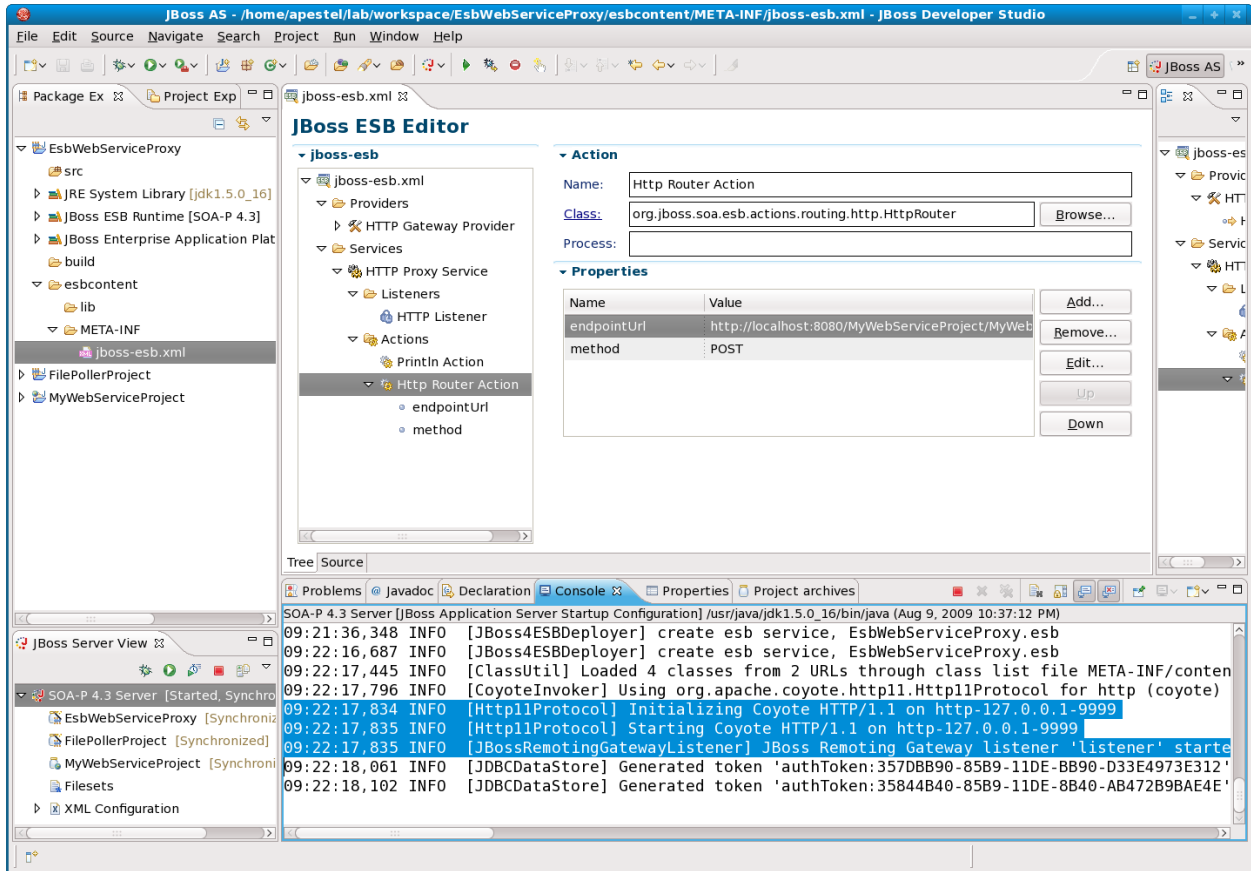


If you have any problems adding these properties, please raise your hand.

That's all we have to do in order to create the proxy in the ESB configuration. Now, we need to save (File | Save All) and publish this ESB project to our server. In the JBoss Server View (lower left portion of JBDS), right click on "SOA-P 4.3 Server" and choose "Add and Remove Projects...".

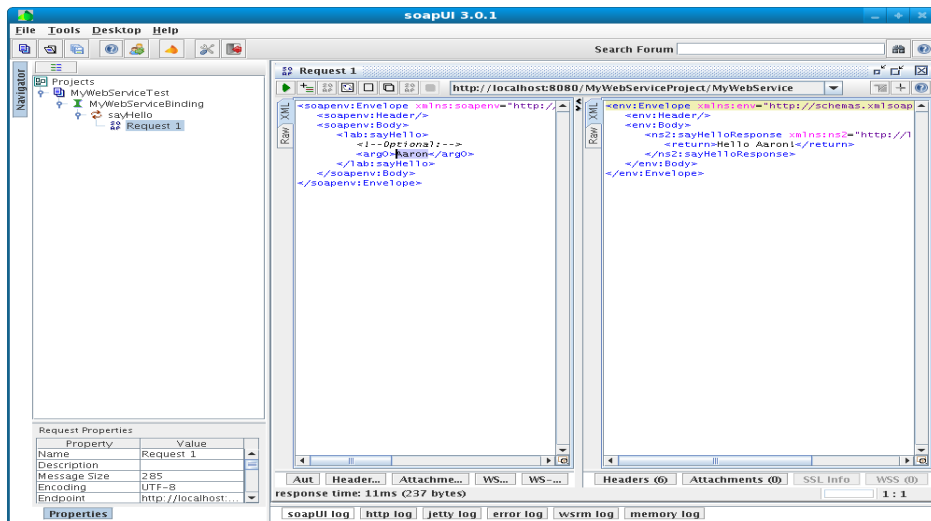


Make sure the "EsbWebServiceProxy" project is in the Configured projects list and click "Finish".

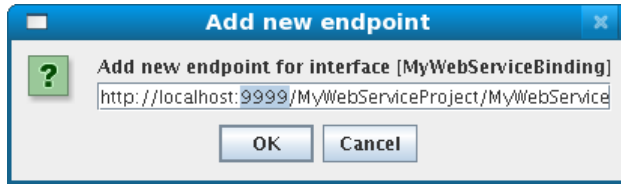


As highlighted in the console above, you should see that the HTTP gateway was started listening on port 9999.

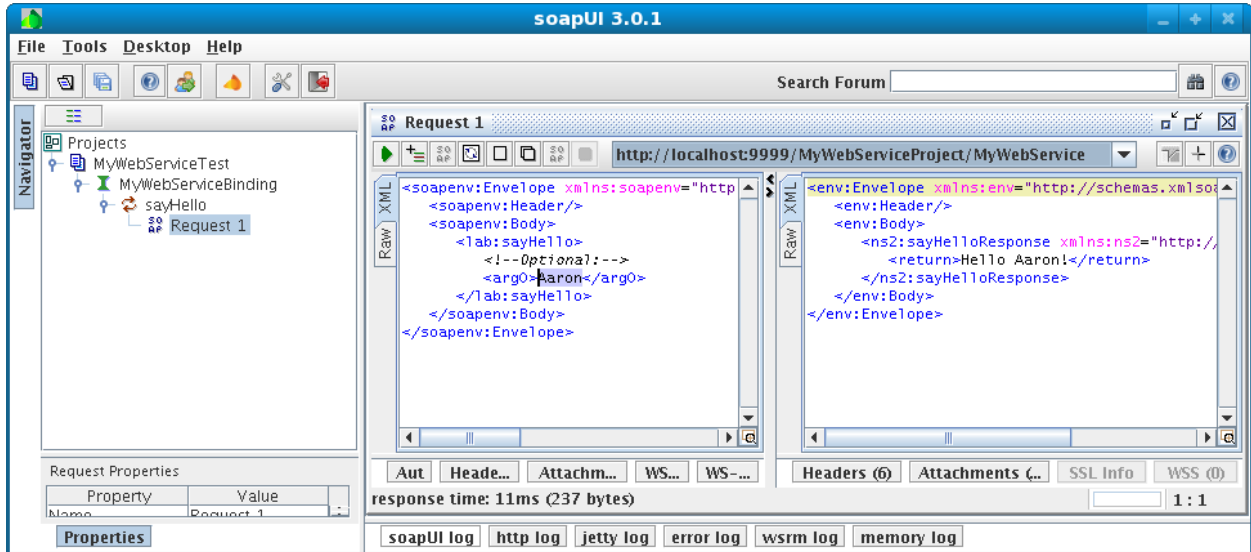
Now, we need to see if we can go back to SoapUI and pass the same request to the ESB that we had been passing to our JSR 181 web service.



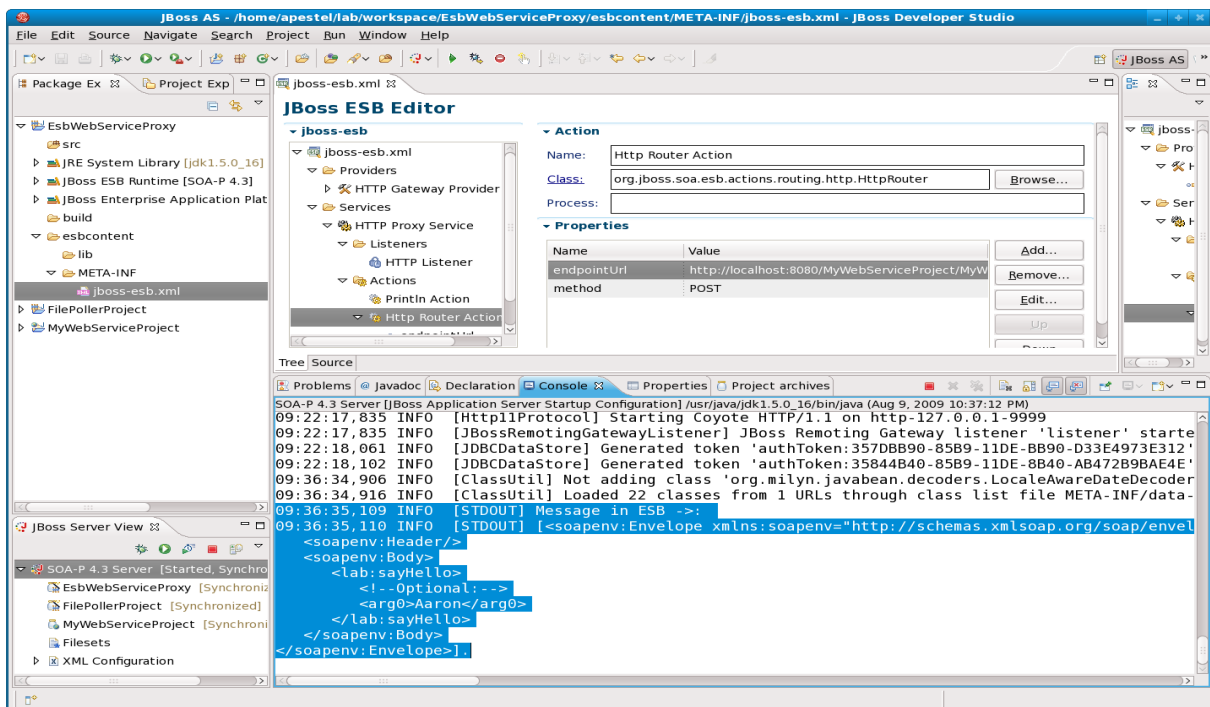
Notice that the web service URL here is pointing to “localhost:8080”. Select that URL drop list and choose “Add new endpoint”. Change the port to 9999 as shown below.



Click "OK" and notice that the endpoint in our SoapUI window is now pointing at localhost:9999 and we haven't changed anything else about the webservice.



Now click the green arrow button to run the web service. It should give us the same response, but have been passed through the ESB instead of directly invoking the JSR 181 web service. How can we know it went through the ESB? Let's look at our server console again in JBDS.

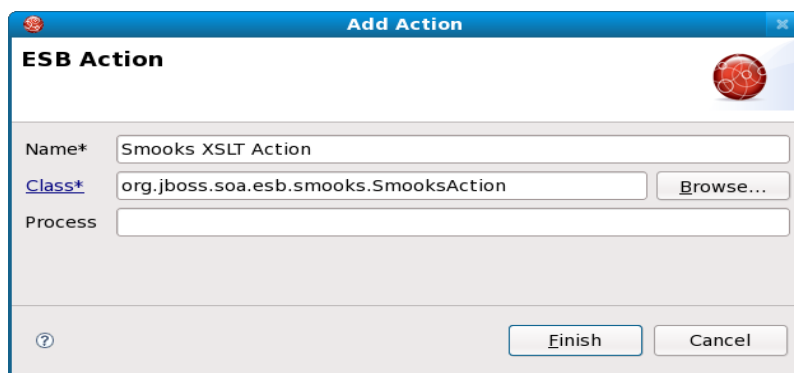


Next, we will add a transformation to show how the ESB can modify the message on the bus before or after invoking the back end web service.

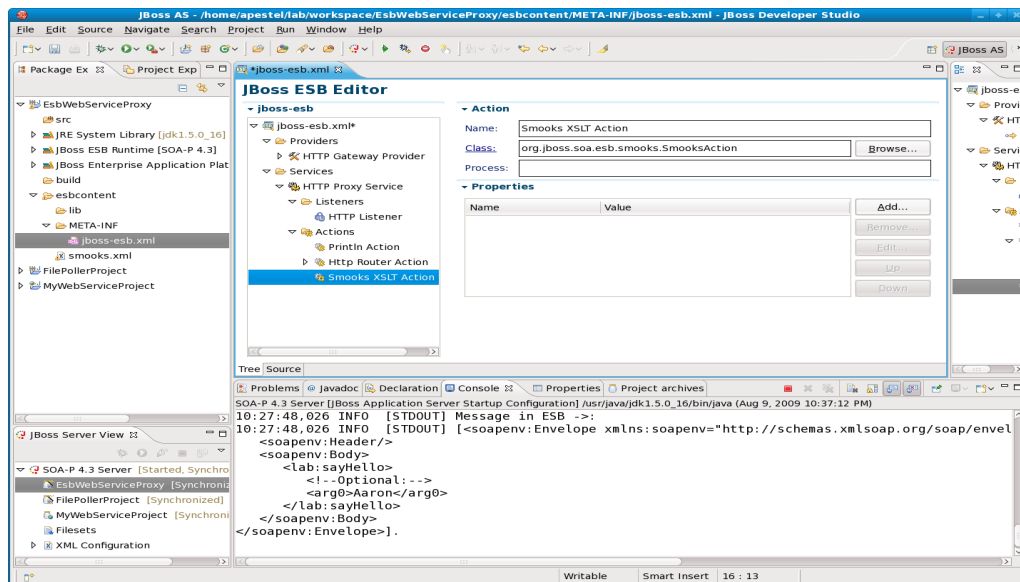
Lab #10: Adding XSLT to WS Proxy on ESB

JBoss SOA-P currently uses Smooks for transformations. Smooks can support many types of transformations (XML to XML, Java to XML, XML to Java, Java to Java, etc.). For this lab, we are going to show using Smooks to apply a simple XSLT transformation to the response coming back to the SoapUI client after the ESB receives the response from the back end web service.

