RiftSaw 2.1.0.CR1

User Guide

by Gary Brown, Kurt Stam, and Heiko Braun

Introduction

1.1. Overview

This is the User Guide for the RiftSaw BPEL process engine.

RiftSaw provides a JBoss AS integration for the Apache ODE BPEL engine. For detailed information on executing BPEL processes within Apache ODE, we would refer the reader to the Apache ODE website and documentation.

In addition to the ability to run the Apache ODE engine within JBoss AS, the RiftSaw project also provides a GWT based administration console, replaces the Axis2 based transport with JBossWS (which can be configured to use Apache CXF), and provides tighter integration with JBossESB.

Administration

2.1. Overview

This section describes the administration capabilities associated with RiftSaw.

2.2. BPM Console

2.2.1. Overview

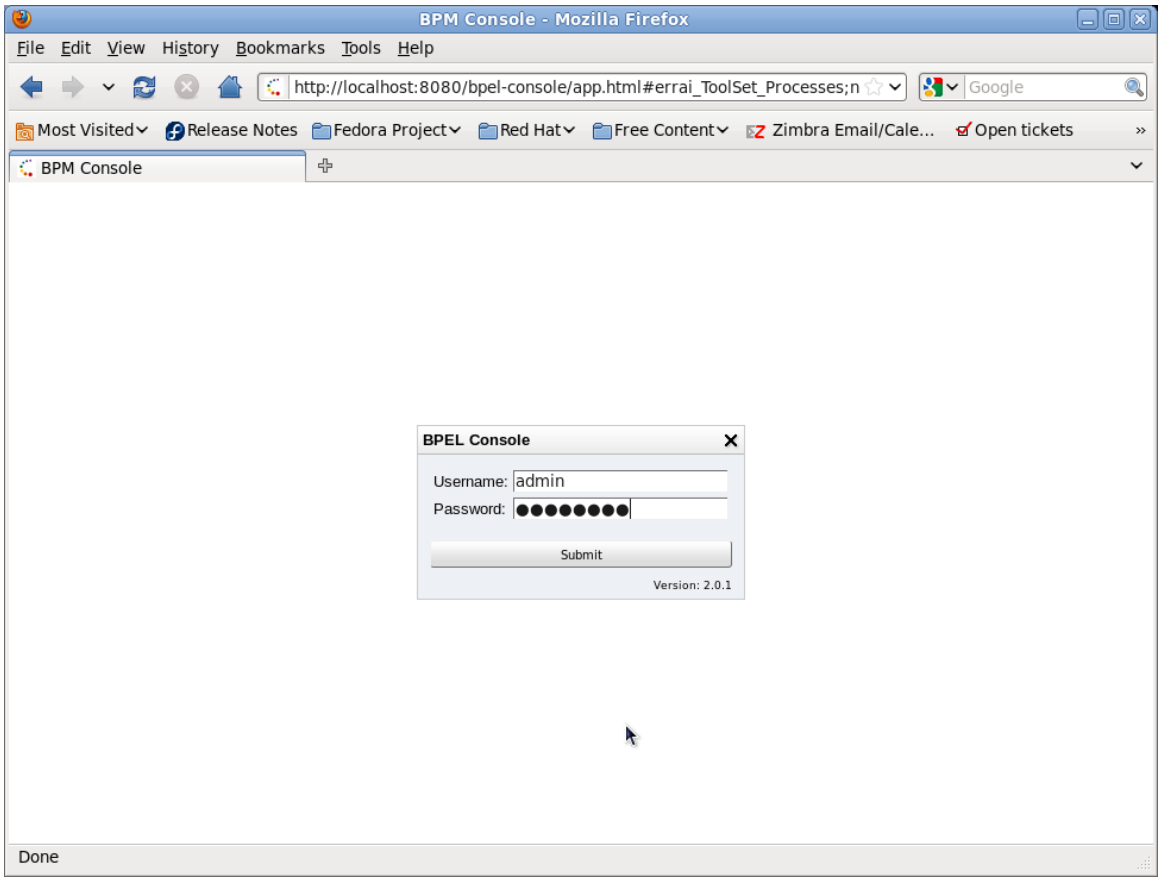
This section provides an overview of the BPEL Console. The console provides the ability to view:

- The process definitions deployed to the BPEL engine
- The process instances executing in the BPEL engine

2.2.2. Logging in

The BPEL console can be located using the URL: http://localhost:8080/bpel-console.

The first screen that is presented is the login screen:



The default username is *admin* with password *password*.

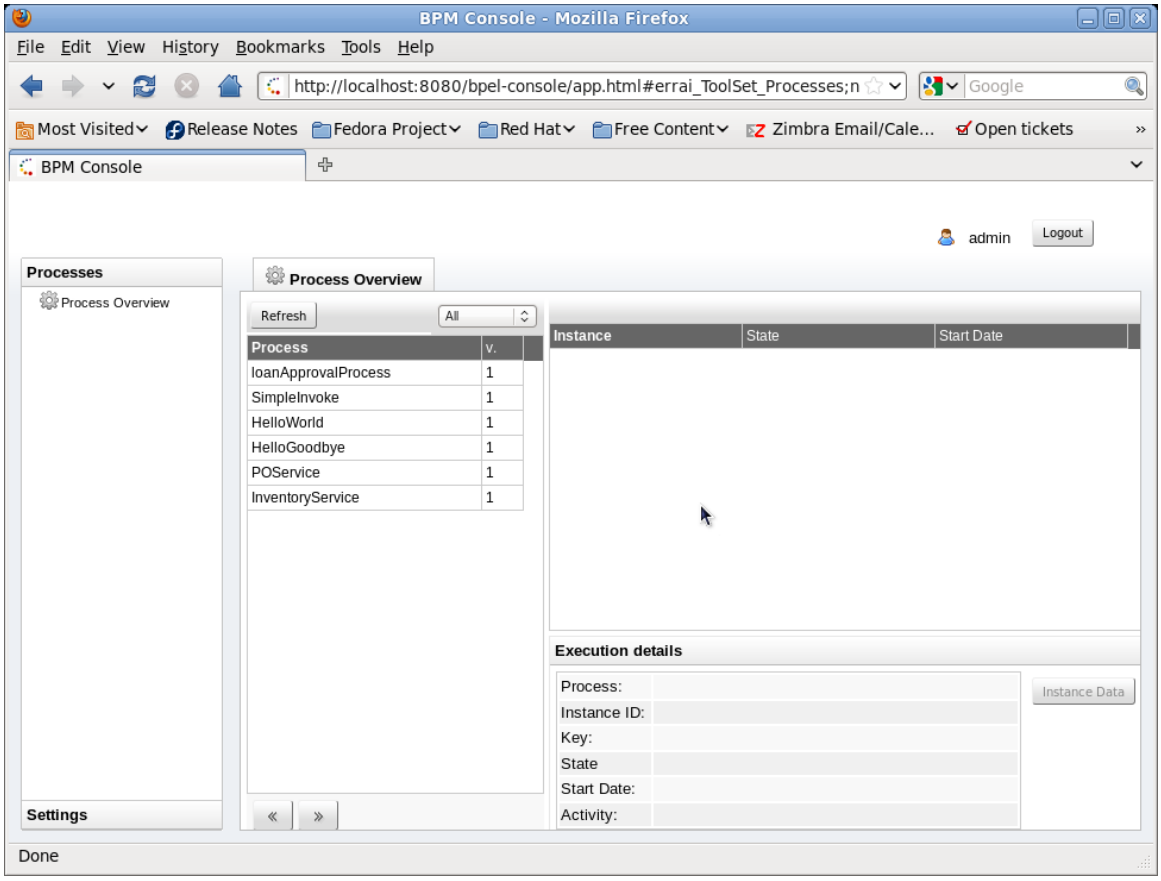
The Access Control mechanism used by the admin console is configured in the `$deployFolder/bpel-console/bpel-identity.sar/META-INF/jboss-service.xml`. The JAAS login module is initially set to use a property file based access mechanism, but can be replaced to use any appropriate alternative implementation.

The users for the default mechanism are configured in the property file `$deployFolder/bpel-console/bpel-identity.sar/bpel-users.properties`. The entries in this file represent *username=password*.

The user roles for the default mechanism are configured in the property file `$deployFolder/bpel-console/bpel-identity.sar/bpel-roles.properties`. The entries in this file represent *username=role*. The only role of interest currently is *administrator*.

