



Developer Guide

Exported from [JBoss Community Documentation Editor](#) at 2017-06-19 14:13:37 EDT
Copyright 2017 JBoss Community contributors.



Table of Contents

1	WildFly Developer Guide	15
1.1	Target Audience	15
1.2	Prerequisites	15
2	Class loading in WildFly	16
2.1	Deployment Module Names	16
2.2	Automatic Dependencies	16
2.3	Class Loading Precedence	17
2.4	WAR Class Loading	17
2.5	EAR Class Loading	17
2.5.1	Class Path Entries	20
2.6	Global Modules	20
2.7	JBoss Deployment Structure File	20
2.8	Accessing JDK classes	22
2.9	The "jboss.api" property and application use of modules shipped with WildFly	22
3	Implicit module dependencies for deployments	24
3.1	What's an implicit module dependency?	24
3.2	How and when is an implicit module dependency added?	25
3.3	Which are the implicit module dependencies?	25
4	How do I migrate my application from JBoss AS 5 or AS 6 to WildFly?	28
5	EJB invocations from a remote standalone client using JNDI	29
5.1	Deploying your EJBs on the server side:	29
5.2	Writing a remote client application for accessing and invoking the EJBs deployed on the server	31
5.3	Setting up EJB client context properties	37
5.4	Summary	41
6	EJB invocations from a remote server	42
6.1	Application packaging	42
6.2	Beans	43
6.3	Security	43
6.4	Configuring a user on the "Destination Server"	44
6.5	Start the "Destination Server"	45
6.6	Deploying the application	45
6.7	Configuring the "Client Server" to point to the EJB remoting connector on the "Destination Server"	45
6.8	Start the "Client Server"	46
6.9	Create a security realm on the client server	46
6.10	Create a outbound-socket-binding on the "Client Server"	48
6.11	Create a "remote-outbound-connection" which uses this newly created "outbound-socket-binding"	48
6.12	Packaging the client application on the "Client Server"	50
6.13	Contents on jboss-ejb-client.xml	51
6.14	Deploy the client application	51
6.15	Client code invoking the bean	52



7	Remote EJB invocations via JNDI - Which approach to use?	53
8	JBoss EJB 3 reference guide	54
8.1	Resource Adapter for Message Driven Beans	54
8.1.1	Specification of Resource Adapter using Metadata Annotations	54
8.2	Run-as Principal	55
8.2.1	Specification of Run-as Principal using Metadata Annotations	55
8.3	Security Domain	55
8.3.1	Specification of Security Domain using Metadata Annotations	55
8.4	Transaction Timeout	55
8.4.1	Specification of Transaction Timeout with Metadata Annotations	56
8.4.2	Specification of Transaction Timeout in the Deployment Descriptor	57
8.5	Timer service	57
8.5.1	Single event timer	58
8.5.2	Recurring timer	58
8.5.3	Calendar timer	59
9	JPA reference guide	60
9.1	Introduction	61
9.2	Update your Persistence.xml for Hibernate 5.0	61
9.3	Entity manager	61
9.4	Application-managed entity manager	62
9.5	Container-managed entity manager	62
9.6	Persistence Context	62
9.7	Transaction-scoped Persistence Context	63
9.8	Extended Persistence Context	63
9.8.1	Extended Persistence Context Inheritance	63
9.9	Entities	65
9.10	Deployment	66
9.11	Troubleshooting	66
9.12	Using the Hibernate 5.x JPA persistence provider	68
9.13	Hibernate ORM 3.x integration is not included	68
9.14	Using the Infinispan second level cache	68
9.15	Replacing the current Hibernate 5.x jars with a newer version	71
9.16	Using Hibernate Search	71
9.17	Packaging the Hibernate JPA persistence provider with your application	72
9.18	Using OpenJPA	73
9.19	Using EclipseLink	73
9.20	Using DataNucleus	75
9.21	Native Hibernate use	75
9.22	Injection of Hibernate Session and SessionFactoryInjection of Hibernate Session and SessionFactory	75
9.23	Hibernate properties	75
9.24	Persistence unit properties	77
9.25	Determine the persistence provider module	79
9.26	Binding EntityManagerFactory/EntityManager to JNDI	80
9.27	Community	81
9.27.1	People who have contributed to the WildFly JPA layer:	81