First thing we need to do is add a new Smooks action to our action chain for our HTTP Proxy service. So, right-click on “Actions” under the HTTP Proxy Service and select “New | Generic Action...”.



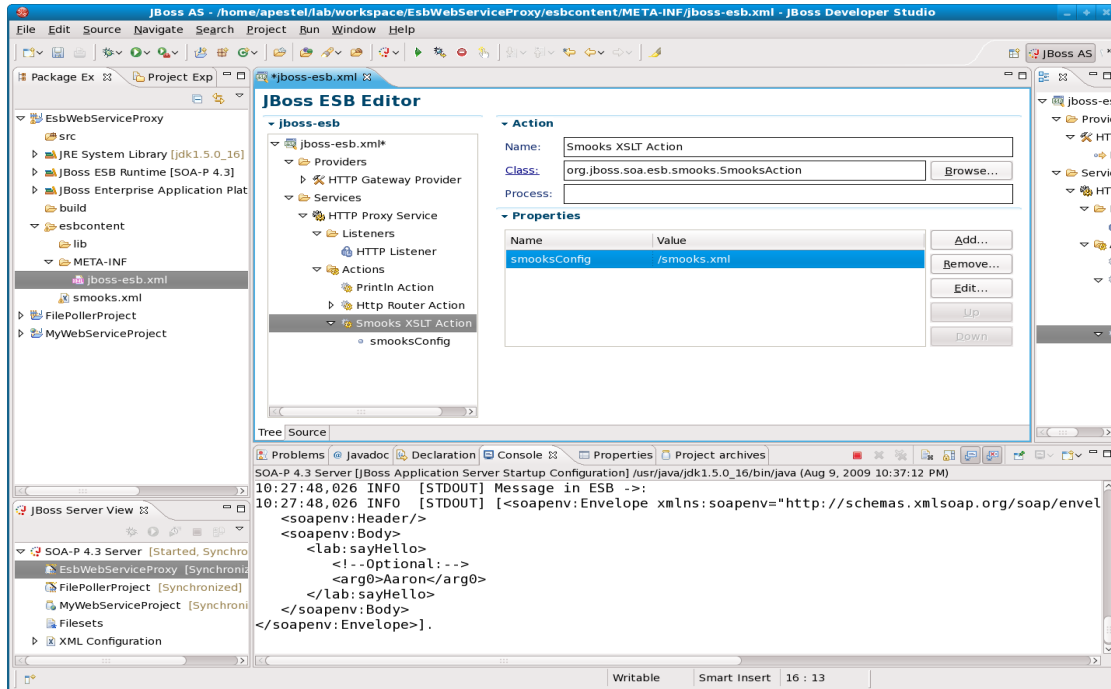
Add a name and the SmooksAction class as shown above and click “Finish”.



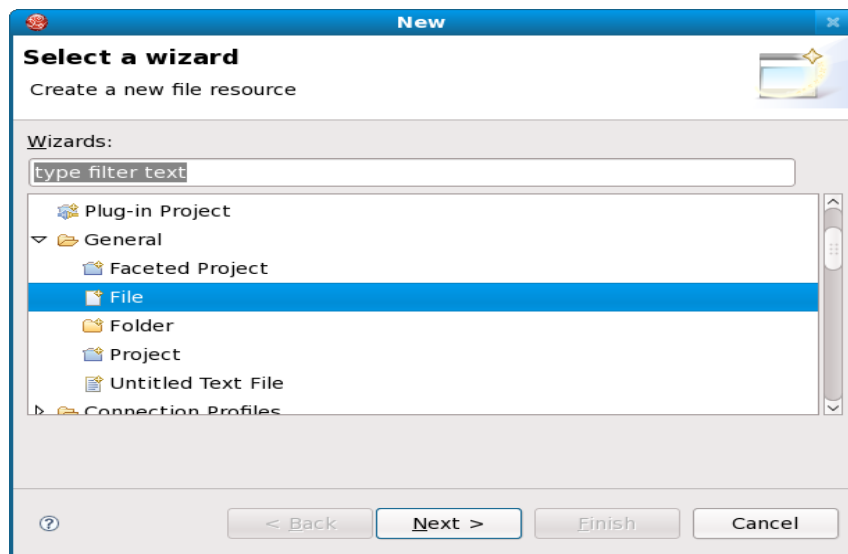
Now we have our Smooks action, but need to add a property to it telling it where the Smooks configuration file is. So, click the “Add...” button in the properties section of this action and add this property:

smooksConfig=/smooks.xml

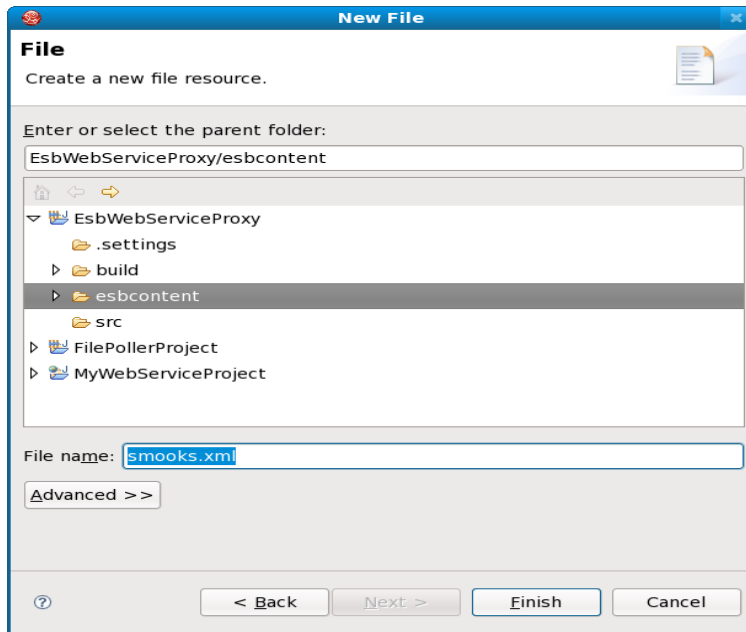
You should see the property like this below.



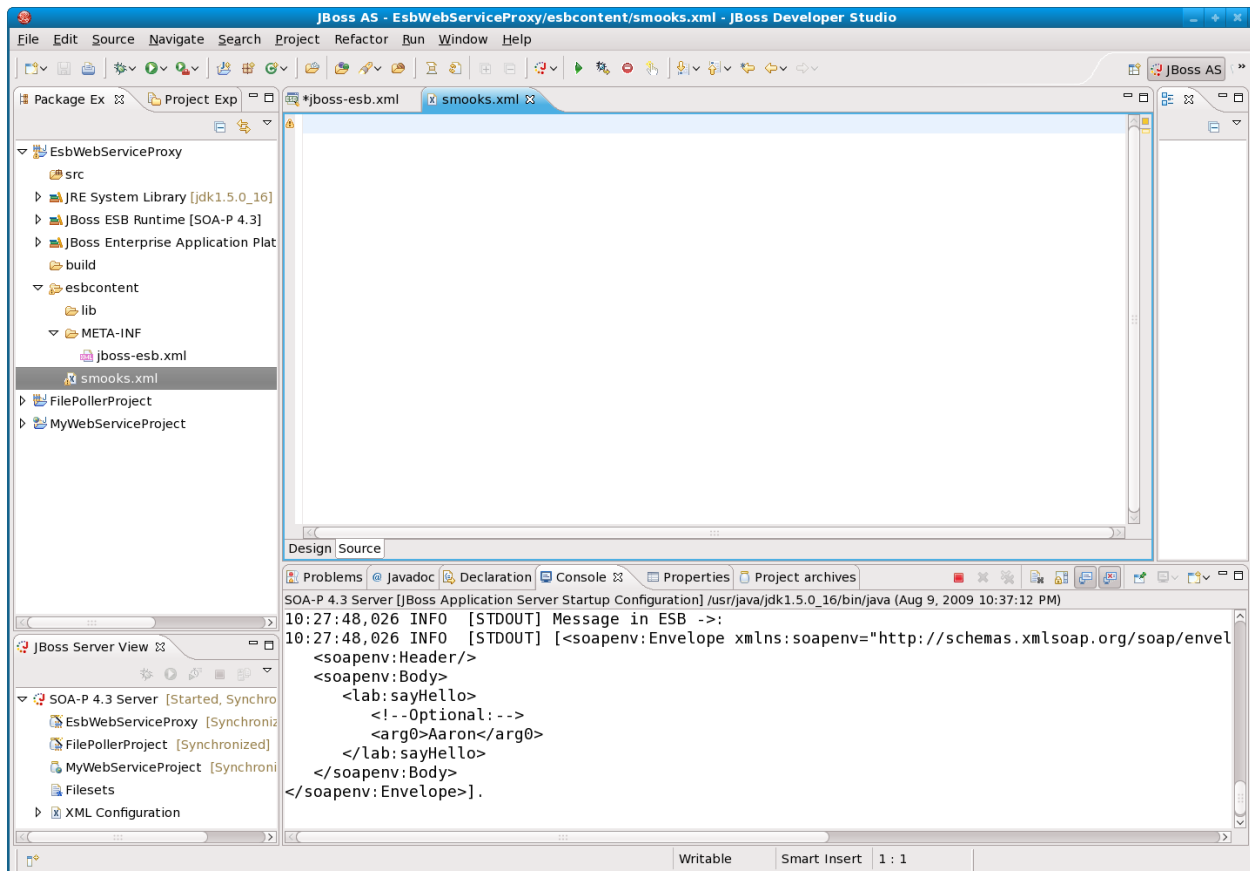
Now, we need to create the smooks.xml file that we just referenced. If you just saved the project and got an error because the file wasn't there, don't worry, we'll fix that now. So, on the left side of JBDS in the project explorer, right-click on “esbcontent” and choose “New | Other...”.



Choose a “General | File” as shown above and click “Next”.



Name the file “smooks.xml” as shown above and click “Finish”. This will open an empty XML file editor like shown below.



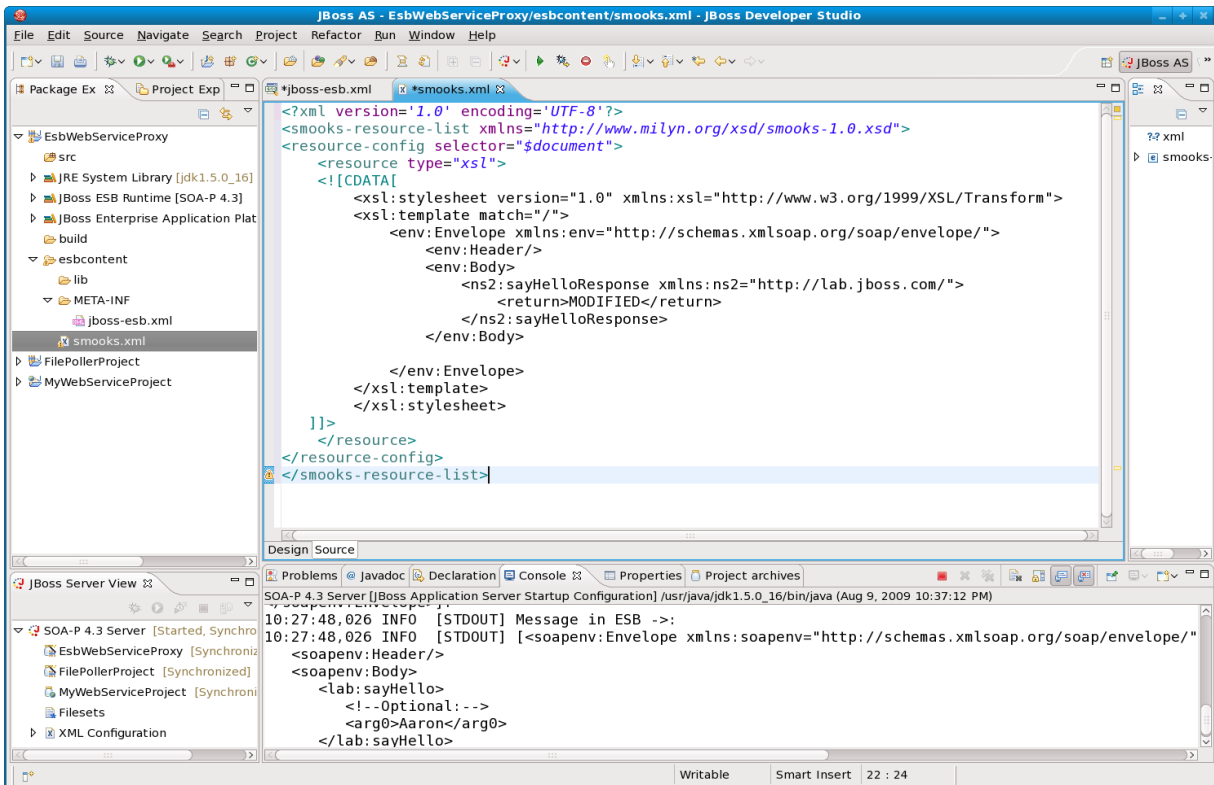
Make sure to select the “Source” tab of this editor. Now, past the following XML into that source tab.

```

<?xml version='1.0' encoding='UTF-8'?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.0.xsd">
<resource-config selector="$document">
  <resource type="xsl">
    <![CDATA[
      <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
        <xsl:template match="/">
          <env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">
            <env:Header/>
            <env:Body>
              <ns2:sayHelloResponse xmlns:ns2="http://lab.jboss.com/">
                <return>MODIFIED</return>
              </ns2:sayHelloResponse>
            </env:Body>
          </env:Envelope>
        </xsl:template>
      </xsl:stylesheet>
    ]]>
  </resource>
</resource-config>
</smooks-resource-list>

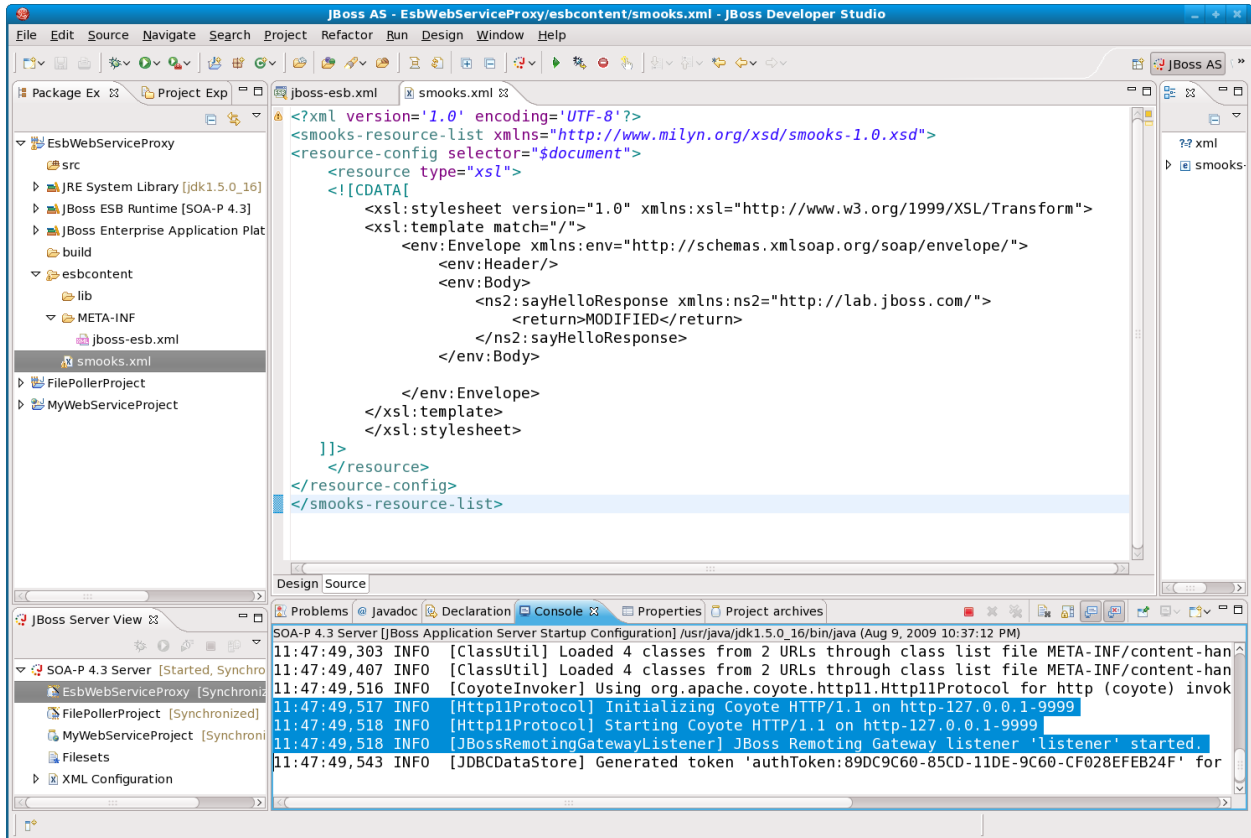
```