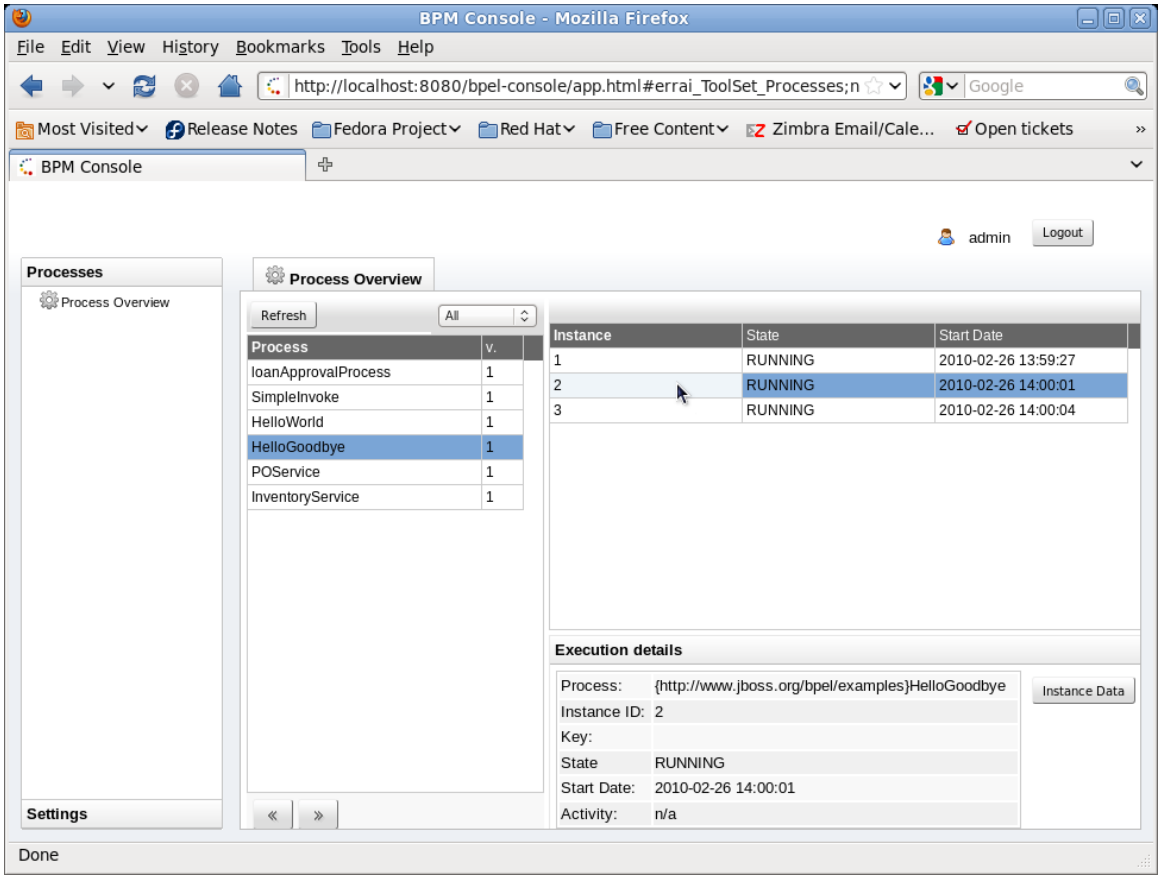
2.2.3. Deployed Process Definitions

Once logged in, the 'Process Overview' tab shows the currently deployed BPEL processes and their versions.

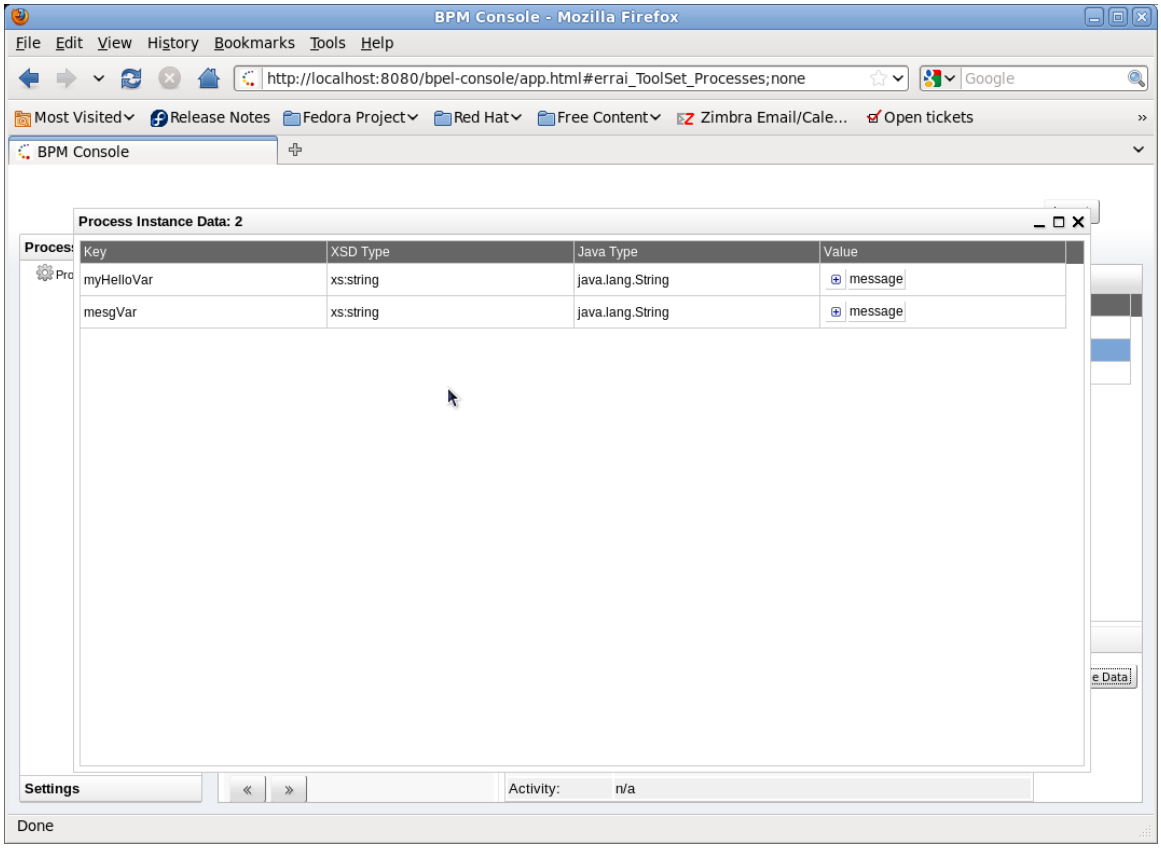


2.2.4. Process Instances

When a process definition is selected, the list of active process instances for that process definition (and version) will be displayed in the right hand panel.



When a process instance is selected, its details will be displayed in the lower *Execution Details* window. The *Instance Data* button will also become enabled, allowing further detail about the process to be displayed.


BPM Console - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:8080/bpel-console/app.html#errai_ToolSet_Processes;none
Most Visited Release Notes Fedora Project Red Hat Free Content Zimbra Email/Cale... Open tickets
BPM Console
Process Instance Data: 2
Key
XSD Type
Java Type
Value
myHelloVar
xs:string
java.lang.String
message
mesgVar
xs:string
java.lang.String
message
Settings
Activity:
n/a
Done

Deploying BPEL Processes

3.1. Overview

This section outlines the mechanisms that can be used to deploy a BPEL process to RiftSaw BPEL engine running within a JBoss AS server.

3.2. Direct deployment to JBossAS server

The direct deployment approach is demonstrated using an *Ant* script in each of the quickstart examples. For example,

```
<!-- Import the base Ant build script... -->
<property file="../../../install/deployment.properties" />

<property name="version" value="1" />

<property name="server.dir" value="${org.jboss.as.home}/server/${org.jboss.as.config}"/>
<property name="conf.dir" value="${server.dir}/conf"/>
<property name="deploy.dir" value="${server.dir}/deploy"/>
<property name="server.lib.dir" value="${server.dir}/lib"/>

<property name="sample.jar.name" value="${ant.project.name}-${version}.jar" />

<target name="deploy">
    <echo>Deploy ${ant.project.name}</echo>
    <jar basedir="bpel" destfile="${deploy.dir}/${sample.jar.name}" />
</target>

<target name="undeploy">
    <echo>Undeploy ${ant.project.name}</echo>
    <delete file="${deploy.dir}/${sample.jar.name}" />
</target>
```

This excerpt from the *Ant* build file for the *hello_world* quickstart example shows that deploying a RiftSaw BPEL process using *Ant* is very straightforward. The main points of interest are:

- It is necessary to identify the location of the JBoss AS server in which the BPEL process will be deployed. This is achieved in this example by referring to the `deployment.properties` file that has been configured in the RiftSaw distribution (install folder).
- If a versioned approach is being used, so that multiple versions of the same BPEL process may be deployed at one time, then the name of the archive (jar) containing the BPEL process (and associated artifacts) has a version number suffix. This would need to be manually incremented for each distinct version of the BPEL process being deployed.