10	OSGi developer guide	82
11	JNDI reference guide	83
11.1	Overview	83
11.2	Local JNDI	83
11.2.1	Binding entries to JNDI	84
11.2.2	Retrieving entries from JNDI	86
11.3	Remote JNDI	88
11.3.1	remote:	88
11.3.2	ejb:	89
12	Spring applications development and migration guide	90
12.1	Dependencies and Modularity	90
12.2	Persistence usage guide	90
12.3	Native Spring/Hibernate applications	90
12.4	JPA-based applications	90
12.4.1	Using server-deployed persistence units	91
12.4.2	Using Spring-managed persistence units	92
12.4.3	Managing dependencies	93
13	All WildFly documentation	94
14	Application Client Reference	95
14.1	Getting Started	95
14.2	Connecting to more than one host	95
14.3	Example	96
15	CDI Reference	97
15.1	Using CDI Beans from outside the deployment	97
15.2	Suppressing implicit bean archives	98
15.2.1	Per-deployment configuration	98
15.2.2		98
15.2.3	Global configuration	98
15.3	Development mode	99
15.3.1	Per-deployment configuration	99
15.3.2		99
15.3.3	Global configuration	99
15.4	Non-portable mode	100
15.4.1	Per-deployment configuration	100
15.4.2	Global configuration	100
16	Class Loading in WildFly	101
16.1	Deployment Module Names	101
16.2	Automatic Dependencies	101
16.3	Class Loading Precedence	102
16.4	WAR Class Loading	102
16.5	EAR Class Loading	102
16.5.1	Class Path Entries	105
16.6	Global Modules	105
16.7	JBoss Deployment Structure File	105
16.8	Accessing JDK classes	107
16.9	The "jboss.api" property and application use of modules shipped with WildFly	107



17	Deployment Descriptors used In WildFly	109
18	Development Guidelines and Recommended Practices	113
19	EE Concurrency Utilities	114
19.1	Overview	114
19.2	Context Service	115
19.3	Managed Thread Factory	115
19.4	Managed Executor Service	117
19.5	Managed Scheduled Executor Service	117
20	EJB 3 Reference Guide	120
20.1	Resource Adapter for Message Driven Beans	120
20.1.1	Specification of Resource Adapter using Metadata Annotations	120
20.2	Run-as Principal	121
20.2.1	Specification of Run-as Principal using Metadata Annotations	121
20.3	Security Domain	121
20.3.1	Specification of Security Domain using Metadata Annotations	121
20.4	Transaction Timeout	121
20.4.1	Specification of Transaction Timeout with Metadata Annotations	122
20.4.2	Specification of Transaction Timeout in the Deployment Descriptor	123
20.5	Timer service	123
20.5.1	Single event timer	124
20.5.2	Recurring timer	124
20.5.3	Calendar timer	125
20.6	Container interceptors	125
20.6.1	Overview	125
20.6.2	Typical EJB invocation call path on the server	126
20.6.3	Feature request for WildFly	126
20.6.4	Configuring container interceptors	126
20.6.5	Container interceptor positioning in the interceptor chain	129
20.6.6	Semantic difference between container interceptor(s) and Java EE interceptor(s) API	129
20.6.7	Testcase	129
20.7	EJB3 Clustered Database Timers	129
20.7.1	Overview	130
20.7.2	Setup	131
20.7.3	Using clustered timers in a deployment	132
20.7.4	Technical details	132
20.8	EJB3 subsystem configuration guide	132
20.8.1	<session-bean>	135
20.8.2	<mdb>	135
20.8.3	<entity-bean>	136
20.8.4		136
20.8.5	<pools>	136
20.8.6	<caches>	136
20.8.7	<passivation-stores>	136
20.8.8	<async>	136
20.8.9	<timer-service>	136
20.8.10	<remote>	137