Now the editor should look like this below.

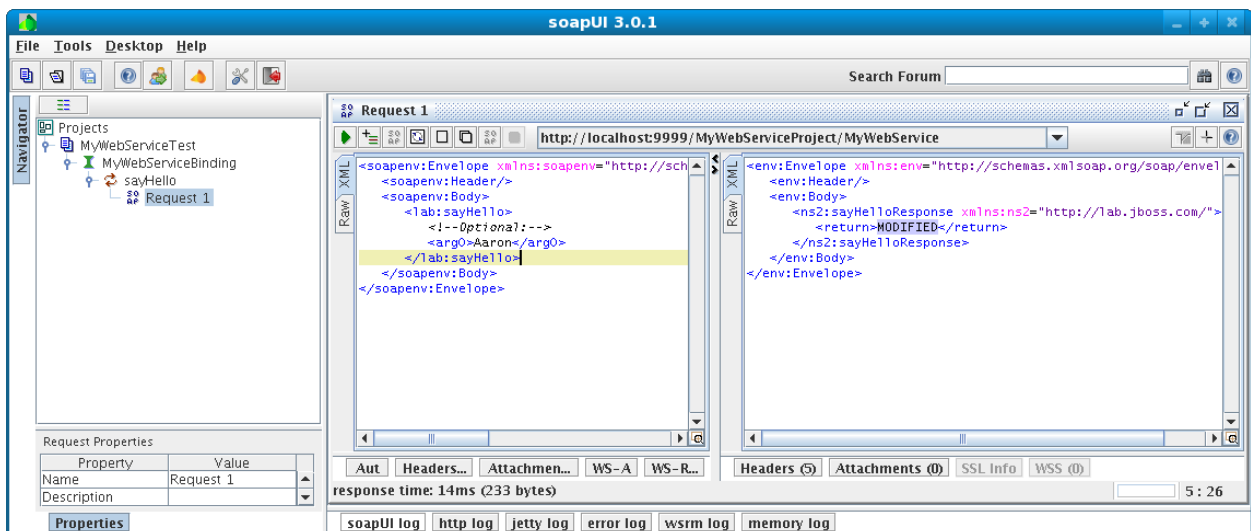


This is not a terribly interesting XSLT, it's basically changing all the XML. But folks that are good with XSLT can obviously do more more interesting and useful things in their XSLT.

Now, we need to save the project via “File | Save All” and republish. To republish, go to the lower left of JBDS in the JBoss Server View and right-click on EsbWebServiceProxy and choose “Full Publish”. The console should show that the ESB service was redeployed as highlighted below.



Now, we just need to go back to SoapUI and hit this web service on the ESB once more and we'll see that the ESB transforms the result to be “MODIFIED” regardless of what name we send as a parameter as shown below.



Congratulations on adding a Smooks XSLT to your ESB proxy service!

Lab #11: Adding CBR to WS Proxy on ESB

For this lab, we are going to add two simple print line services to our EsbWebServiceProxyProject. In addition we will add a very simple XPath content based router action to our existing ESB service that will route to one of the two new services depending on the SOAP message passed in.

We could go through and add the services as we have done in the past, but for this lab, we're just going to add the services by editing the jboss-esb.xml file manually. In some cases, copy/paste in the source view of jboss-esb.xml can be more convenient than using the graphical ESB editor.

Below is the new jboss-esb.xml file (you can copy/paste the entire file - changes are two new services and new content based router action) into the "Source" tab of the jboss-esb.xml file:

```

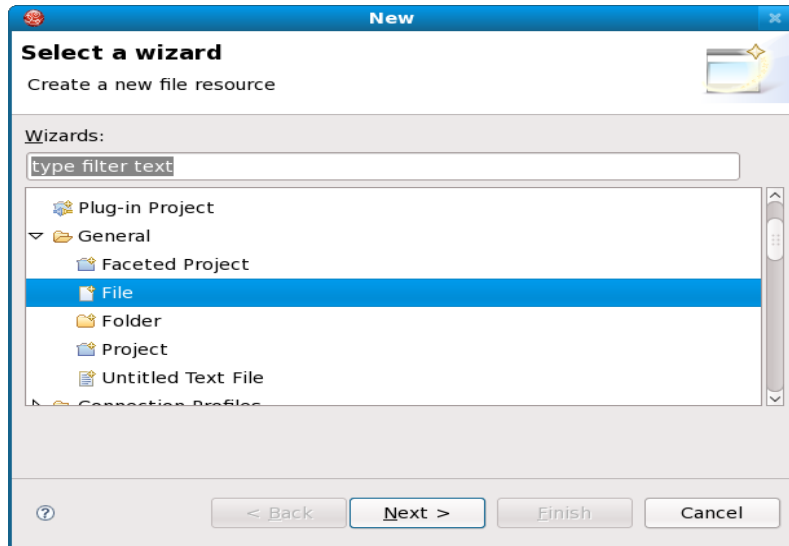
<?xml version="1.0"?>
<jbossesb parameterReloadSecs="5"
xmlns="http://anonsvn.labs.jboss.com/labs/jbossesb/trunk/product/etc/schemas/xml/jbossesb-1.0.1.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://anonsvn.labs.jboss.com/labs/jbossesb/trunk/product/etc/schemas/xml/jbossesb-1.0.1.xsd http://anonsvn.labs.jboss.com/labs/jbossesb/trunk/product/etc/schemas/xml/jbossesb-1.0.1.xsd">
  <providers>
    <jbr-provider name="HTTP Gateway Provider" protocol="http">
      <jbr-bus busid="HTTP Gateway Provider" port="9999"/>
    </jbr-provider>
  </providers>
  <services>
    <service category="lab services"
description="Service to proxy web services" invmScope="GLOBAL" name="HTTP Proxy Service">
      <listeners>
        <jbr-listener busidref="HTTP Gateway Provider" is-gateway="true" name="HTTP Listener"/>
      </listeners>
      <actions>
        <action class="org.jboss.soa.esb.actions.SystemPrintln" name="Println Action">
          <property name="message" value="Message in ESB ->"/>
        </action>
        <action class="org.jboss.soa.esb.actions.ContentBasedRouter" name="ContentBasedRouter">
          <property name="ruleSet" value="cbr.drl"/>
          <property name="ruleLanguage" value="XPathLanguage.dsl"/>
          <property name="ruleReload" value="true"/>
          <property name="destinations">
            <route-to destination-name="ServiceA" service-category="category" service-name="ServiceA"/>
            <route-to destination-name="ServiceB" service-category="category" service-name="ServiceB"/>
          </property>
        </action>
        <action class="org.jboss.soa.esb.actions.SystemPrintln" name="after cbr">
          <property name="message" value="after cbr"/>
        </action>
        <action class="org.jboss.soa.esb.actions.routing.http.HttpRouter" name="Http Router Action">
          <property name="endpointUrl" value="http://localhost:8080/MyWebServiceProject/MyWebService"/>
          <property name="method" value="POST"/>
        </action>
        <action class="org.jboss.soa.esb.smooks.SmooksAction" name="Smooks XSLT Action">
          <property name="smooksConfig" value="/smooks.xml"/>
        </action>
      </actions>
    </service>
    <service category="category" description="asdf" invmScope="GLOBAL" name="ServiceA">
      <actions>
        <action class="org.jboss.soa.esb.actions.SystemPrintln" name="ServiceA">
          <property name="message" value="ServiceA"/>
        </action>
      </actions>
    </service>
    <service category="category" description="asdf" invmScope="GLOBAL" name="ServiceB">
      <actions>
        <action class="org.jboss.soa.esb.actions.SystemPrintln" name="ServiceB">
          <property name="message" value="ServiceB"/>
        </action>
      </actions>
    </service>
  </services>
</jbossesb>

```

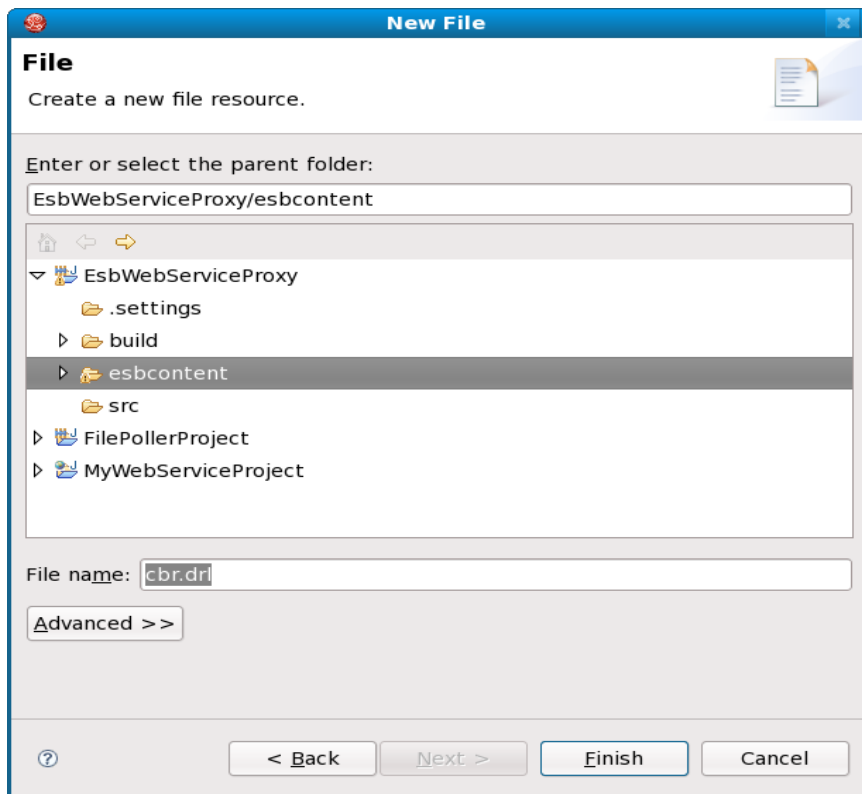
Please note that the only thing ServiceA and ServiceB do is print out the service name and the body of the ESB message. Also, please note that this content based router is placed before the http router.

This means that the http router will never get invoked, so when we invoke this service, we won't get the response from the actual web service. In a real world situation, a CBR would probably be the last action in the action chain.

You should also notice that the CBR action is referring to a cbr.drl file. This is the file that will actually contain the routing rules that are based on JBoss Rules. You'll need to add this file by right-clicking on "esbcontent" in the explorer tab of the ESB project and selecting "New | Other...".



Choose "General | File" as shown above and click "Next".



Enter filename “cbr.drl” as shown above and click “Finish”.