Warning

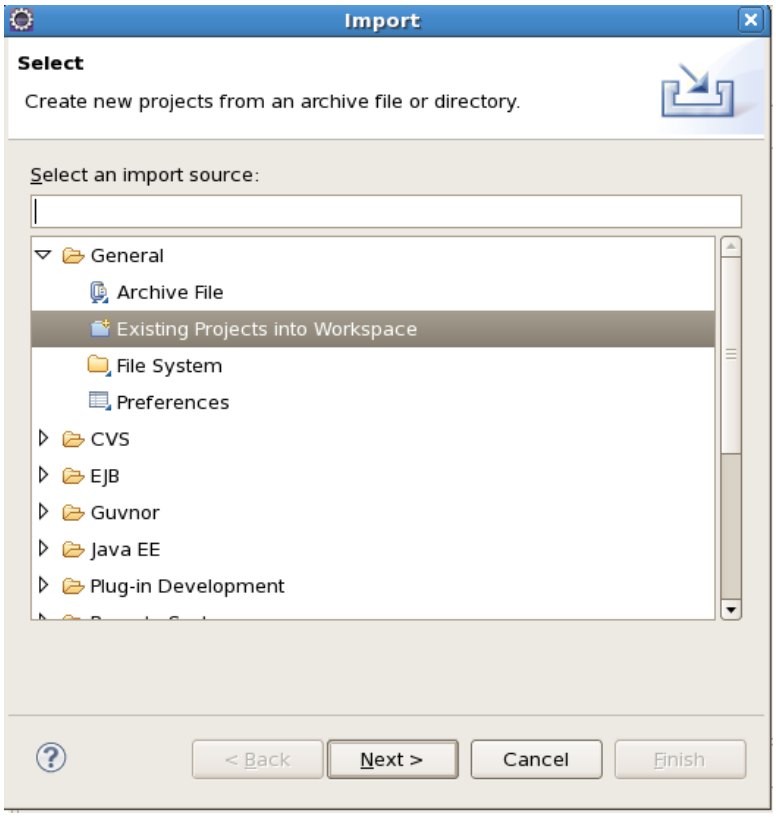
Currently the version must be specified as a single integer value. Non-numeric values, such as versions expressed in a major.minor.incremental (maven style), will result in an exception when deployed to the server.

- The next step is to define the *deploy* target, which will create the BPEL process archive, using the contents of the bpel sub-folder in this case, and store it within the JBoss AS server's `deploy` folder.
- The final step is to define the *undeploy* target, which simply removes the BPEL process archive from the JBoss AS server's `deploy` folder.

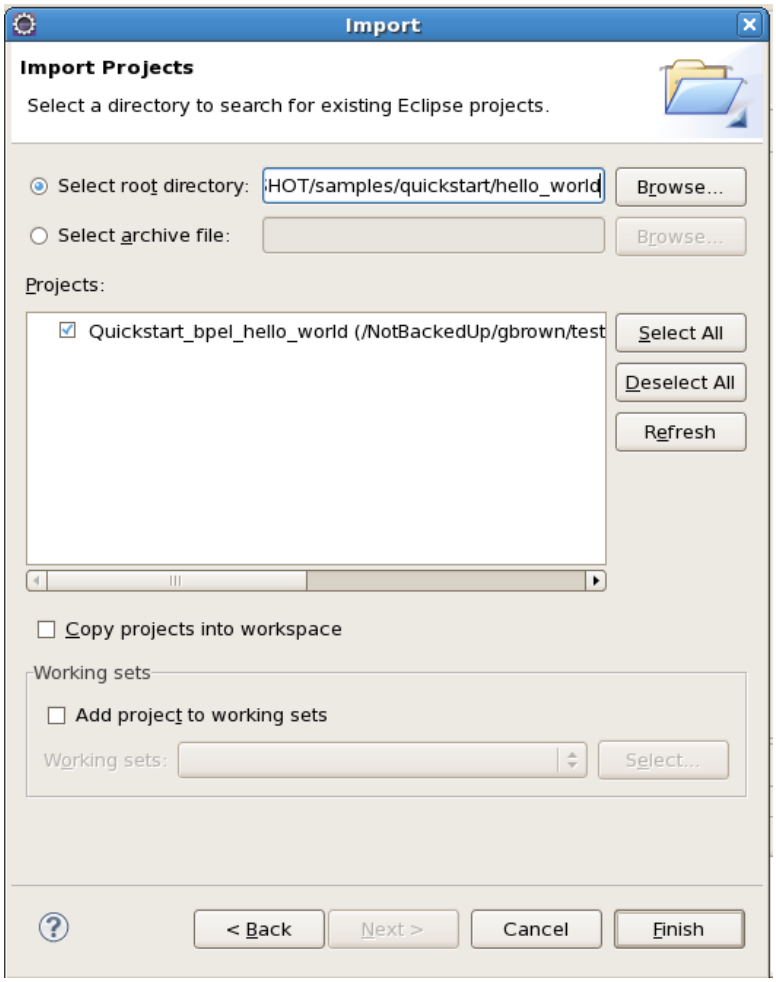
3.3. Eclipse based Deployment

This section will explain how to deploy an Eclipse BPEL project to the RiftSaw BPEL engine running in a JBossAS server.

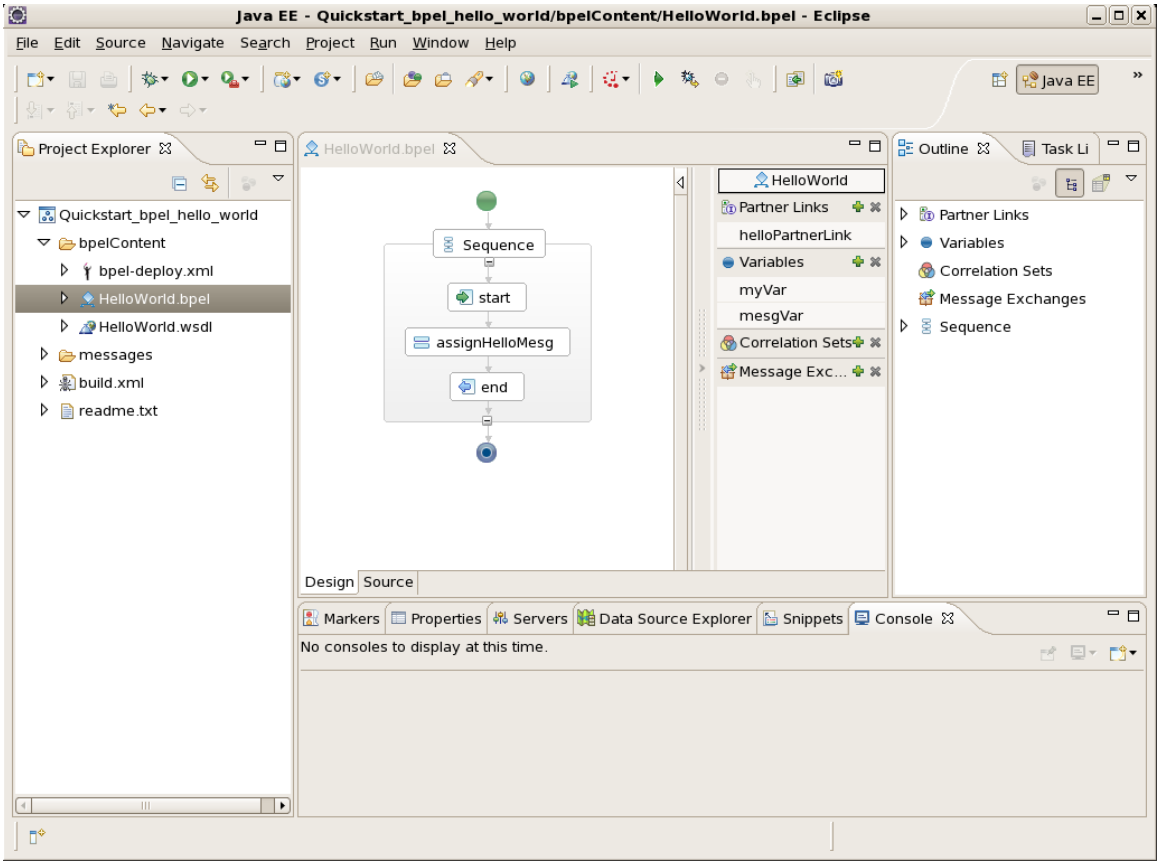
The first step is to create or import the Eclipse BPEL project. In this case we are going to import an existing project from the `${RiftSaw}/samples/quickstart/hello_world` folder. This can be achieved by selecting the *Import ...* menu item associated with the lefthand navigator panel in Eclipse, and then select the *General->Existing Projects into Workspace* entry and press the *Next* button.