20.8.11	<thread-pools>	137
20.8.12	<iiop>	137
20.8.13	<in-vm-remote-interface-invocation>	138
20.9	EJB IIOB Guide	139
20.9.1	Enabling IIOB	139
20.9.2	Enabling JTS	139
20.9.3	Dynamic Stub's	139
20.9.4	Configuring EJB IIOB settings via jboss-ejb3.xml	139
20.10	jboss-ejb3.xml Reference	139
20.10.1	Example File	139
20.11	Message Driven Beans Controlled Delivery	141
20.11.1	Delivery Active	142
20.11.2	Delivery Groups	143
20.11.3	Clustered Singleton Delivery	147
20.11.4	Using Multiple MDB Delivery Control Mechanisms	149
20.12	Securing EJBs	149
20.12.1	Overview	149
20.12.2	Security Domain	149
20.12.3	Absence of security domain configuration but presence of other security metadata	151
20.12.4	Access to methods without explicit security metadata, on a secured bean	151
21	EJB invocations from a remote client using JNDI	154
21.1	Deploying your EJBs on the server side:	154
21.2	Writing a remote client application for accessing and invoking the EJBs deployed on the server	156
21.3	Setting up EJB client context properties	162
21.4	Summary	166
22	EJB invocations from a remote server instance	167
22.1	Application packaging	167
22.2	Beans	168
22.3	Security	168
22.4	Configuring a user on the "Destination Server"	169
22.5	Start the "Destination Server"	170
22.6	Deploying the application	170
22.7	Configuring the "Client Server" to point to the EJB remoting connector on the "Destination Server"	170
22.8	Start the "Client Server"	171
22.9	Create a security realm on the client server	171
22.10	Create a outbound-socket-binding on the "Client Server"	173
22.11	Create a "remote-outbound-connection" which uses this newly created "outbound-socket-binding"	173
22.12	Packaging the client application on the "Client Server"	175
22.13	Contents on jboss-ejb-client.xml	176
22.14	Deploy the client application	176
22.15	Client code invoking the bean	177
23	Example Applications - Migrated to WildFly	178
23.1	Example Applications Migrated from Previous Releases	178
23.1.1	Seam 2 JPA example	178



23.1.2 Seam 2 DVD Store example	178
23.1.3 Seam 2 Booking example	178
23.1.4 Seam 2 Booking - step-by-step migration of binaries	178
23.1.5 jBPM-Console application	178
23.1.6 Order application used for performance testing	179
23.1.7 Migrate example application	179
23.2 Example Applications Based on EE6	179
23.3 Porting the Order Application from EAP 5.1 to WildFly 8	179
23.3.1 Overview of the application	179
23.3.2 Summary of changes	179
23.4 Seam 2 Booking Application - Migration of Binaries from EAP5.1 to WildFly	185
23.4.1 Step 1: Build and deploy the EAP5.1 version of the Seam Booking application	186
23.4.2 Step 2: Debug and resolve deployment errors and exceptions	186
23.4.3 Step 3: Debug and resolve runtime errors and exceptions	201
23.4.4 Step 4: Access the application	205
23.4.5 Summary of Changes	205
24 How do I migrate my application from AS7 to WildFly	207
24.1 About this Document	208
24.2 Overview of WildFly	209
24.3 Server Migration	209
24.3.1 JacORB Subsystem	209
24.3.2 JBoss Web Subsystem	211
24.3.3 Messaging Subsystem	218
24.4 Application Migration	222
24.4.1 EJBs	222
24.4.2 JMS	225
24.4.3 JPA (and Hibernate)	226
24.4.4 Web Applications	227
24.4.5 Web Services	227
24.4.6 Application Clustering	229
24.4.7 Other Specifications and Frameworks	232
25 How do I migrate my application to WildFly from other application servers	233
25.1 Choose from the list below:	233
25.2 How do I migrate my application from WebLogic to WildFly	233
25.2.1 Introduction	233
25.3 How do I migrate my application from WebSphere to WildFly	234
25.3.1 Introduction	234
26 Implicit module dependencies for deployments	235
26.1 What's an implicit module dependency?	235
26.2 How and when is an implicit module dependency added?	236
26.3 Which are the implicit module dependencies?	236
27 JAX-RS Reference Guide	239
27.1 Subclassing javax.ws.rs.core.Application and using @ApplicationPath	239
27.2 Subclassing javax.ws.rs.core.Application and using web.xml	240
27.3 Using web.xml	240
28 JNDI Reference	241