That should open the new empty file in a text editor and you can copy/paste the following text in the editor:

```
-----
package com.jboss.soa.esb.routing.cbr

#list any import classes here.
import org.jboss.soa.esb.message.Message;
import org.jboss.soa.esb.message.format.MessageType;

expander XpathLanguage.dsl

#declare any global variables here
global java.util.List destinations;

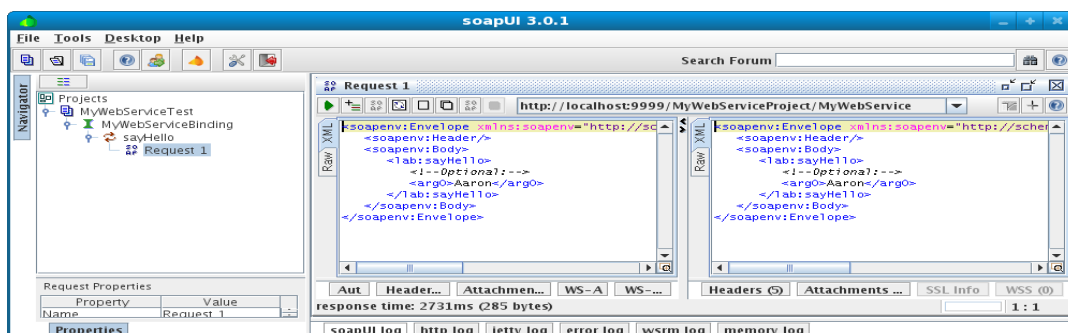
rule "Blue Routing Rule using XPATH"
when
    xpathMatch expr "/soapenv:Envelope/soapenv:Body/lab:sayHello/arg0[.='Aaron']" use namespaces "soapenv=http://schemas.xmlsoap.org/soap/envelope/lab=http://lab.jboss.com/"
then
    Log : "ServiceA";
    Destination : "ServiceA";
end

rule "Red Routing Rule using XPATH"
when
    xpathMatch expr "/soapenv:Envelope/soapenv:Body/lab:sayHello/arg0[!.='Aaron']" use namespaces "soapenv=http://schemas.xmlsoap.org/soap/envelope/lab=http://lab.jboss.com/"
then
    Log : "ServiceB";
    Destination : "ServiceB";
end
-----
```

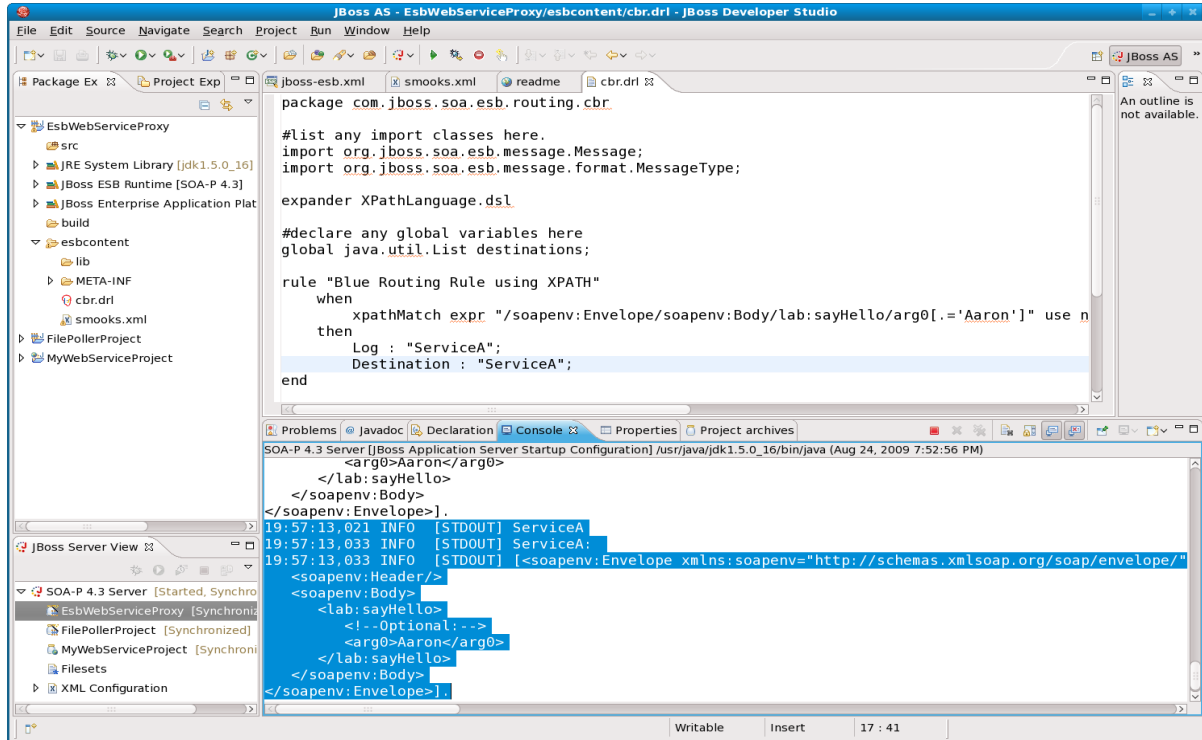
This is what the XPath routing rules look like. You'll see there are two routing rules. The first sets the routing destination to ServiceA if we say hello to “Aaron” in the SOAP message. Else, it routes to ServiceB. You can also do CBR without XPath if the payload of the ESB message were not XML.

Next we, need to redeploy. So select “File | Save All” and then right-click on the “ESBWebServiceProxy” deployment under the SOA-P 4.3 Service in the JBoss AS view on the lower left of the JBDS window and select “Full Publish”.

Now, we can invoke this service again from SoapUI as shown below.



But, we can look at the server console log to see what ESB service got routed to.



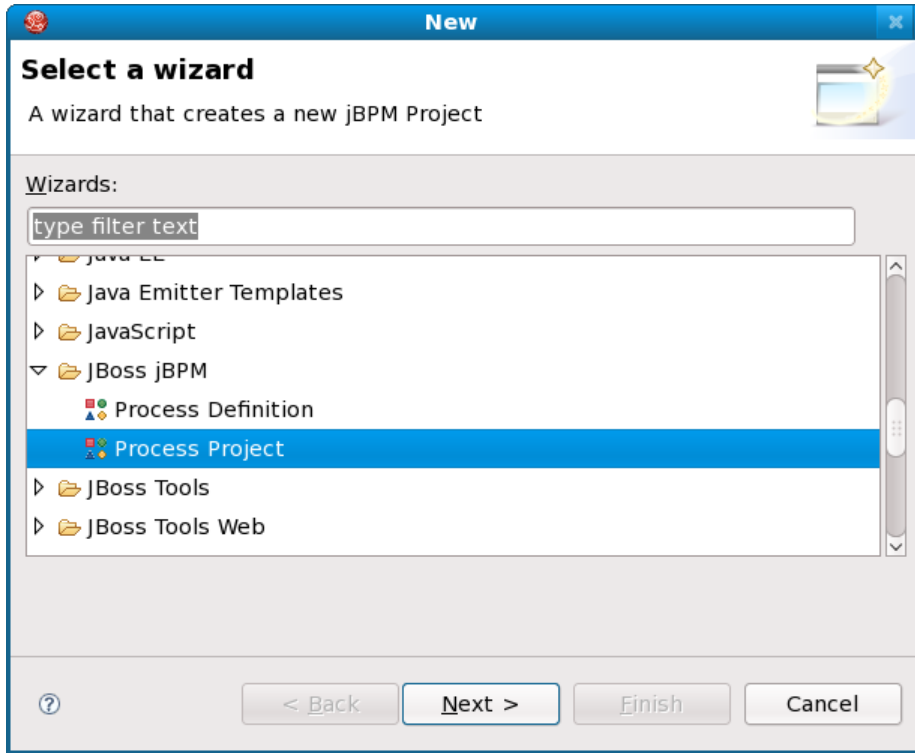
As shown above, you'll notice that ServiceA was invoked. Now, go back to SoapUI and change the parameter to something other than "Aaron" and you should see in the server log that ServiceB was invoked instead of ServiceA.

Congratulations on writing your first XPath based content based router!

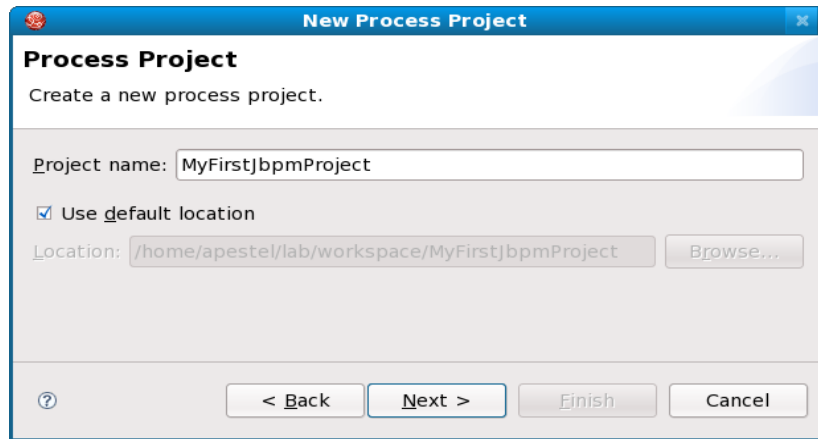
Lab #12: Using jBPM

For our last lab, we're going to create a simple jBPM project. We could do a whole workshop on jBPM, so we're clearly not going to cover all the features in this single lab. However, the goal of this lab is help you gain the fundamental skills necessary to create a basic jBPM project, deploy the project to the server, and test the service in the jBPM administration console.

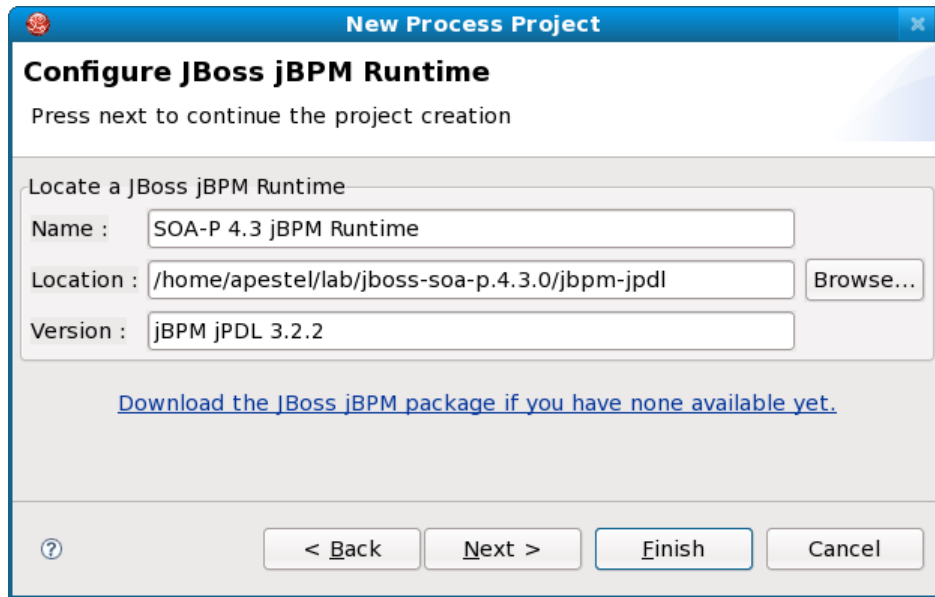
The first thing we need to do is create a new jBPM project. So, select "File | New | Other...".



Select a JBoss jBPM Process Project as shown above and click “Next”.

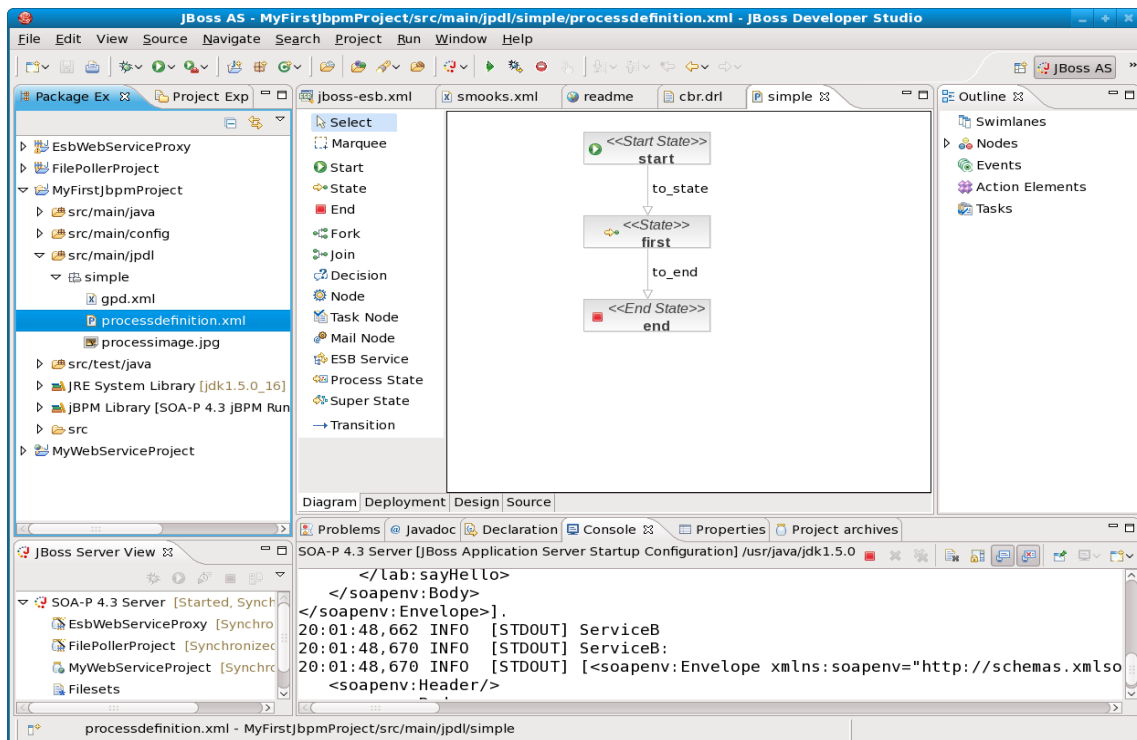


Give the project a name as shown above and click “Next”.

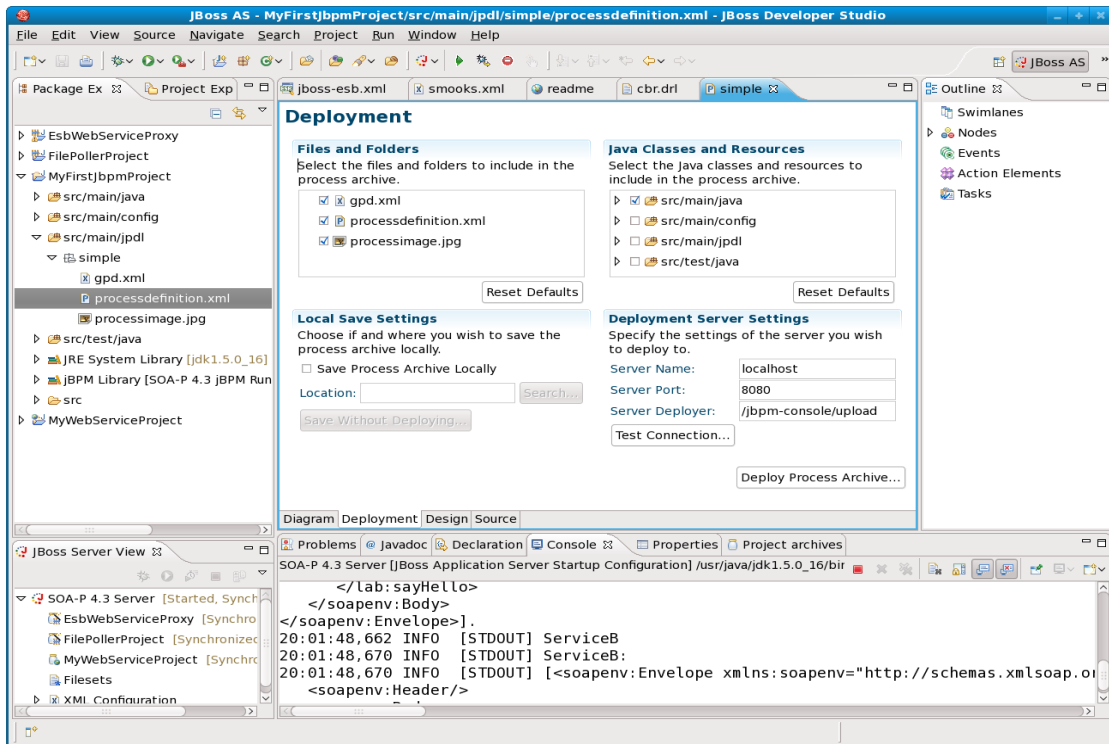


Fill out the runtime information as shown above (the version should be filled in automatically after specifying the location), and click “Finish”.