Then press the *Browse* button and navigate to the `hello_world` quickstart folder. Once located, press the *Finish* button.

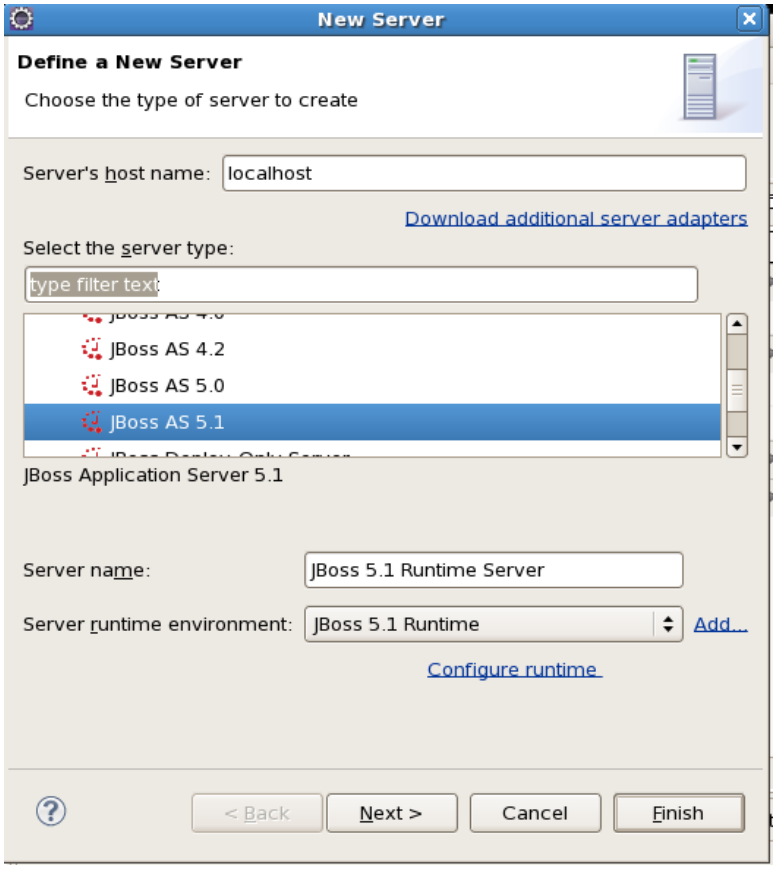


Once the project has been imported, you can inspect the contents, such as the BPEL process and WSDL description.



The next step is to create a server configuration for the JBoss AS environment in which the RiftSaw BPEL engine has previously been installed. From the Eclipse *Java EE* perspective, the *Server* tab should be visible in the lower region of the Eclipse window. If this view is not present, then go to the *Window->Show Views->Servers* menu item to open the view explicitly.

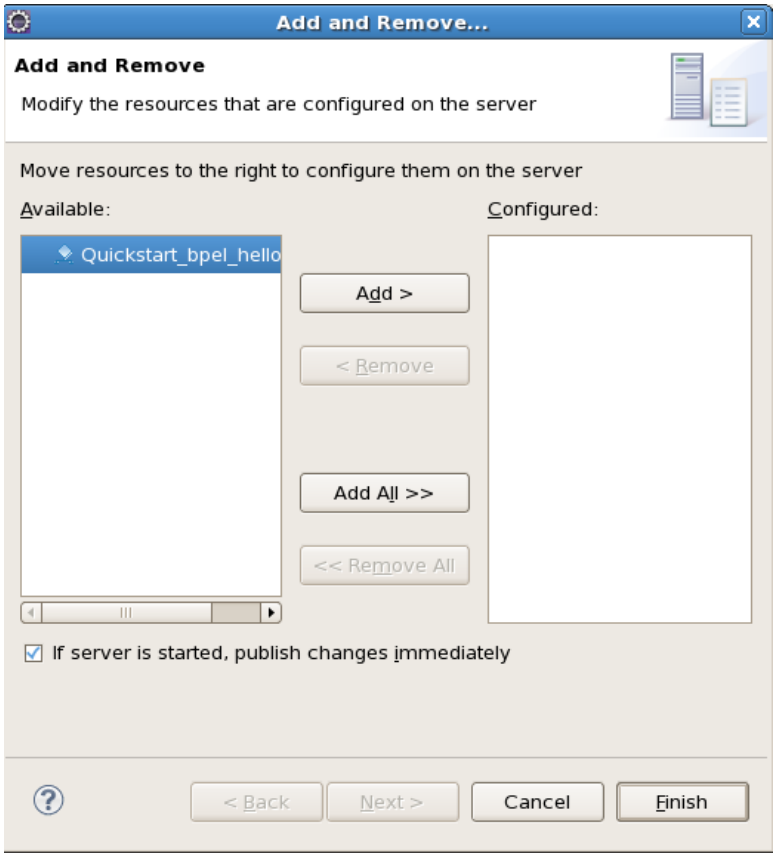
In the *Servers* view, right click and select the *New->Server* menu item.



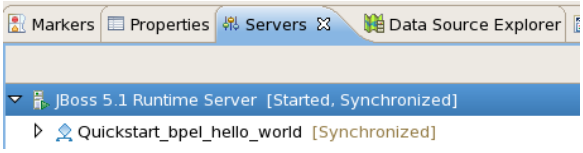
Select the appropriate JBoss AS version, and then press *Finish*.

Before being able to deploy an example, we should start the new server. This can be achieved by right clicking on the server in the *Servers* tab, and selecting the *Start* menu item. The output from the server will be displayed in the *Console* tab.

Once the server has been started, right click on the server entry again, and select the *Add and Remove ...* menu item.

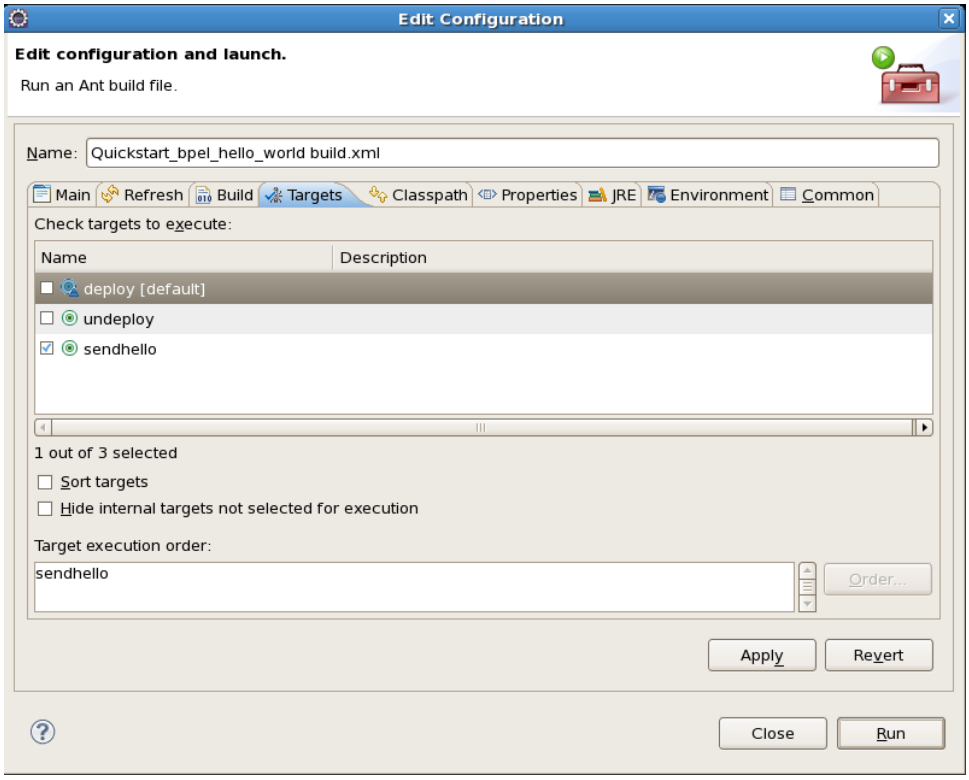


Select the *Quickstart_bpel_hello_world* project, press the *Add* button and the press the *Finish* button. This will cause the project to be deployed to the server.