28.1 Overview	241
28.2 Local JNDI	241
28.2.1 Binding entries to JNDI	242
28.2.2 Retrieving entries from JNDI	244
28.3 Remote JNDI	246
28.3.1 remote:	246
28.3.2 ejb:	247
28.4 Local JNDI	247
28.4.1 Binding entries to JNDI	248
28.4.2 Retrieving entries from JNDI	250
28.5 Remote JNDI Reference	252
28.5.1 Remote JNDI	252
28.5.2 Remote JNDI Access	253
29 JPA Reference Guide	256
29.1 Introduction	257
29.2 Update your Persistence.xml for Hibernate 5.0	257
29.3 Entity manager	257
29.4 Application-managed entity manager	258
29.5 Container-managed entity manager	258
29.6 Persistence Context	258
29.7 Transaction-scoped Persistence Context	259
29.8 Extended Persistence Context	259
29.8.1 Extended Persistence Context Inheritance	259
29.9 Entities	261
29.10 Deployment	262
29.11 Troubleshooting	262
29.12 Using the Hibernate 5.x JPA persistence provider	264
29.13 Hibernate ORM 3.x integration is not included	264
29.14 Using the Infinispan second level cache	264
29.15 Replacing the current Hibernate 5.x jars with a newer version	267
29.16 Using Hibernate Search	267
29.17 Packaging the Hibernate JPA persistence provider with your application	268
29.18 Using OpenJPA	269
29.19 Using EclipseLink	269
29.20 Using DataNucleus	271
29.21 Native Hibernate use	271
29.22 Injection of Hibernate Session and SessionFactory	271
29.23 Hibernate properties	271
29.24 Persistence unit properties	273
29.25 Determine the persistence provider module	275
29.26 Binding EntityManagerFactory/EntityManager to JNDI	276
29.27 Community	277
29.27.1 People who have contributed to the WildFly JPA layer:	277
30 OSGi	278
31 Remote EJB invocations via JNDI - EJB client API or remote-naming project	279



31.1 Purpose	279
31.2 History	279
31.3 Overview	279
31.3.1 Client code relying on jndi.properties in classpath	279
31.3.2 How does remotng naming work	281
31.3.3 JNDI operations allowed using remote-naming project	282
31.3.4 Pre-requisites of remotely accessible JNDI objects	283
31.3.5 JNDI lookup strings for remote clients backed by the remote-naming project	283
31.3.6 How does remote-naming project implementation transfer the JNDI objects to the clients	284
31.4 Summary	284
31.5 Remote EJB invocations backed by the remote-naming project	284
31.6 Why use the EJB client API approach then?	287
31.6.1 Is the lookup optimization applicable for all bean types?	289
31.6.2 Restrictions for EJB's	290
32 Scoped EJB client contexts	291
32.1 Overview	291
32.2 Potential shortcomings of a single EJB client context	291
32.3 Scoped EJB client contexts	292
32.4 Lifecycle management of scoped EJB client contexts	294
32.4.1 How to close EJB client contexts?	295
32.4.2 How to close scoped EJB client contexts?	295
32.4.3 Can't the scoped EJB client context be automatically closed by the EJB client API when the JNDI context is no longer in scope (i.e. on GC)?	299
33 Spring applications development and migration guide	300
33.1 Dependencies and Modularity	300
33.2 Persistence usage guide	300
33.3 Native Spring/Hibernate applications	300
33.4 JPA-based applications	300
33.4.1 Using server-deployed persistence units	301
33.4.2 Using Spring-managed persistence units	302
33.4.3 Managing dependencies	303
34 Sharing sessions between wars in an ear	304
35 Webservices reference guide	305
35.1 JAX-WS User Guide	305
35.1.1 Web Service Endpoints	305
35.1.2 Web Service Clients	309
35.1.3 Common API	315
35.1.4 JAX-WS Annotations	318
35.1.5 JSR-181 Annotations	320
35.2 JAX-WS Tools	321
35.2.1 Server side	321
35.2.2 Client Side	326
35.2.3 wsconsume	329
35.2.4 wsprovide	336
35.3 Advanced User Guide	341