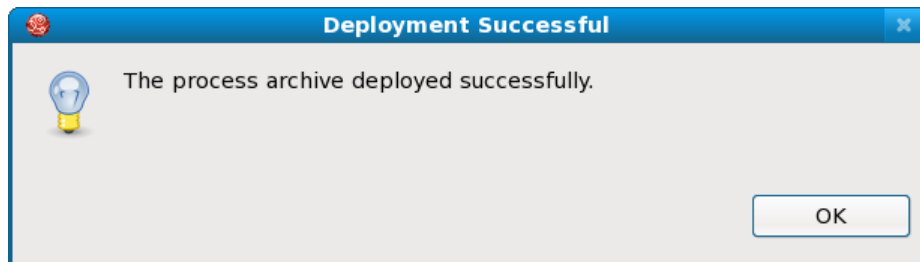
This will create a new jBPM Project and you can view the main jBPM process diagram by double clicking the process definition file from the project explorer as shown below. You may see a warning when opening the process. If so, just select “Convert and Open”.



So, we now have a process! It's not terribly interesting, but we can deploy it as is to test the deployment mechanism. So, click the "Deployment" tab at the bottom of the jBPM process diagram editor. You will see something like shown below.



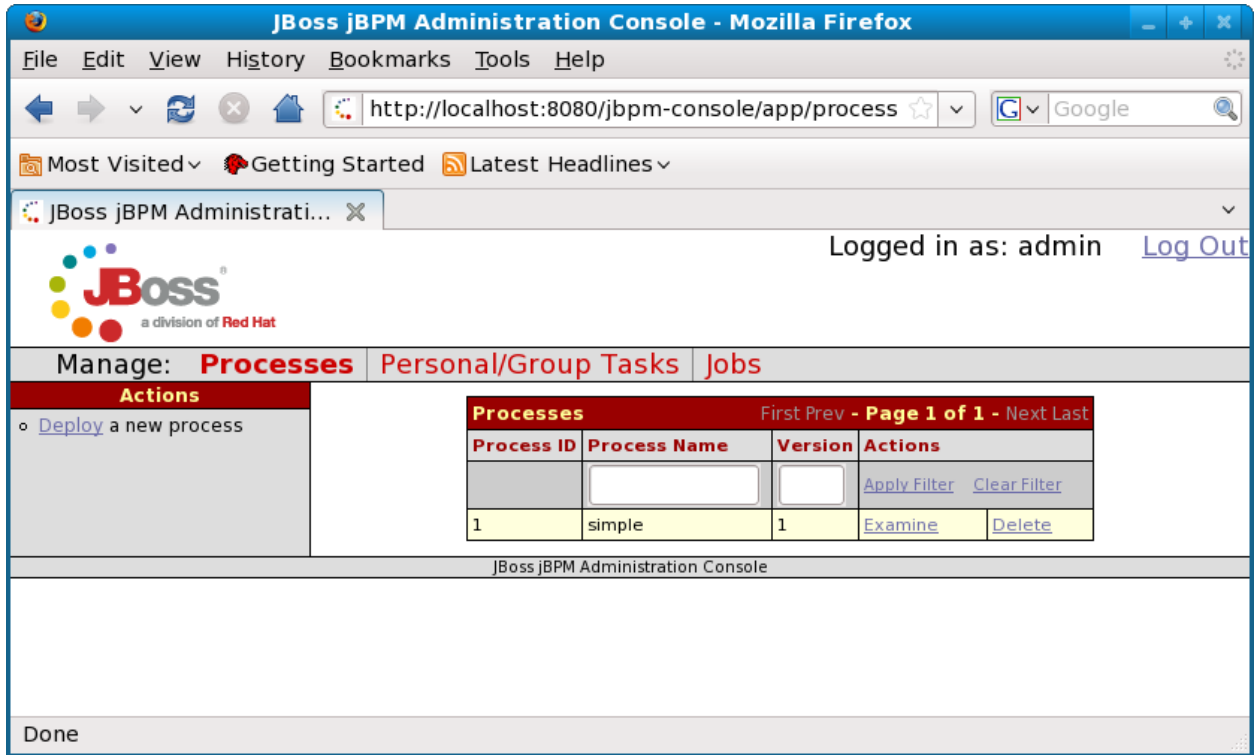
All the defaults should be correct, so we can just click the "Deploy Process Archive..." button.



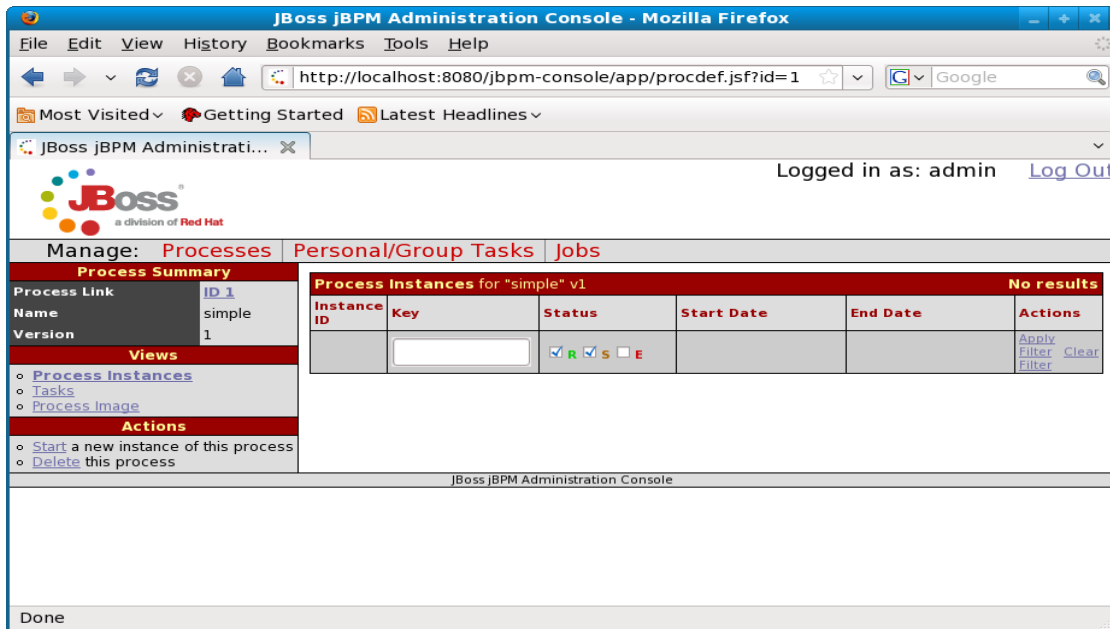
So, where did this process get deployed? Well, it actually invoked the jBPM application on the application server, passed in the process we defined, and stored it in the database configured under jBPM. There are four ways to deploy jBPM processes: 1.) From JBDS, as we just did, 2.) From the jBPM administration console, which we'll be looking at next, 3.) From an Ant task, and 4.) From Java code.

Now, we can open a browser window and go see this jBPM process that we deployed by going to this URL and logging in as admin/admin: <http://localhost:8080/jbpm-console>

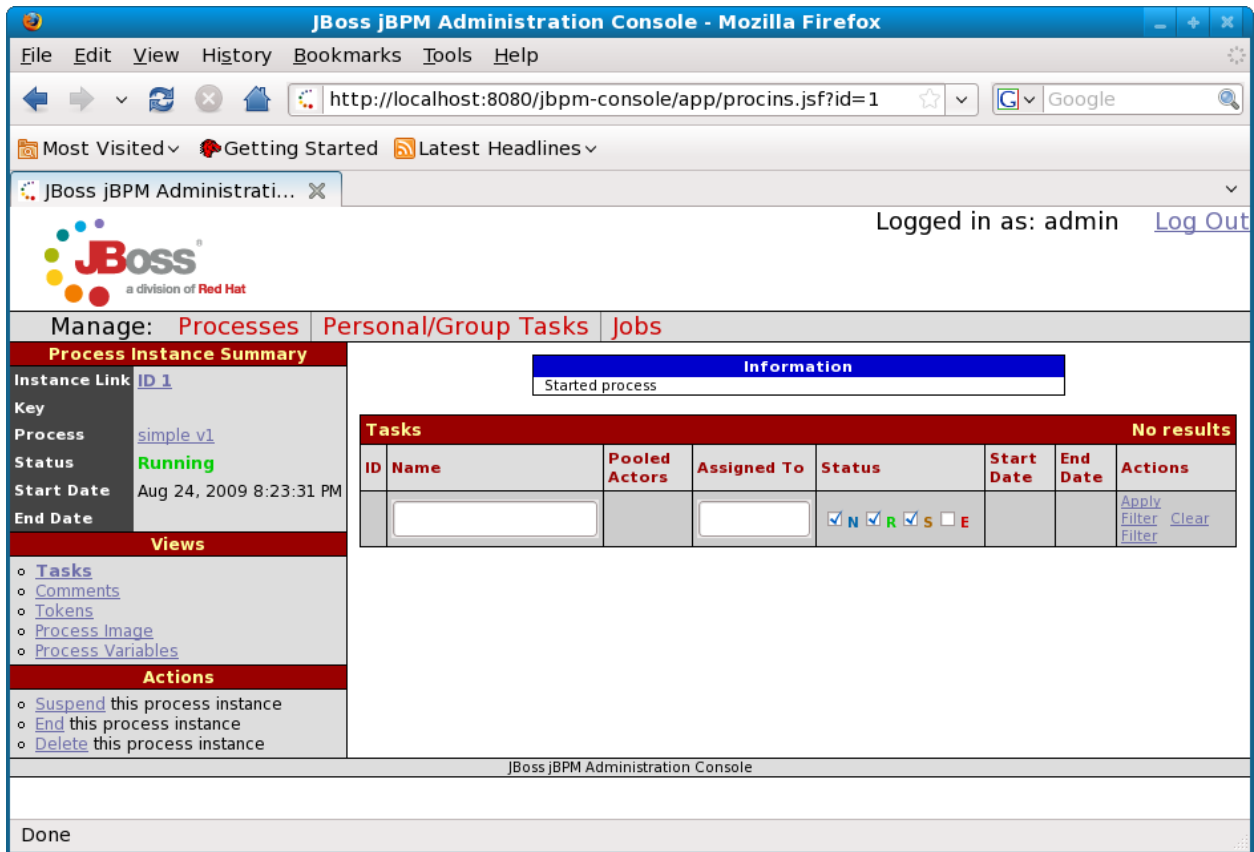
After logging in, you should see this:



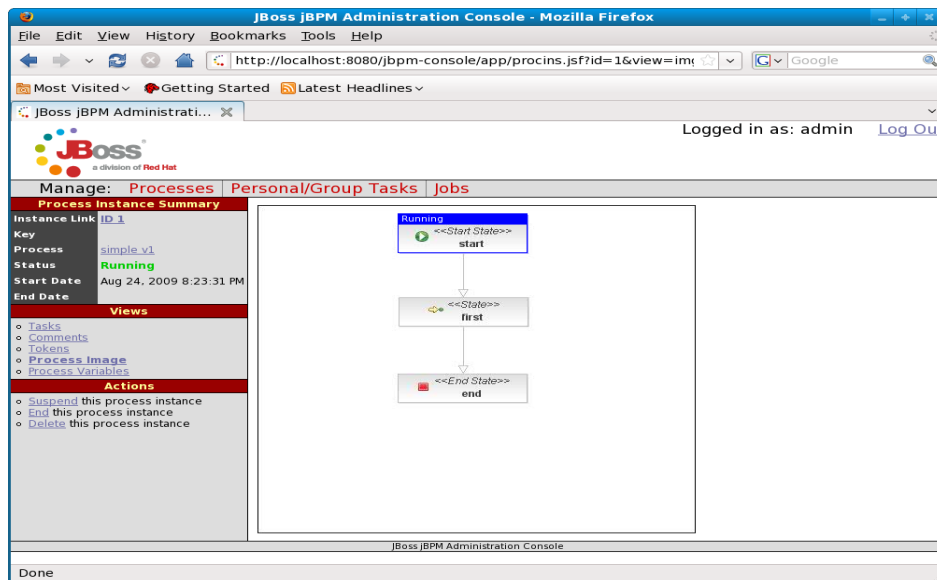
There is the jBPM process “definition” we created - version #1. The default process name is “simple”. We can click the “Examine” link to see more information about this process.



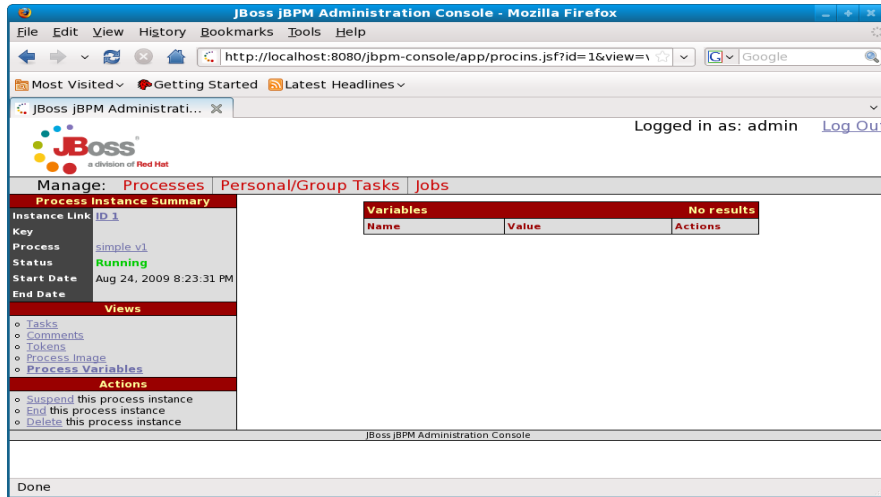
There are no process “instances” for this process “definition”. If there were, we'd see them in the Process Instances table. So, let's create an instance of this process by selecting the “Start a new instance of this process” on the left.