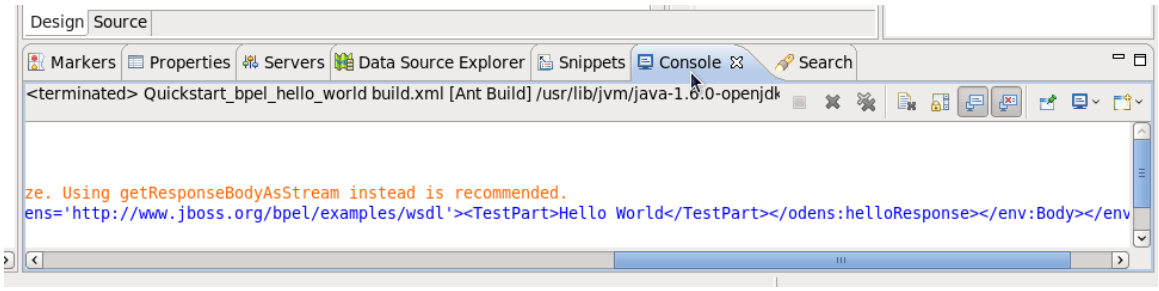


Once the project has been deployed, it will show up as an entry below the server in the *Servers* tab.

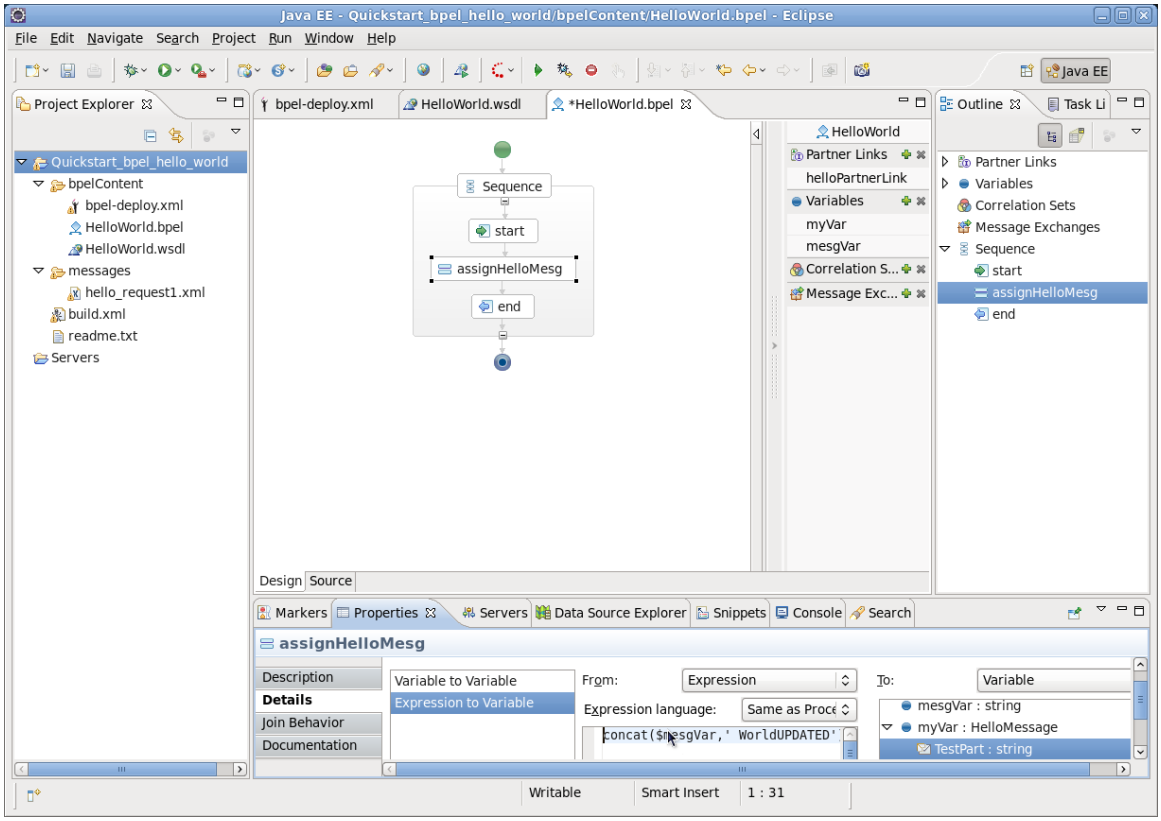
The final step is to test the deployed BPEL process. In this example, we can do this using the *ant* script provided with the quickstart sample. Right click on the `build.xml` file in the root folder of the project, and select the *Run As->Ant Build ...* menu item. NOTE: It is important to select the menu item with the *"..."*, as this provides a dialog window to enable you to select which *ant target* you wish to perform.



Deselect the *deploy* target, and select the *sendhello* target, before pressing the *Run* button. This was send a test 'hello' message to the server, and then display the response in the *Console* tab.

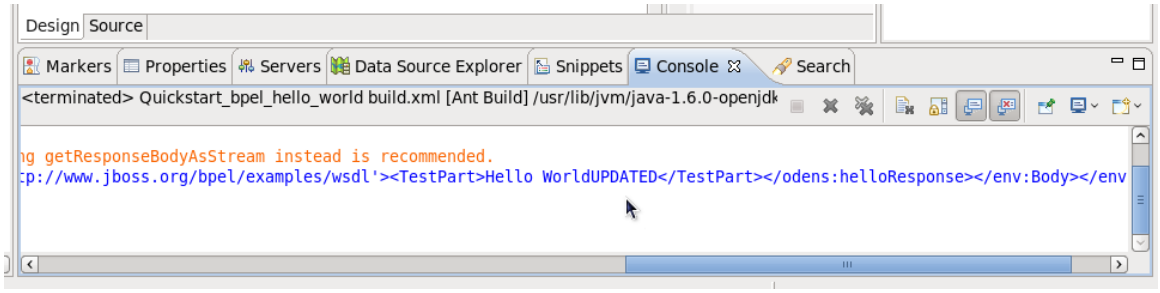


If you now want to update the BPEL process, select the *assignHelloMesg* node in the diagram, and select the *Properties* view. On the left of the view is a vertical list of tabs. Select the *Details* tab. Then select the "Expression to Variable" from the list, and update the *concat* function's second parameter - for example to add 'UPDATED' to the text.



Once the update has been saved, go to the *Server View* and select the *Full Publish* option from the menu associated with the *Quickstart_bpel_hello_world* project. This will cause the project to be re-deployed to the RiftSaw server.

The final step is to then re-run the 'sendhello' target within the `build.xml` file, to send a new request, and view the response. The response should now be modified according to the changes you made in the BPEL process.



If you expand the deployed project node, in the *Server View*, you will see that both of the deployed versions are shown. The older version is retained as there may still be BPEL process instances using that version of the process. If you right-click on each of the child nodes, you will see that it is also possible to undeploy the specific versions. However, if you explicitly undeploy a version, then any remaining active process instances for that version will be terminated.

You can then use the menu associated with the project, contained in the *Server View*, to undeploy the project (using the *Add and Remove* ... menu item) and finally use the menu associated with the server itself to *Stop* the server.

3.4. Changing Endpoint Configuration Properties

Apache ODE provides the means to customise certain properties, associated with a BPEL endpoint, by specifying the properties in a file with an extension of `.endpoint`.

For information on the properties that can be specified in this file, please see the Apache ODE documentation, located at: http://ode.apache.org/endpoint-configuration.html.

Note

RiftSaw currently only supports the following properties: *mex.timeout*

This section explains how to deploy these `.endpoint` files as part of a RiftSaw deployment.

Apache ODE supports two locations for finding these `.endpoint` files. A 'global' configuration folder, which by default is `ode/WEB-INF/conf`, and a process deployment specific location, which is `ode/WEB-INF/processes/$your_process`. Properties associated with the 'global' configuration override any property values provided in the process specific location.

RiftSaw currently does not support a 'global' configuration location, so it will only obtain the configuration files from the deployed BPEL bundle. More specifically, from the root location within the BPEL deployment unit, along side the BPEL deployment descriptor.