35.3.1 Logging	342
35.3.2 WS-* support	343
35.3.3 Address rewrite	344
35.3.4 Configuration through deployment descriptor	344
35.3.5 Schema validation of SOAP messages	348
35.3.6 JAXB Introductions	349
35.3.7 WSDL system properties expansion	349
35.3.8 Predefined client and endpoint configurations	349
35.3.9 Authentication	358
35.3.10 Apache CXF integration	362
35.3.11 WS-Addressing	376
35.3.12 WS-Security	379
35.3.13 WS-Trust and STS	404
35.3.14 WS-Reliable Messaging	500
35.3.15 SOAP over JMS	506
35.3.16 HTTP Proxy	517
35.3.17 WS-Discovery	520
35.3.18 WS-Policy	520
35.3.19 Published WSDL customization	525
35.4 JBoss Modules and WS applications	529
35.4.1 Setting module dependencies	530



- [WildFly Developer Guide](#)
 - [Target Audience](#)
 - [Prerequisites](#)
- [Class loading in WildFly](#)
 - [Deployment Module Names](#)
 - [Automatic Dependencies](#)
 - [Class Loading Precedence](#)
 - [WAR Class Loading](#)
 - [EAR Class Loading](#)
 - [Class Path Entries](#)
 - [Global Modules](#)
 - [JBoss Deployment Structure File](#)
 - [Accessing JDK classes](#)
 - [The "jboss.api" property and application use of modules shipped with WildFly](#)
- [Implicit module dependencies for deployments](#)
 - [What's an implicit module dependency?](#)
 - [How and when is an implicit module dependency added?](#)
 - [Which are the implicit module dependencies?](#)
- [How do I migrate my application from JBoss AS 5 or AS 6 to WildFly?](#)
- [EJB invocations from a remote standalone client using JNDI](#)
 - [Deploying your EJBs on the server side:](#)
 - [Writing a remote client application for accessing and invoking the EJBs deployed on the server](#)
 - [Setting up EJB client context properties](#)
 - [Using a different file for setting up EJB client context](#)
 - [Setting up the client classpath with the jars that are required to run the client application](#)
 - [Summary](#)
- [EJB invocations from a remote server](#)
 - [Application packaging](#)
 - [Beans](#)
 - [Security](#)
 - [Configuring a user on the "Destination Server"](#)
 - [Start the "Destination Server"](#)
 - [Deploying the application](#)
 - [Configuring the "Client Server" to point to the EJB remoting connector on the "Destination Server"](#)
 - [Start the "Client Server"](#)
 - [Create a security realm on the client server](#)
 - [Create a outbound-socket-binding on the "Client Server"](#)
 - [Create a "remote-outbound-connection" which uses this newly created "outbound-socket-binding"](#)
 - [Packaging the client application on the "Client Server"](#)
 - [Contents on jboss-ejb-client.xml](#)
 - [Deploy the client application](#)
 - [Client code invoking the bean](#)
- [Remote EJB invocations via JNDI - Which approach to use?](#)



- [JBoss EJB 3 reference guide](#)
 - [Resource Adapter for Message Driven Beans](#)
 - [Specification of Resource Adapter using Metadata Annotations](#)
 - [Run-as Principal](#)
 - [Specification of Run-as Principal using Metadata Annotations](#)
 - [Security Domain](#)
 - [Specification of Security Domain using Metadata Annotations](#)
 - [Transaction Timeout](#)
 - [Specification of Transaction Timeout with Metadata Annotations](#)
 - [Specification of Transaction Timeout in the Deployment Descriptor](#)
 - [Example of `trans-timeout`](#)
 - [Timer service](#)
 - [Single event timer](#)
 - [Recurring timer](#)
 - [Calendar timer](#)
 - [Programmatic calendar timer](#)
 - [Annotated calendar timer](#)