This created a new instance called “ID 1” that you can see next to “Instance Link” on the left under Process Instance Summary”. In a real scenario, this process would likely have been created by an ESB service or via the jBPM Java API. We can now click the “Process Image” link to see the what state this process is currently in.



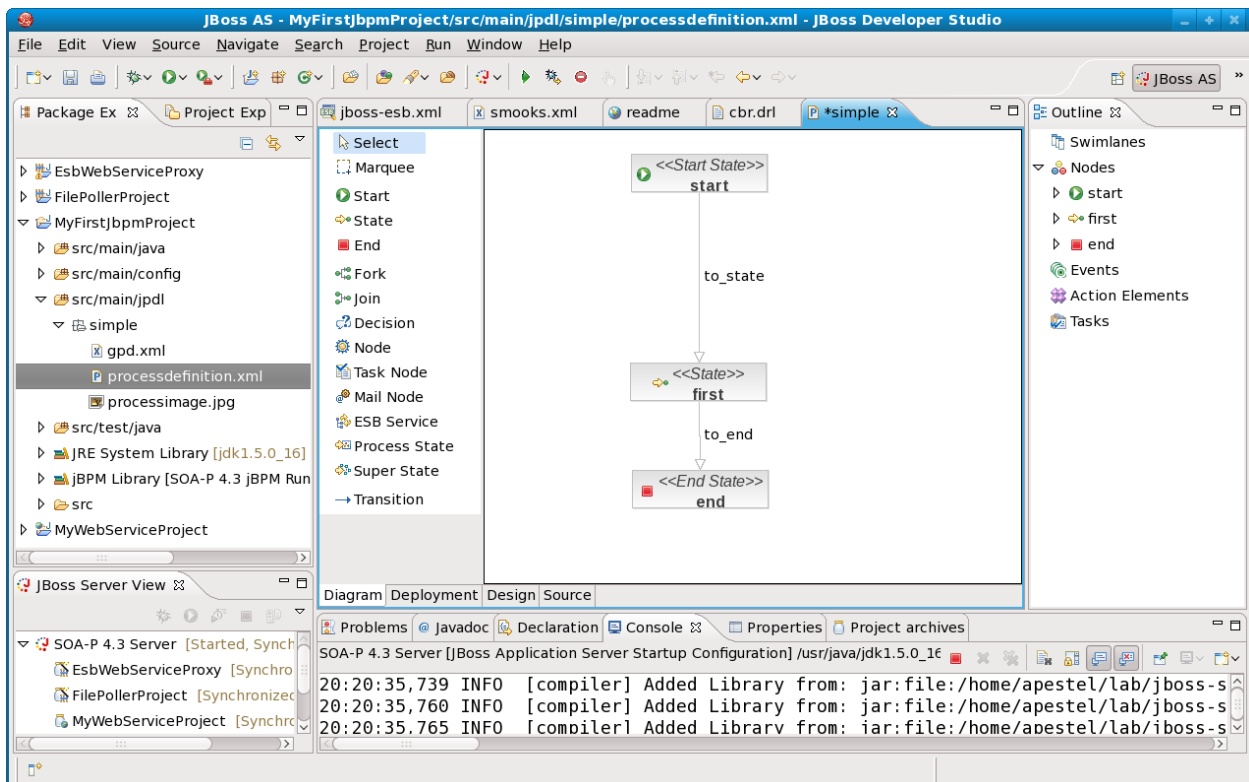
You'll notice that we are still in the first node, since it is highlighted blue. You can also see the variables associated with a process instance by clicking on the “Process Variables” link.



This simple process does not have any variables, but if it did, you would see them here - as we'll see shortly.

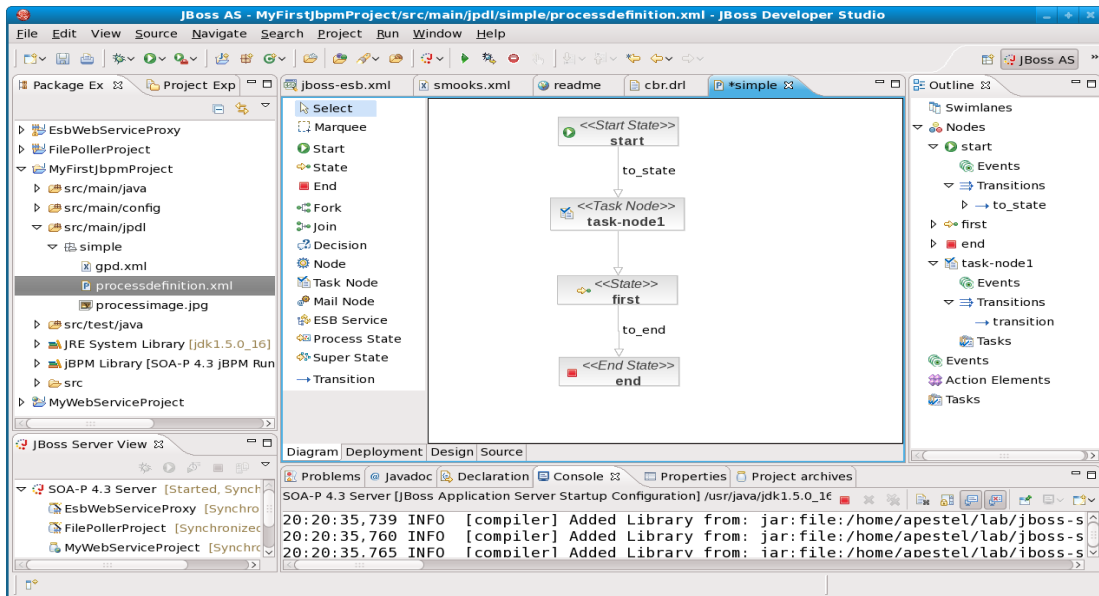
Ok, now that we've got some of the fundamentals down, let's go back to JBDS and add a simple user task to our jBPM process.

First, click back on the "Diagram" tab of the process editor as shown below.



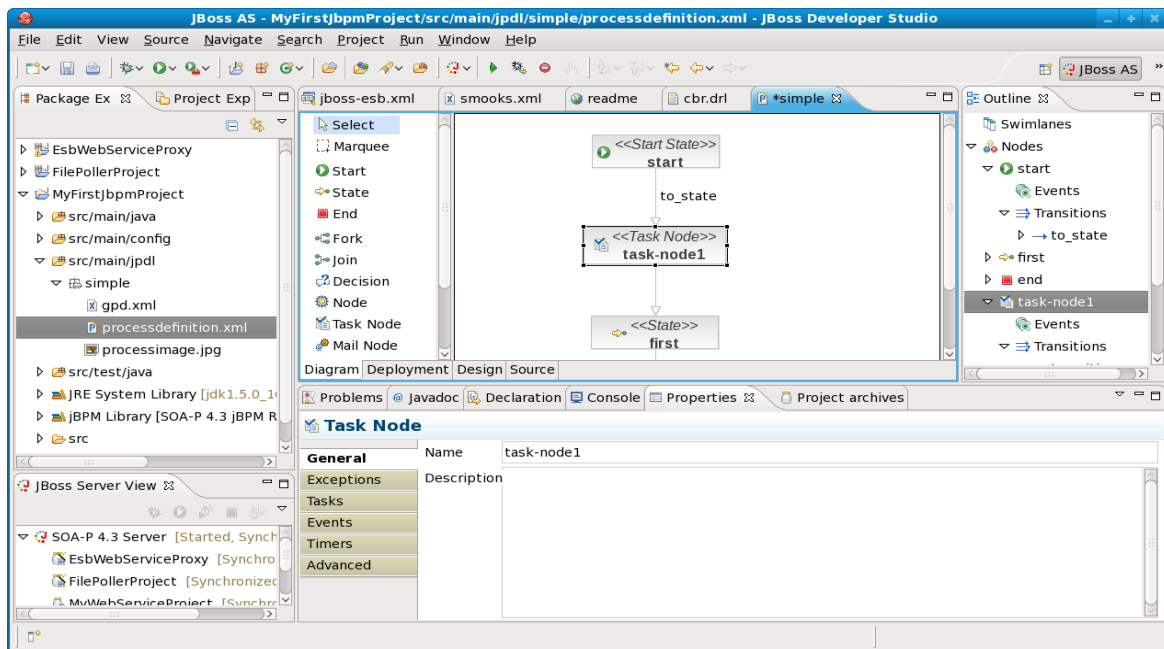
Now you'll get to use your drag and drop skills because we want to add a "Task Node" with transitions between the "start" and "first" nodes. Click "Task Node" and then click on the palate where you want it. Select the "to_state" transition and move it to the new task-node1. Then, click "Transition"

and click the task-node1 node and drag to the State node titled “first”. It should look like below when done.

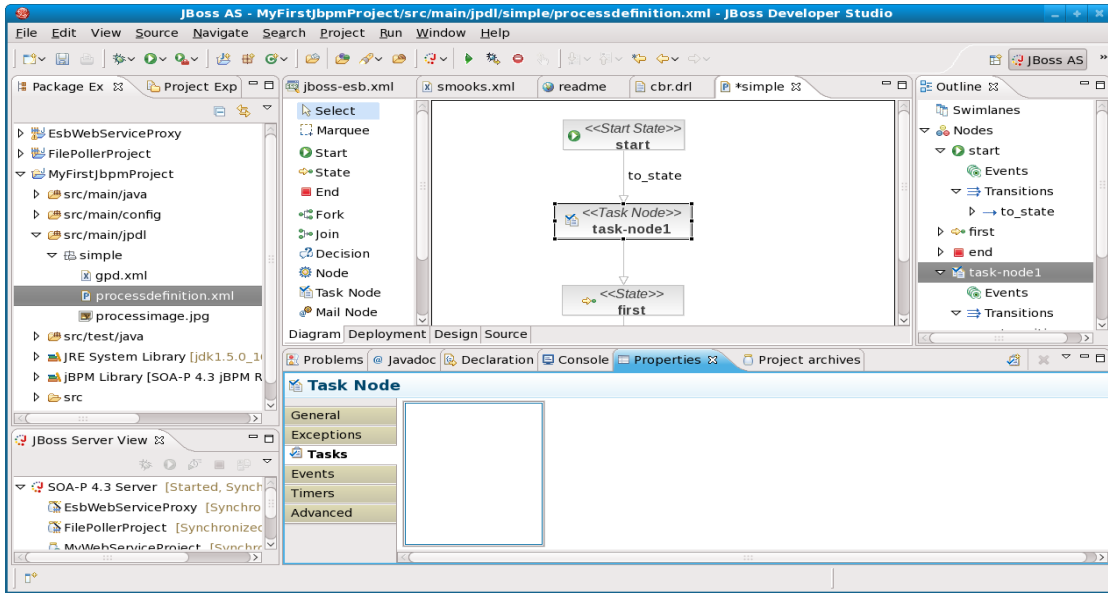


Make sure you have the transitions as shown above.

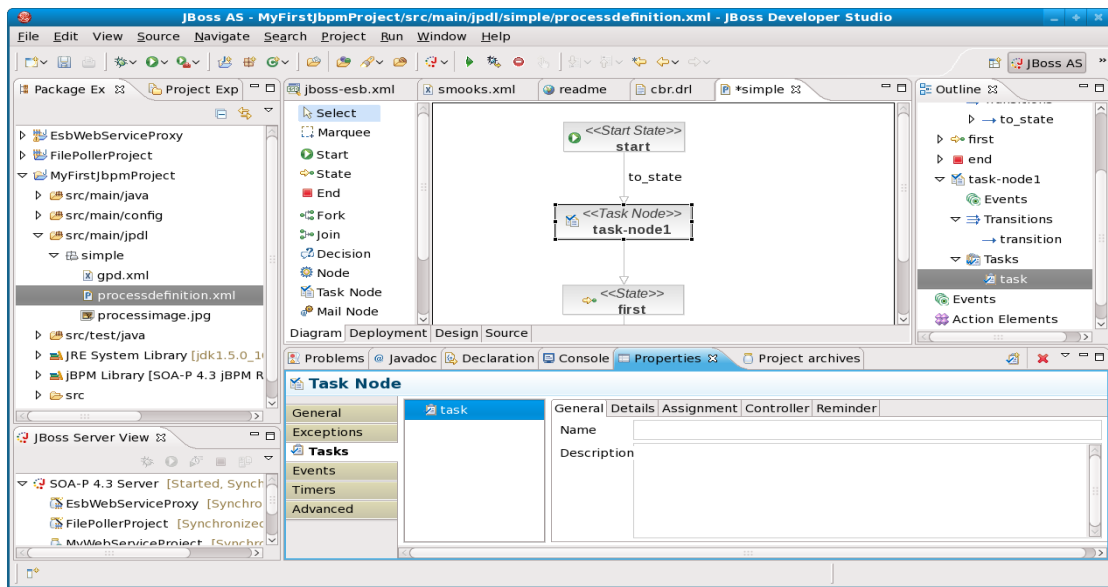
To edit the properties of the task node, we need to select the “Properties” tab at the bottom of the screen and then select the “task-node1” node. You should see this below.



In the Properties tab, we want to select the “Tasks” sub-tab. As shown below.