So, for example, if you place a file called `test.endpoint` in the `${RiftSaw}/samples/quickstart/hello_world/bpelContent` folder, with the following content:

```
# 3 minutes
mex.timeout=180000
```

then once deployed, the helloworld example could wait up to a maximum of 3 minutes to respond. To test this out, edit the `hello_world.bpel` and insert a *wait* activity before the response - similar to the following:

```
<wait>
    <for>'PT150S'</for>
</wait>
```

This will wait 2 minutes 30 seconds before responding, which is 30 seconds more than the default timeout, but still within the new timeout period specified within the `test.endpoint` file. If you then wish to try forcing the timeout to occur, simply increase the wait duration to 3 minutes 30 seconds, and resubmit the test message using the *ant sendhello* command.

Web Service Configuration

4.1. Overview

This section outlines the mechanisms that are available for configuring the web service stack used in providing the web service for a BPEL process, as well as invoking external web services from a BPEL process.

4.2. Configuring a JAX-WS Handler

JAX-WS is a standard Java API for client and server support of web services. The JAX-WS handler mechanism can be used by a client or server (i.e. the web service) to invoke a user specified class whenever a message (or fault) is sent or received. The handler is therefore installed into the message pipeline, and can manipulate the message header or body as required.

The handlers are usually installed either programmatically, or through a *HandlerChain* annotation on the Java interface representing the Web Service. However, in the case of a BPEL process deployed to RiftSaw, the JAX-WS service (representing the web service associated with the BPEL process) is dynamically created on deployment.

Therefore to associate the configuration of a JAX-WS handler chain with the Web Service dynamically created to support the BPEL process, the user must place a file called `jws_handler.xml` alongside the BPEL process definition and deployment descriptor.

The following provides an example of the XML configuration associated with the `jws_handler.xml` file. This particular example is used by the *service_handler* quickstart sample.

```
<handler-chains xmlns="http://java.sun.com/xml/ns/javaee"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://java.sun.com/xml/ns/javaee">
    <handler-chain>
        <handler>
            <handler-name>JAXWSHandler</handler-name>
            <handler-class>org.jboss.soa.bpel.examples.jaxws.JAXWSHandler</handler-class>
            <init-param>
                <param-name>TestParam</param-name>
                <param-value>TestValue</param-value>
            </init-param>
        </handler>
    </handler-chain>
</handler-chains>
```

The format of this file is the standard JAX-WS handler chain configuration. One or more handler elements can be specified, with each handler defining a name and class. The handler configuration can optionally provide initialization parameters that are passed to the *init* method on the handler implementation.

Note

This mechanism only installs JAX-WS handlers on the 'provider' web service. It is not currently possible to configure JAX-WS handlers for the client endpoints that invoke external web services from a BPEL process.

An example of this mechanism can be found in the *service_handler* quickstart sample.

4.3. Apache CXF Configuration

RiftSaw integrates with JBossWS, using the JAX-WS standard API, to support the following web service stacks: JBossWS native, Metro and Apache CXF. This section explains how RiftSaw deployed BPEL processes can include additional configuration specifically applicable to the Apache CXF web service stack - and is therefore only relevant if the JBossAS application server has been configured to use this stack. See the *Getting Started Guide* for information on how to switch to the Apache CXF stack when installing RiftSaw.

This section will explain how web service endpoints, whether server (i.e. representing the BPEL process) or client (i.e. being used to invoke external web services), are configured using the Apache CXF configuration format. It will also discuss reasons why you may wish to do this additional CXF specific configuration. However, for further information on how to configure CXF, and the features that it offers, the reader is referred to the Apache CXF website http://cxf.apache.org.

4.3.1. Configuring the Server endpoint

To create a CXF configuration that will be used by the RiftSaw web service provider (i.e. the server), it is simply a case of placing a file called `jbosssws-cxf.xml` into the root folder of the BPEL deployment (along side the deployment descriptor).

This is the same filename as used by jbossws-cxf, when deploying a web service based on the use of JAXWS annotations. An example of the file content is:

```
<beans
  xmlns='http://www.springframework.org/schema/beans'
  xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
  xmlns:beans='http://www.springframework.org/schema/beans'
  xmlns:jaxws='http://cxf.apache.org/jaxws'
  xsi:schemaLocation='http://cxf.apache.org/core
    http://cxf.apache.org/schemas/core.xsd
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
    http://cxf.apache.org/jaxws
    http://cxf.apache.org/schemas/jaxws.xsd'>

  <bean id="UsernameTokenSign_Request"
            class="org.apache.cxf.ws.security.wss4j.WSS4JInInterceptor">
```

```
    <constructor-arg>
      <map>
        <entry key="action" value="UsernameToken Timestamp Signature"/>
        <entry key="passwordType" value="PasswordDigest"/>
        <entry key="user" value="serverx509v1"/>
        <entry key="passwordCallbackClass"
                value="org.jboss.test.ws.jaxws.samples.wsse.ServerUsernamePasswordCallback"/>
        <entry key="signaturePropFile" value="etc/Server_SignVerf.properties"/>
        <entry key="signatureKeyIdentifier" value="DirectReference"/>
      </map>
    </constructor-arg>
  </bean>

  <bean id="UsernameTokenSign_Response"
           class="org.apache.cxf.ws.security.wss4j.WSS4JOutInterceptor">
    <constructor-arg>
      <map>
        <entry key="action" value="UsernameToken Timestamp Signature"/>
        <entry key="passwordType" value="PasswordText"/>
        <entry key="user" value="serverx509v1"/>
        <entry key="passwordCallbackClass"
                value="org.jboss.test.ws.jaxws.samples.wsse.ServerUsernamePasswordCallback"/>
        <entry key="signaturePropFile" value="etc/Server_Decrypt.properties"/>
        <entry key="signatureKeyIdentifier" value="DirectReference"/>
        <entry key="signatureParts"

   value="{Element}{http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-
1.0.xsd}Timestamp;{Element}{http://schemas.xmlsoap.org/soap/envelope/}Body"/>
      </map>
    </constructor-arg>
  </bean>

  <jaxws:endpoint
    id='SecureHelloWorldWS'
    address='http://@jboss.bind.address@:8080/Quickstart_bpel_secure_serviceWS'
    implementor='@provider@'>
    <jaxws:inInterceptors>
        <ref bean="UsernameTokenSign_Request"/>
        <bean class="org.apache.cxf.binding.soap.saaj.SAAJInInterceptor"/>
    </jaxws:inInterceptors>
    <jaxws:outInterceptors>
        <ref bean="UsernameTokenSign_Response"/>
        <bean class="org.apache.cxf.binding.soap.saaj.SAAJOutInterceptor"/>
    </jaxws:outInterceptors>
  </jaxws:endpoint>


</beans>
```

This example configures the web service to use username token and digital signature authentication.

Note

The *jaxws:endpoint* element has an attribute called *implementor* that defines the Java class implementing the JAXWS service. RiftSaw dynamically creates this class, and therefore

it is important that the attribute is set to the value *@provider@* to enable the dynamically created Java class to be correctly configured during deployment.

4.3.2. Configuring the Client endpoint

When configuring client endpoints, representing web services invoked by a BPEL process, the configuration is currently separated into different files on a per port basis - similar to the approach used by the Axis2 ODE integration.

The file name is of the form `jbossws-cxf-{portname_local_part}.xml`, where the *portname_local_part* represents the local part of the portname of the web service being invoked. For example, if the WSDL for the invoked web service is:

```
<definitions name='SecureHelloWorldWSService'
        targetNamespace='http://secure_invoke/helloworld' .... >
    <portType name='SecureHelloWorld'>
        ...
    </portType>
    <service name='SecureHelloWorldWSService'>
       <port name='SecureHelloWorldPort' ... >
           ...
       </port>
    </service>
</definitions>
```

then the CXF configuration file would be `jbossws-cxf-SecureHelloWorldPort.xml`.