- [JPA reference guide](#)
 - [Introduction](#)
 - [Update your Persistence.xml for Hibernate 5.0](#)
 - [Entity manager](#)
 - [Application-managed entity manager](#)
 - [Container-managed entity manager](#)
 - [Persistence Context](#)
 - [Transaction-scoped Persistence Context](#)
 - [Extended Persistence Context](#)
 - [Extended Persistence Context Inheritance](#)
 - [Entities](#)
 - [Deployment](#)
 - [Troubleshooting](#)
 - [Using the Hibernate 5.x JPA persistence provider](#)
 - [Hibernate ORM 3.x integration is not included](#)
 - [Using the Infinispan second level cache](#)
 - [Replacing the current Hibernate 5.x jars with a newer version](#)
 - [Using Hibernate Search](#)
 - [Packaging the Hibernate JPA persistence provider with your application](#)
 - [Using OpenJPA](#)
 - [Using EclipseLink](#)
 - [Using DataNucleus](#)
 - [Native Hibernate use](#)
 - [Injection of Hibernate Session and SessionFactory](#)
 - [Hibernate properties](#)
 - [Persistence unit properties](#)
 - [Determine the persistence provider module](#)
 - [Binding EntityManagerFactory/EntityManager to JNDI](#)
 - [Community](#)
 - [People who have contributed to the WildFly JPA layer:](#)
- [OSGi developer guide](#)



- [JNDI reference guide](#)
 - [Overview](#)
 - [Local JNDI](#)
 - [Binding entries to JNDI](#)
 - [Using a deployment descriptor](#)
 - [Programmatically](#)
 - [Java EE Applications](#)
 - [WildFly Modules and Extensions](#)
 - [Naming Subsystem Configuration](#)
 - [Retrieving entries from JNDI](#)
 - [Resource Injection](#)
 - [Standard Java SE JNDI API](#)
 - [Remote JNDI](#)
 - [remote:](#)
 - [ejb:](#)
- [Spring applications development and migration guide](#)
 - [Dependencies and Modularity](#)
 - [Persistence usage guide](#)
 - [Native Spring/Hibernate applications](#)
 - [JPA-based applications](#)
 - [Using server-deployed persistence units](#)
 - [Using Spring-managed persistence units](#)
 - [Placement of the persistence unit definitions](#)
 - [Managing dependencies](#)
- [All WildFly documentation](#)



1 WildFly Developer Guide

1.1 Target Audience

Java Developers

1.2 Prerequisites



2 Class loading in WildFly

Since JBoss AS 7, Class loading is considerably different to previous versions of JBoss AS. Class loading is based on the [JBoss Modules](#) project. Instead of the more familiar hierarchical class loading environment, WildFly's class loading is based on modules that have to define explicit dependencies on other modules. Deployments in WildFly are also modules, and do not have access to classes that are defined in jars in the application server unless an explicit dependency on those classes is defined.

2.1 Deployment Module Names

Module names for top level deployments follow the format `deployment.myarchive.war` while sub deployments are named like `deployment.myear.ear.mywar.war`.

This means that it is possible for a deployment to import classes from another deployment using the other deployments module name, the details of how to add an explicit module dependency are explained below.

2.2 Automatic Dependencies

Even though in WildFly modules are isolated by default, as part of the deployment process some dependencies on modules defined by the application server are set up for you automatically. For instance, if you are deploying a Java EE application a dependency on the Java EE API's will be added to your module automatically. Similarly if your module contains a `beans.xml` file a dependency on [Weld](#) will be added automatically, along with any supporting modules that weld needs to operate.

For a complete list of the automatic dependencies that are added, please see [Implicit module dependencies for deployments](#).

Automatic dependencies can be excluded through the use of `jboss-deployment-structure.xml`.



2.3 Class Loading Precedence

A common source of errors in Java applications is including API classes in a deployment that are also provided by the container. This can result in multiple versions of the class being created and the deployment failing to deploy properly. To prevent this in WildFly, module dependencies are added in a specific order that should prevent this situation from occurring.

In order of highest priority to lowest priority

1. System Dependencies - These are dependencies that are added to the module automatically by the container, including the Java EE api's.
2. User Dependencies - These are dependencies that are added through `jboss-deployment-structure.xml` or through the `Dependencies: manifest` entry.
3. Local Resource - Class files packaged up inside the deployment itself, e.g. class files from `WEB-INF/classes` or `WEB-INF/lib` of a war.
4. Inter deployment dependencies - These are dependencies on other deployments in an ear deployment. This can include classes in an ear's lib directory, or classes defined in other ejb jars.