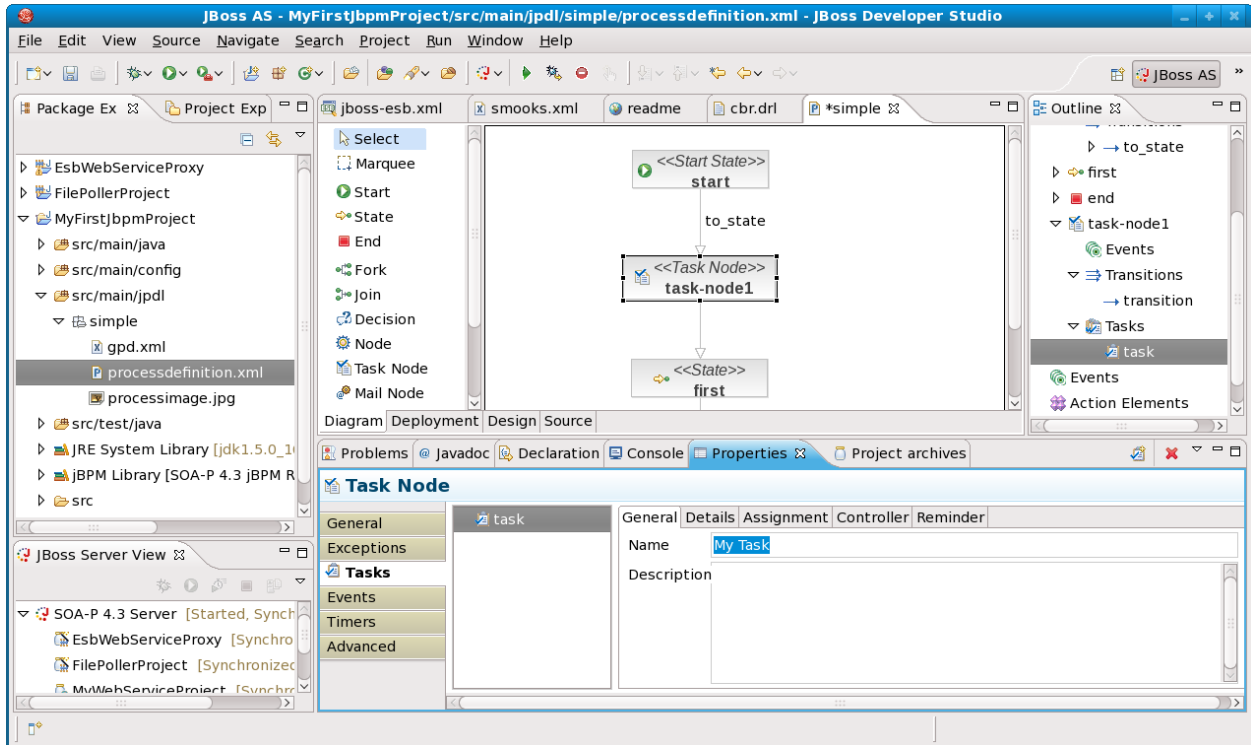


Now, right-click in the little window to the right of “Tasks” and select “New Task”.

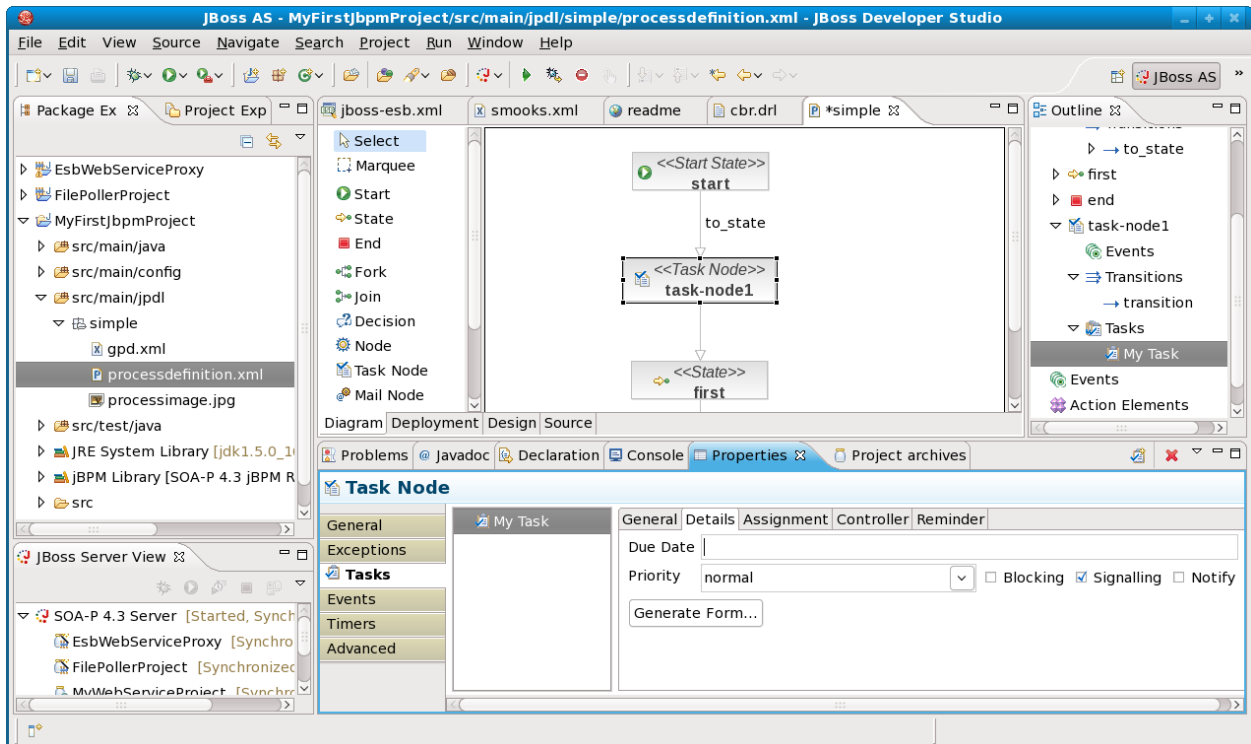


Give the task a name as shown below.

JBoss SOA Platform with JBoss Developer Studio Workshop

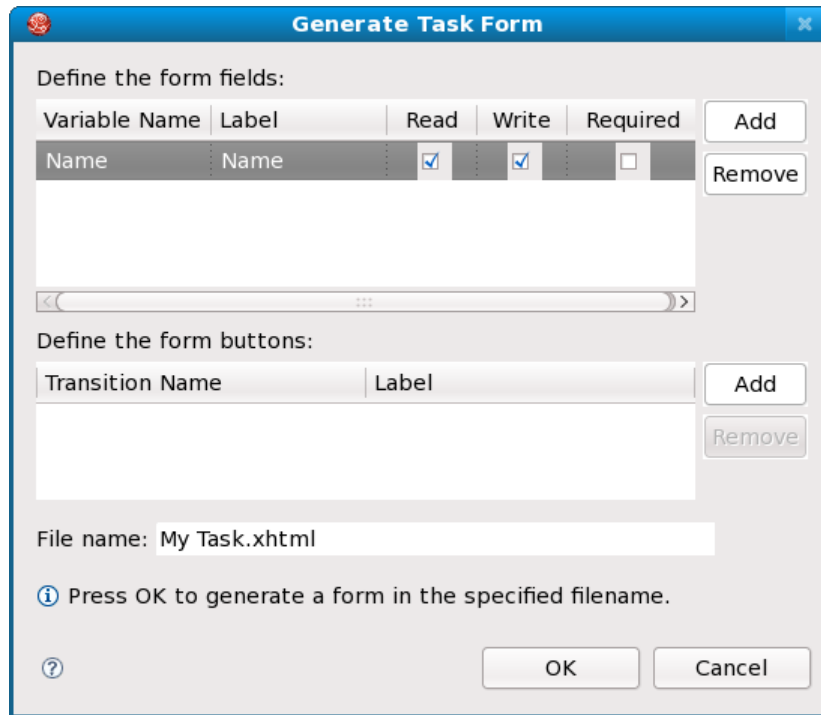


Now, click the “Details” tab.



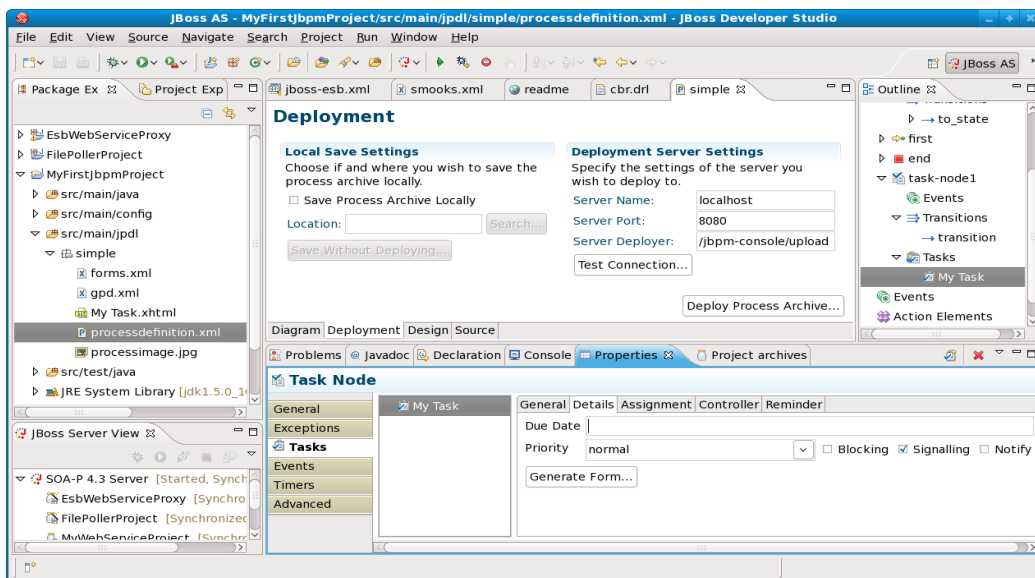
Here, we want to click “Generate Form...” to create a simple form for this task.

Fill out the form as shown below and be sure to press “Enter” after typing the variable name and label – otherwise, it won't accept the text you typed.



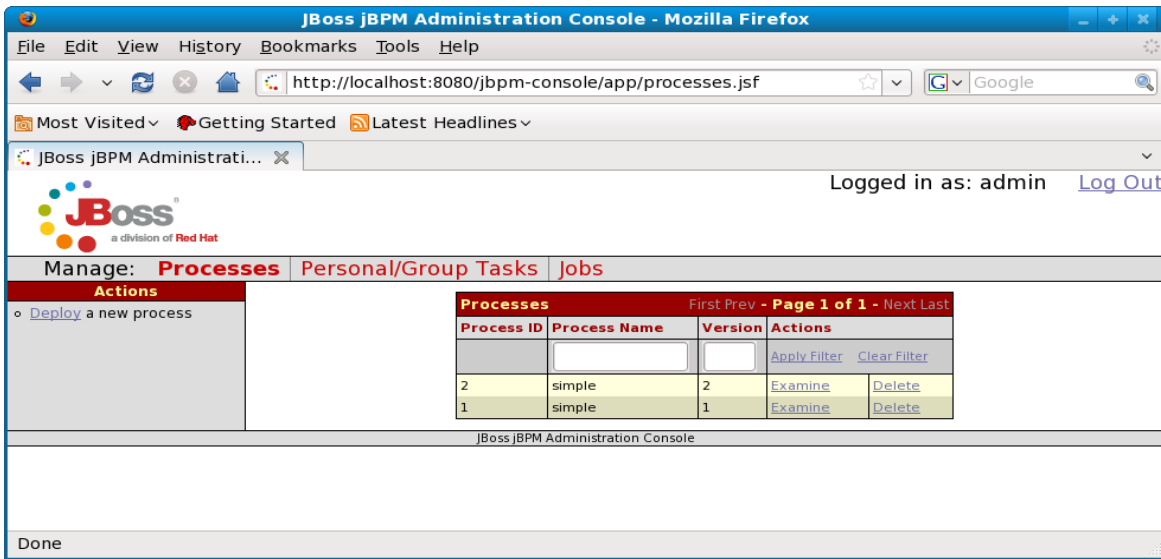
We could create transition buttons as well, but we're trying to keep this lab as simple as possible. Press “OK”.

That's all we have to do! Now make sure to do “File | Save All”. Then click the “Deployment” tab of the main process editor to see the view shown below.

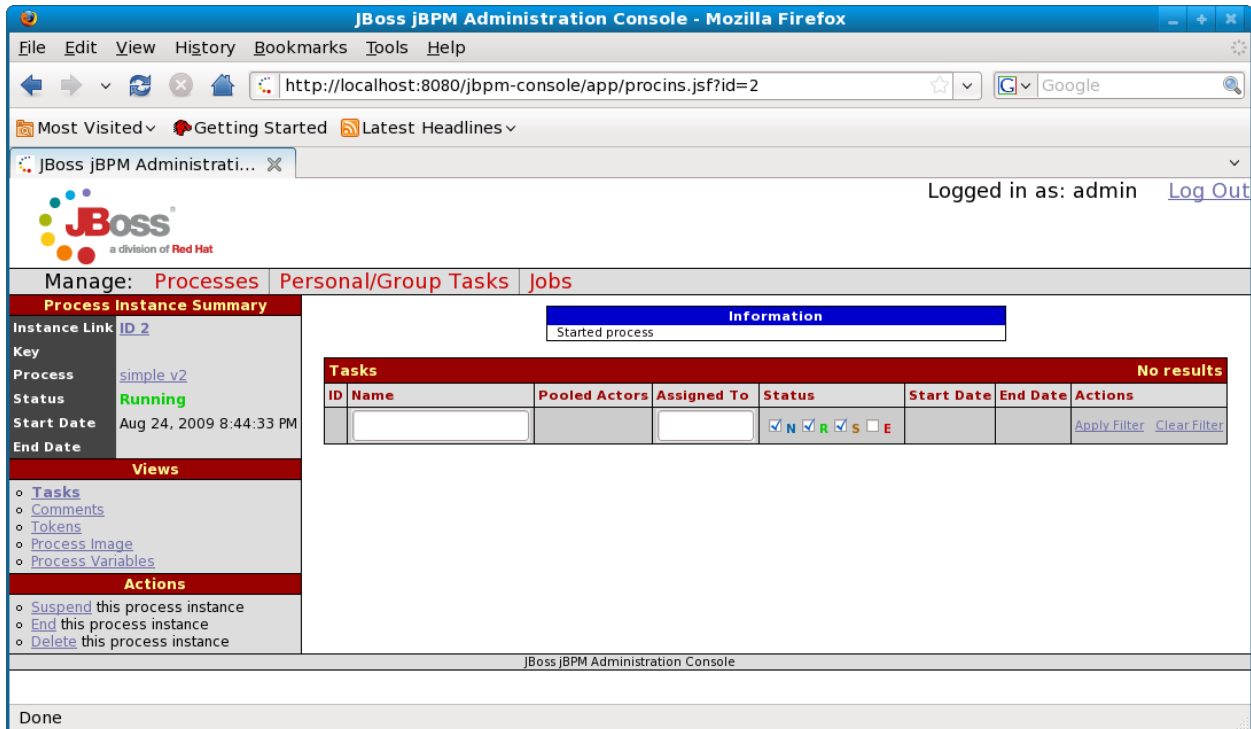


Now, click the “Deploy Process Archive” button and OK on the confirmation dialog.