The CXF configuration information within this file is associated with the CXF bus. For example:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:cxf="http://cxf.apache.org/core"
       xmlns:wsa="http://cxf.apache.org/ws/addressing"
       xmlns:http="http://cxf.apache.org/transports/http/configuration"
       xmlns:wsrm-policy="http://schemas.xmlsoap.org/ws/2005/02/rm/policy"
       xmlns:wsrm-mgr="http://cxf.apache.org/ws/rm/manager"
       xmlns:beans='http://www.springframework.org/schema/beans'
       xmlns:jaxws='http://cxf.apache.org/jaxws'
       xmlns:ns1='http://secure_invoke/helloworld'
       xsi:schemaLocation="
       http://cxf.apache.org/core http://cxf.apache.org/schemas/core.xsd
           http://cxf.apache.org/transports/http/configuration  http://cxf.apache.org/schemas/
configuration/http-conf.xsd
        http://schemas.xmlsoap.org/ws/2005/02/rm/policy http://schemas.xmlsoap.org/ws/2005/02/
rm/wsrm-policy.xsd
         http://cxf.apache.org/ws/rm/manager http://cxf.apache.org/schemas/configuration/wsrm-
manager.xsd
```

```
      http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/
spring-beans.xsd
    http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd">

  <bean id="UsernameTokenSign_Request"
          class="org.apache.cxf.ws.security.wss4j.WSS4JOutInterceptor" >
    <constructor-arg>
      <map>
        <entry key="action" value="UsernameToken Timestamp Signature"/>
        <entry key="passwordType" value="PasswordDigest"/>
        <entry key="user" value="clientx509v1"/>
        <entry key="passwordCallbackClass"
               value="org.jboss.test.ws.jaxws.samples.wsse.ClientUsernamePasswordCallback"/>
        <entry key="signaturePropFile" value="etc/Client_Sign.properties"/>
        <entry key="signatureKeyIdentifier" value="DirectReference"/>
        <entry key="signatureParts"

   value="{Element}{http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-
1.0.xsd}Timestamp;{Element}{http://schemas.xmlsoap.org/soap/envelope/}Body"/>
      </map>
    </constructor-arg>
  </bean>

  <bean id="UsernameTokenSign_Response"
          class="org.apache.cxf.ws.security.wss4j.WSS4JInInterceptor" >
    <constructor-arg>
      <map>
        <entry key="action" value="UsernameToken Timestamp Signature"/>
        <entry key="passwordType" value="PasswordText"/>
        <entry key="user" value="serverx509v1"/>
        <entry key="passwordCallbackClass"
               value="org.jboss.test.ws.jaxws.samples.wsse.ClientUsernamePasswordCallback"/>
        <entry key="signaturePropFile" value="etc/Client_Encrypt.properties"/>
        <entry key="signatureKeyIdentifier" value="DirectReference"/>
      </map>
    </constructor-arg>
  </bean>

  <cxf:bus>
    <cxf:outInterceptors>
        <ref bean="UsernameTokenSign_Request"/>
        <bean class="org.apache.cxf.binding.soap.saaj.SAAJOutInterceptor"/>
    </cxf:outInterceptors>
    <cxf:inInterceptors>
        <ref bean="UsernameTokenSign_Response"/>
        <bean class="org.apache.cxf.binding.soap.saaj.SAAJInInterceptor"/>
    </cxf:inInterceptors>
  </cxf:bus>

</beans>
```

This example configures the web service client to use username token and digital signature authentication.

UDDI Integration

5.1. Overview

The integration of a UDDI client into the RiftSaw runtime codebase allows for the auto-registration of BPEL services to an UDDI registry upon deployment of the service. The registration process uses the jUDDI-3 client libraries which are capable of communicating to any UDDI v3 complaint registry.

Upon deployment both the Service and its BindingTemplate (EndPoint information) are registered, and a partnerLinkChannel is created for each partnerLink. UDDI lookup will obtain the WSDL Endpoint from the UDDI and access this URL to obtain the WSDL straight from the Service. Upon undeployment the BindingTemplate is removed from the UDDI Registry.

5.2. UDDI config properties

By default RiftSaw uses the jUDDI client libraries of JBossESB/SOA-P, and the client configuration is found in the deploy/jbossesb.saw/esb.juddi.client.xml. Both the name of the ClerkManager and the Clerk itself are specified in the *bpel.properties* file.

Table 5.1. The UDDI related properties in the bpel.properties

attribute	type (default)	description
bpel.uddi.registration	boolean (true)	If set to 'false', the UDDI integration is turned off. The RiftSaw installation process sets this value to 'true' only if the jbossesb-registry.sar is detected containing a jUDDI v3 registry. In all other case it is defaulted to false.
bpel.webservice.secure	boolean (false)	The UDDI Registration process registers an WSDL AccessPoint in the BindingTemplate for the BPEL Service it is registering. The BPEL server exposes the service WSDL Endpoint on the WS stack (Currently we support JBossWS and CXF), if your WS stack is configured to a use a secure (https) protocol, then you need to switch this setting to 'true'. Note that this setting is used during the registration process only.

attribute	type (default)	description
bpel.uddi.client.impl	String (org.jboss.soa.bpel.uddi.UDDIRegistrationImpl)	Name of the a class that implements the org.jboss.soa.bpel.runtime.engine.ode.UDDIRegistratio interface. The RiftSaw installation process set this value to org.jboss.soa.bpel.uddi300.UDDI300RegistrationImpl if JBossESB-4.7 is detected. JBossESB-4.7 uses the jUDDI-3.0.0 client libary, while SOA-P-5.0.0 and future version of JBossESB will use jUDDI-3.0.1 or higher and for those it is set to org.jboss.soa.bpel.uddi.UDDIRegistrationImpl.
bpel.uddi.clerk.config	String (not used by default)	Defines the path to the bpel.uddi.client.xml config file. This can be left commented out if you want to use the jbossesb.sar/esb.uddi.client.xml. However in that case a bpel.uddi.clerk.manager needs to be defined.
bpel.uddi.clerk.manager	String (esb-registry)	Defines the ClerkManager name that will be used if the bpel.uddi.clerk.config is left commented out. This value should correspond to the name of the manager in the esb.juddi.client.xml. For JBossESB-4.7 this is 'test-manager', while for SOA-P-5.0.0 and newer version of JBossESB it is 'esb-registry'. Note that if the bpel.uddi.clerk.config is defined, the setting of the bpel.uddi.clerk.manager is ignored.
bpel.uddi.clerk	String (BPEL_clerk)	Defines the Clerk name that will be used. This value should correspond to the name of the

attribute	type (default)	description
		clerk in the esb.juddi.client.xml. By default this is set to 'BPEL_clerk'.
bpel.uddi.lookup	boolean (true)	If set to true, the creating process of the partner channel will do a lookup by serviceName in the UDDI, and a WSDL Endpoint is retrieved. This WSDL Endpoint is then used to obtain the WSDL. This process makes it easier to move Endpoints around within your deployment, without having to update the partnerlink WSDL files in your bpel deployments. Note that an it is still a requirement to deploy the initial partnerlink WSDL file for each partnerLink.

5.3. Default configurations

When RiftSaw is deployed to a JBossAS-5.1.0, jUDDI v3 is not installed, and therefore the UDDI integration is turned off (bpel.uddi.registration=false).

When RiftSaw is deployed to JBossAS-5.1.0 + JBossESB-4.7, the UDDI integration is turned on. By default we use the jUDDI client library which ships with the JBossESB, which is configured in the `jbossesb.sar/esb.uddi.client.xml`, with manager name 'test-manager'. The *bpel.uddi.client.impl* is set to *org.jboss.soa.bpel.uddi300.UDDI300RegistrationImpl*.

When RiftSaw is deployed to SOA-P-5.0.0 (or JBossESB > 4.7) UDDI integration is turned on and the *bpel.uddi.client.impl* is set to *org.jboss.soa.bpel.uddi.UDDIRegistrationImpl*. Again the `jbossesb.sar/esb.uddi.client.xml` is used, with manager name 'esb.registry'.

5.4. Other UDDI v3 Registries

Other UDDI v3 compliant registries can be used, however the UDDIv3 spec only requires communication using the UDDI WebServices. To set up SOAP based communication specify the JAXWS-Transport. At this point it makes sense to no longer use the the esb.uddi.client.xml, but rather use your own *bpel.uddi.client.xml*. For more details please see the jUDDI v3 documentation.

5.5. UDDI Registry Entities and UDDI Seed Data

In the `esb.uddi.client.xml` a few properties are defined that are used by the Clerk at registration time. These settings of these values can be customized, however they must correspond

to the UDDI seed data specified for the jbossesb publisher, in the `jbossesb-registry.sar/juddi_custom_install_data`. So you will need to change it there as well.

The clerk is configured to use the jbossesb publisher and the *keyDomain* is set to "esb.jboss.org".

The *businessKey* is set to "redhat-jboss".

The *serviceDescription* is set to "BPEL Service deployed by Riftsaw".

The *bindingDescription* is set to "BPEL Endpoint deployed by Riftsaw".

Note that in SOA-P-5 the `jbossesb-registry.sar/esb.uddi.xml` contains a property *juddi.seed.always* which is set to "true". This means that that it is always trying to load the root seed data on startup of the server. It is recommended to turn this value to "false" once you are content with the UDDI Seed Data.

Warning

When running on JBossESB-4.7, the root seed data will only be installed if the UDDI tables are still empty. This means that you need to make sure that the jUDDI Root Seed Data install process did not run before installing RiftSaw. In other words, do not start JBossESB-4.7 before installing RiftSaw.

JBoss ESB Integration

6.1. Overview

This section outlines the support provided for the direct integration between RiftSaw and JBossESB.

Bi-directional loose integration is available through the use of web services. For example, an ESB action may invoke a BPEL process running within RiftSaw by invoking the appropriate web service represented by a WSDL interface. Similarly, a BPEL process can invoke an ESB managed service that is capable of presenting itself as a web service.

However this section will describe how integration between RiftSaw and JBossESB actions can be achieved without the use of web services (i.e. WSDL and SOAP).

6.2. Using the *BPELInvoke* ESB action

The *BPELInvoke* ESB action can be used within a *jboss-esb.xml* to request an invocation on a BPEL process running inside RiftSaw. The only constraints are that RiftSaw is installed within the same Java VM and that the requested BPEL process must have been deployed to the local RiftSaw engine.

The following example illustrates the *BPELInvoke* ESB action being used as part of the *bpel_helloworld* sample.