2.4 WAR Class Loading

The war is considered to be a single module, so classes defined in `WEB-INF/lib` are treated the same as classes in `WEB-INF/classes`. All classes packaged in the war will be loaded with the same class loader.

2.5 EAR Class Loading

Ear deployments are multi-module deployments. This means that not all classes inside an ear will necessarily have access to all other classes in the ear, unless explicit dependencies have been defined. By default the `EAR/lib` directory is a single module, and every WAR or EJB jar deployment is also a separate module. Sub deployments (wars and ejb-jars) always have a dependency on the parent module, which gives them access to classes in `EAR/lib`, however they do not always have an automatic dependency on each other. This behaviour is controlled via the `ear-subdeployments-isolated` setting in the ee subsystem configuration:

```
<subsystem xmlns="urn:jboss:domain:ee:1.0" >
  <ear-subdeployments-isolated>false</ear-subdeployments-isolated>
</subsystem>
```


By default this is set to false, which allows the sub-deployments to see classes belonging to other sub-deployments within the .ear.

For example, consider the following .ear deployment:



```
myapp.ear
|
|--- web.war
|
|--- ejb1.jar
|
|--- ejb2.jar
```


If the `ear-subdeployments-isolated` is set to `false`, then the classes in `web.war` can access classes belonging to `ejb1.jar` and `ejb2.jar`. Similarly, classes from `ejb1.jar` can access classes from `ejb2.jar` (and vice-versa).

 The `ear-subdeployments-isolated` element value has no effect on the isolated classloader of the `.war` file(s). i.e. irrespective of whether this flag is set to `true` or `false`, the `.war` within a `.ear` will have a isolated classloader and other sub-deployments within that `.ear` will not be able to access classes from that `.war`. This is as per spec.

If the `ear-subdeployments-isolated` is set to `true` then no automatic module dependencies between the sub-deployments are set up. User must manually setup the dependency with `Class-Path` entries, or by setting up explicit module dependencies.

Portability

The Java EE specification says that portable applications should not rely on sub deployments having access to other sub deployments unless an explicit `Class-Path` entry is set in the `MANIFEST.MF`. So portable applications should always use `Class-Path` entry to explicitly state their dependencies.

 It is also possible to override the `ear-subdeployments-isolated` element value at a per deployment level. See the section on `jboss-deployment-structure.xml` below.

Dependencies: **Manifest Entries**

Deployments (or more correctly modules within a deployment) may set up dependencies on other modules by adding a `Dependencies: manifest` entry. This entry consists of a comma separated list of module names that the deployment requires. The available modules can be seen under the `modules` directory in the application server distribution. For example to add a dependency on `javassist` and `apache velocity` you can add a manifest entry as follows:

```
Dependencies: org.javassist export,org.apache.velocity export services,organtlr
```


Each dependency entry may also specify some of the following parameters by adding them after the module name:



- `export` This means that the dependencies will be exported, so any module that depends on this module will also get access to the dependency.
- `services` By default items in `META-INF` of a dependency are not accessible, this makes items from `META-INF/services` accessible so `services` in the modules can be loaded.
- `optional` If this is specified the deployment will not fail if the module is not available.
- `meta-inf` This will make the contents of the `META-INF` directory available (unlike `services`, which just makes `META-INF/services` available). In general this will not cause any deployment descriptors in `META-INF` to be processed, with the exception of `beans.xml`. If a `beans.xml` file is present this module will be scanned by Weld and any resulting beans will be available to the application.
- `annotations` If a jandex index has been created for the module these annotations will be merged into the deployments annotation index. The `Jandex` index can be generated using the `Jandex ant task`, and must be named `META-INF/jandex.idx`. Note that it is not necessary to break open the jar being indexed to add this to the modules class path, a better approach is to create a jar containing just this index, and adding it as an additional resource root in the `module.xml` file.