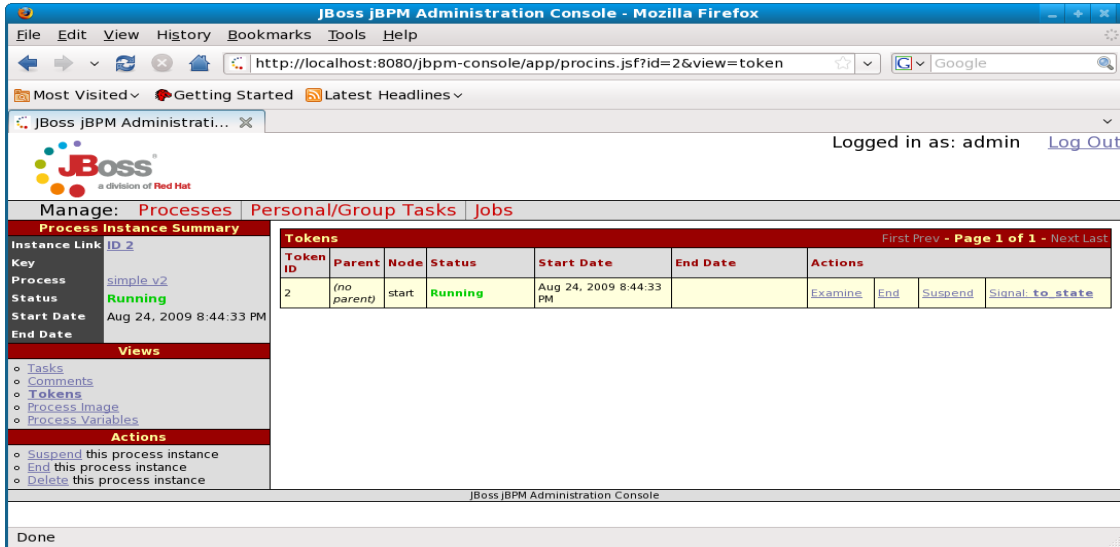
Now our process is deployed and we can see the new version by going to a <http://localhost:8080/jbpm-console> in a browser as shown below.



Let's click "Examine" on our version #2 of the "simple" process definition. You'll see that there are no processes, because we haven't created any process instances for this version of the process definition. So, click the "Start a new instance of this process" link.

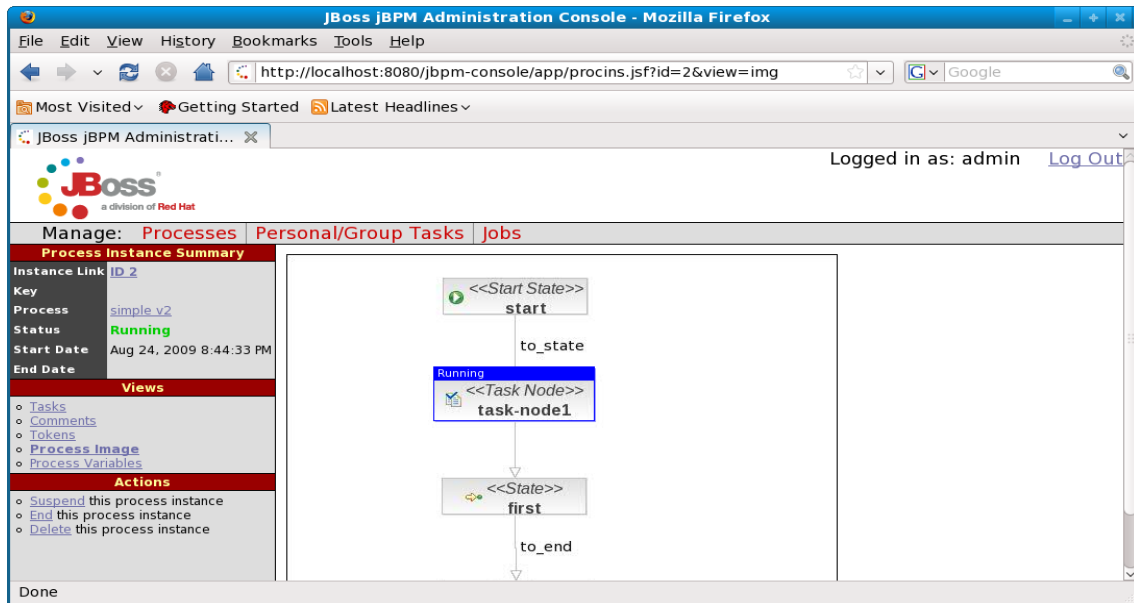


As shown above, there are no tasks for this new process instance. Why not? It's because the process is still in the "start" state and not in our task state yet. So, click on the "Tokens" link on the left which will show our Tokens (logical threads) for this process.

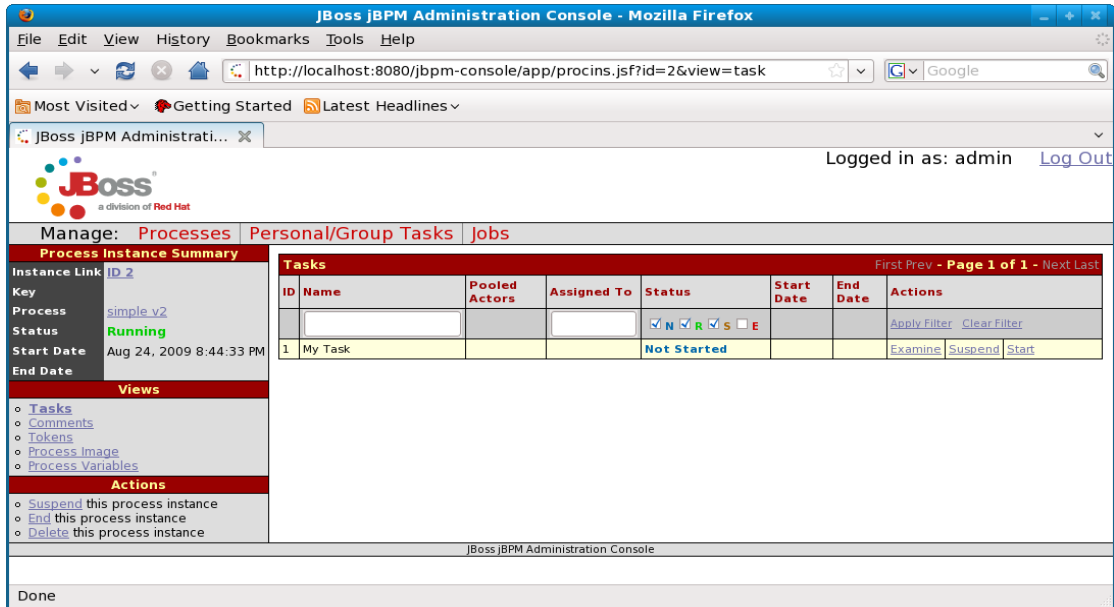


This token is in the “start” node. We can signal it to move to the next node by clicking “Signal to_state” link on the right.

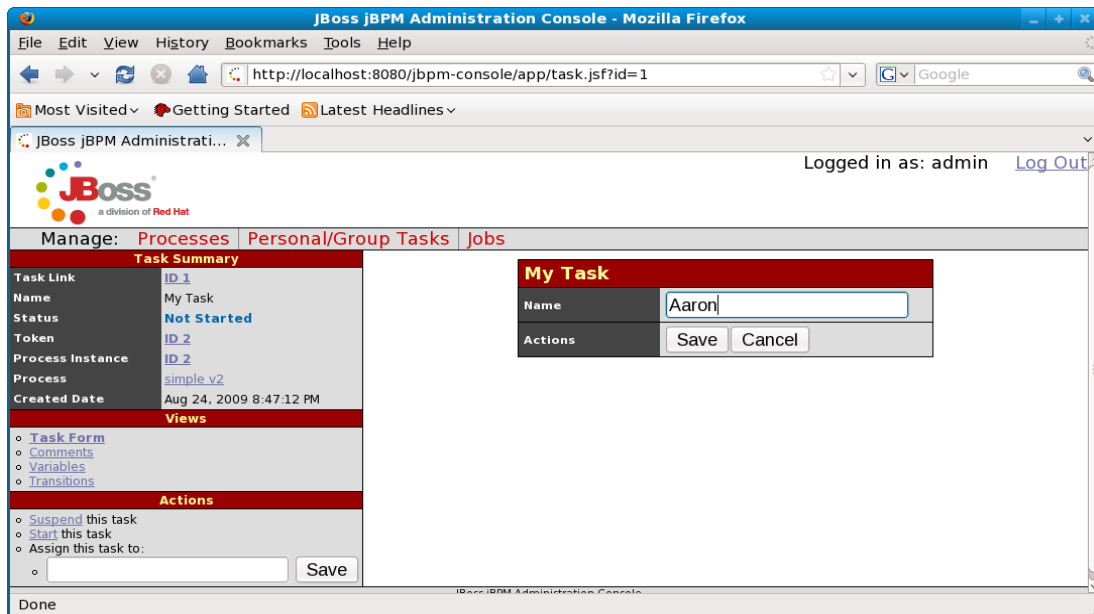
Now if we click on the “Process Image” link on the left, we'll see that we are in the task node as shown below.



We can also click the “Tasks” link on the left and we'll see that we have a task as shown below.

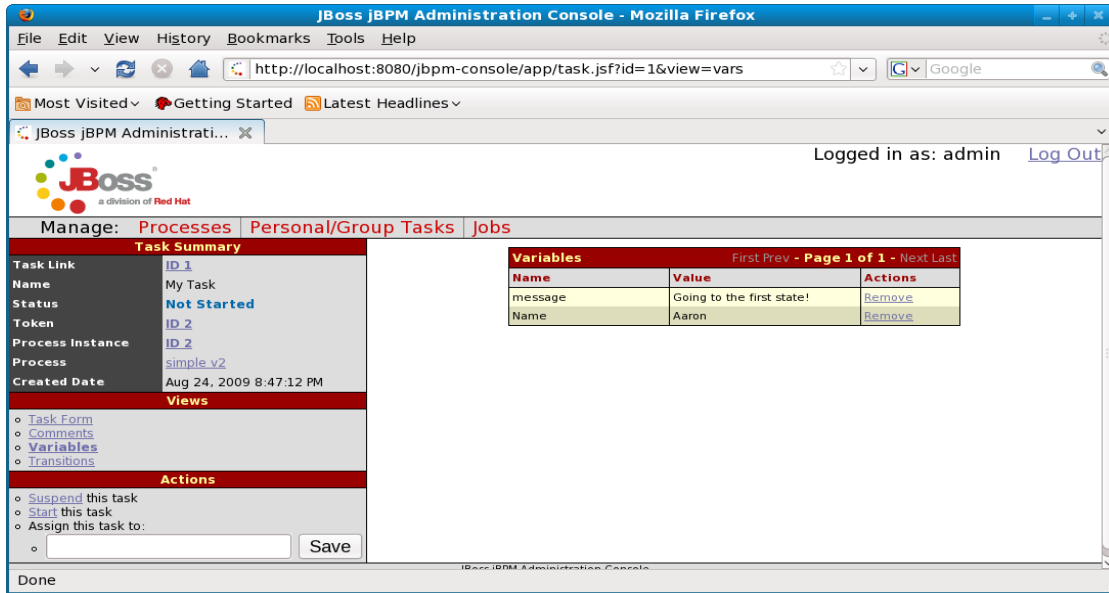


We can now click the “Examine” link to submit our task.

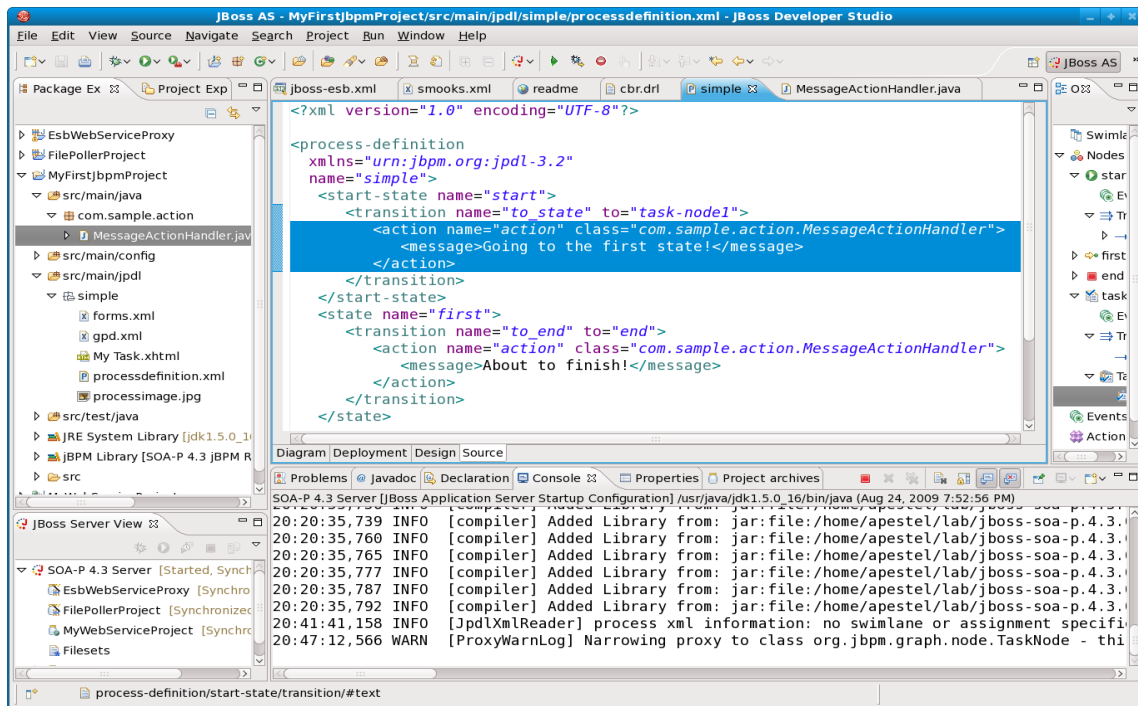


Fill in you name as shown above and click the “Save” button.

Now, click the “Variables” link on the left and you'll see two variables as shown below.



The “Name” variable came from our form. Where did the “message” variable come from? That will be left as an exercise for the reader. But for a hint, go back to JBDS and look at the source tab of the jBPM editor as shown below.



Congratulations, you've now created your first jBPM project with a user task! There is so much more that can be done with jBPM - invoking ESB services, getting invoked by ESB services, sending emails, creating custom actions, forks, joins, conditions, timers, and much, much more. Hopefully this lab has given you enough background that you will have a head start looking into these other capabilities.

Conclusion

We hope that this lab has been useful for you and increased your interest in JBoss SOA Platform. There is much functionality that we were not able to cover in this short workshop. Here are some areas for recommended reading:

<http://www.redhat.com/docs/soa-platform>

<http://www.jboss.com/soa-platform>

<http://www.jboss.org/soa-p-wiki>

Also, if you are interested in how JBoss is going to support BPEL in the future, you are encouraged to look at our jboss.org Riftsaw project that will be the basis for our BPEL platform to be released in early 2010.

<http://www.jboss.org/riftsaw>

Thank you for taking the time to work through this lab. Please support open source!