```
<action name="action2" class="org.jboss.soa.esb.actions.bpel.BPELInvoke">
    <property name="service" value="{http://www.jboss.org/bpel/examples/wsdl}HelloService"/>
    <property name="port" value="HelloPort" />
    <property name="operation" value="hello" />
    <property name="requestPartName" value="TestPart" />
    <property name="responsePartName" value="TestPart" />
</action>
```

The ESB action class is *org.jboss.soa.esb.actions.bpel.BPELInvoke*.

The properties for this ESB action are:

- service

  This property is mandatory, and defines the service name registered in the WSDL associated with the deployed BPEL process.

- port

  This property is optional, and defines the port name registered in the WSDL associated with the deployed BPEL process. This parameter is only required if port specific endpoint configuration information has been registered as part of the BPEL process deployment.

- operation

  This property is mandatory, and represents the WSDL operation that is being invoked.

- requestPartName

  This optional property can be used to define the WSDL message part that the inbound ESB message content should be mapped to. This property should be used where the ESB message does not already represent a multi-part message.

- responsePartName

  This optional property can be used to extract the content of a response multi-part WSDL message, and place this in the ESB message being passed to the next ESB action in the pipeline. If this property is not defined, then the complete multi-part message value will be placed in the ESB message.

- abortOnFault

  This optional property can be used to indicate whether a fault, generated during the invocation of a BPEL process, should be treated as a message (when the value of this property is 'false'), or as an exception that will abort the ESB service. The default value is 'true', causing the ESB service to abort.

This ESB action supports inbound messages with content defined as either:

- DOM

  If the message content is a DOM document or element, then this can either be used as the complete multi-part message, or as the content of a message part defined using the *requestPartName* property.

  If the message content is a DOM text node, then this can ONLY be used if a multi-part name has been defined in the *requestPartName* property.

- Java String

  If the message content is a string representation of an XML document, then the *requestPartName* is optional. If not specified, then the document must represent the multipart message.

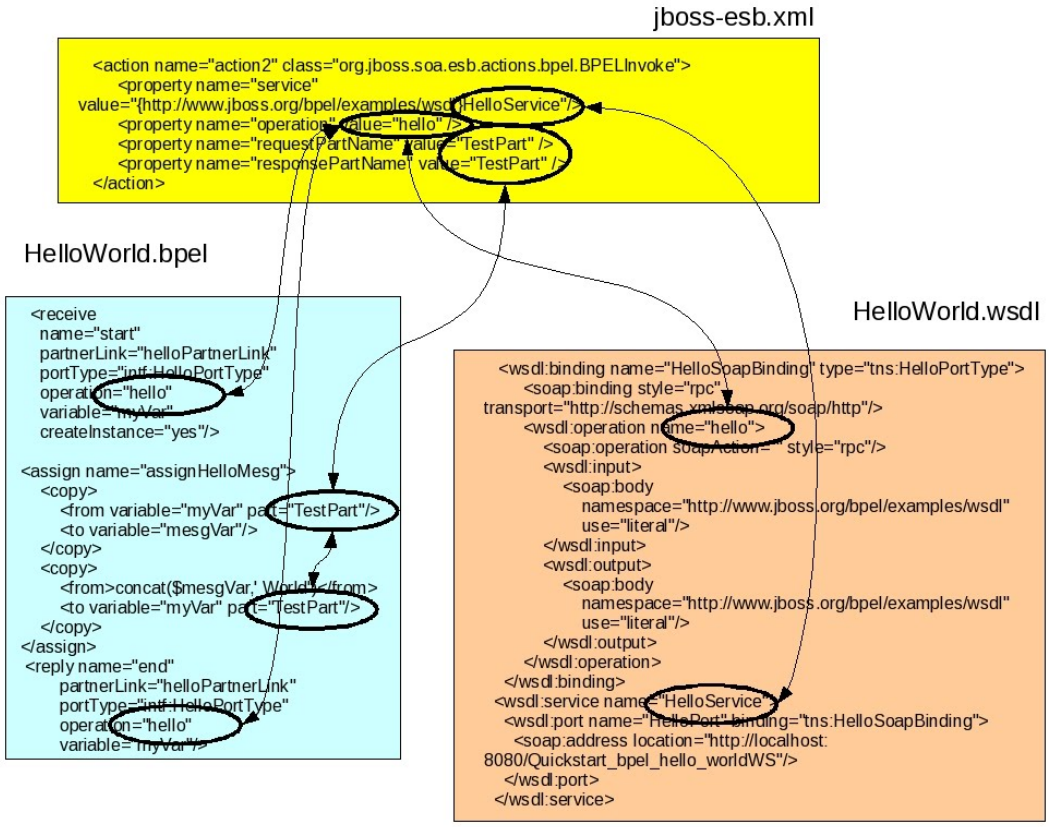
  If the message content is a string that does not represent an XML document, then the *requestPartName* must be specified.

When the message content represents the complete multipart message, this must be defined as a top level element (whose name is irrelevant) with immediate child elements that represent each of the multiple parts of the message. Each of these elements must then have a single element/node, that represents the value of the named part.

```
<message>
    <TestPart>
        Hello World
    </TestPart>
</message>
```

This shows an example of a multipart message structure. The top element (i.e. *message*) is unimportant. The elements at the next level represent the part names - in this case there is only a single part, with name *TestPart*. The value of this part is defined as a text node, with value "Hello World". However this could have been an element representing the root node of a more complex XML value.

The following diagram illustrates the inter-relationship of the JBossESB bpel_helloworld quickstart and the RiftSaw BPEL process configuration files.



6.2.1. Fault Handling

The normal response from a WSDL operation will be returned from the *BPELInvoke* ESB action as a normal message and placed on the action pipeline ready for processing by the next ESB action, or alternatively if no further actions have been defined, then returned back to the service client.

Faults, associated with a WSDL operation, are handled slightly differently. Depending on configuration it is possible to receive the fault as an ESB message or for the fault to be treated as an exception which aborts the action pipeline. The configuration property used to determine which behaviour is used is called *abortOnFault*. The default value for this property is "true". As an example, from the loan fault quickstart sample,

```
<action name="action2" class="org.jboss.soa.esb.actions.bpel.BPELInvoke">
    <property name="service" value="{http://example.com/loan-approval/wsdl/}loanService"/>
    <property name="operation" value="request" />
    <property name="abortOnFault" value="true" />
</action>
```

A WSDL fault has two relevant pieces of information, the fault type (or code) and the fault details. These are both returned in specific parts of ESB message's body.

1. Fault code (as javax.xml.namespace.QName)
   ESB message body part: *org.jboss.soa.esb.message.fault.detail.code*

   This body part identifies the specific WSDL fault returned by the BPEL process, associated with the WSDL operation that was invoked.

   Warning

   The specific version of the QName used by the JBoss server is from the stax-api.jar found in the server's lib/endorsed directory. If the client does not also include this jar in a folder that is in its endorsed directories, then a class version exception will occur when this ESB message part is accessed.

2. Fault code (as textual representation of QName)
   ESB message body part: *org.jboss.soa.bpel.message.fault.detail.code*

   This body part will return the textual representation of the QName for the fault code. The textual representation is of the form "{namespace}localpart" and can be converted back into a QName using the *javax.xml.namespace.QName.valueOf(String)* method.

3. Fault details
   ESB message body part: *org.jboss.soa.esb.message.fault.detail.detail*

   This body part will contain the textual representation of the message content associated with the fault.