Adding a dependency to all modules in an EAR

Using the `export` parameter it is possible to add a dependency to all sub deployments in an ear. If a module is exported from a `Dependencies:` entry in the top level of the ear (or by a jar in the `ear/lib` directory) it will be available to all sub deployments as well.

 To generate a MANIFEST.MF entry when using maven put the following in your pom.xml:

pom.xml

```
<build>
  ...
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-war-plugin</artifactId>
      <configuration>
        <archive>
          <manifestEntries>
            <Dependencies>org.slf4j</Dependencies>
          </manifestEntries>
        </archive>
      </configuration>
    </plugin>
  </plugins>
</build>
```

If your deployment is a jar you must use the `maven-jar-plugin` rather than the `maven-war-plugin`.



2.5.1 Class Path Entries

It is also possible to add module dependencies on other modules inside the deployment using the `Class-Path` manifest entry. This can be used within an ear to set up dependencies between sub deployments, and also to allow modules access to additional jars deployed in an ear that are not sub deployments and are not in the `EAR/lib` directory. If a jar in the `EAR/lib` directory references a jar via `Class-Path`: then this additional jar is merged into the parent ear's module, and is accessible to all sub deployments in the ear.

2.6 Global Modules

It is also possible to set up global modules, that are accessible to all deployments. This is done by modifying the configuration file (`standalone/domain.xml`).

For example, to add `javassist` to all deployments you can use the following XML:

`standalone.xml/domain.xml`

```
<subsystem xmlns="urn:jboss:domain:ee:1.0" >
  <global-modules>
    <module name="org.javassist" slot="main" />
  </global-modules>
</subsystem>
```

Note that the `slot` field is optional and defaults to `main`.

2.7 JBoss Deployment Structure File

`jboss-deployment-structure.xml` is a JBoss specific deployment descriptor that can be used to control class loading in a fine grained manner. It should be placed in the top level deployment, in `META-INF` (or `WEB-INF` for web deployments). It can do the following:

- Prevent automatic dependencies from being added
- Add additional dependencies
- Define additional modules
- Change an EAR deployments isolated class loading behaviour
- Add additional resource roots to a module

An example of a complete `jboss-deployment-structure.xml` file for an ear deployment is as follows:

`jboss-deployment-structure.xml`


```
<jboss-deployment-structure>
  <!-- Make sub deployments isolated by default, so they cannot see each others classes without
  a Class-Path entry -->
```



```
<ear-subdeployments-isolated>true</ear-subdeployments-isolated>
<!-- This corresponds to the top level deployment. For a war this is the war's module, for an
ear -->
<!-- This is the top level ear module, which contains all the classes in the EAR's lib folder
-->
<deployment>
  <!-- exclude-subsystem prevents a subsystems deployment unit processors running on a
deployment -->
  <!-- which gives basically the same effect as removing the subsystem, but it only affects
single deployment -->
  <exclude-subsystems>
    <subsystem name="resteasy" />
  </exclude-subsystems>
  <!-- Exclusions allow you to prevent the server from automatically adding some dependencies
-->
  <exclusions>
    <module name="org.javassist" />
  </exclusions>
  <!-- This allows you to define additional dependencies, it is the same as using the
Dependencies: manifest attribute -->
  <dependencies>
    <module name="deployment.javassist.proxy" />
    <module name="deployment.myjavassist" />
    <!-- Import META-INF/services for ServiceLoader impls as well -->
    <module name="myservicemodule" services="import"/>
  </dependencies>
  <!-- These add additional classes to the module. In this case it is the same as including
the jar in the EAR's lib directory -->
  <resources>
    <resource-root path="my-library.jar" />
  </resources>
</deployment>
<sub-deployment name="myapp.war">
  <!-- This corresponds to the module for a web deployment -->
  <!-- it can use all the same tags as the <deployment> entry above -->
  <dependencies>
    <!-- Adds a dependency on a ejb jar. This could also be done with a Class-Path entry -->
    <module name="deployment.myear.ear.myejbjar.jar" />
  </dependencies>
  <!-- Set's local resources to have the lowest priority -->
  <!-- If the same class is both in the sub deployment and in another sub deployment that -->
  <!-- is visible to the war, then the Class from the other deployment will be loaded, -->
  <!-- rather than the class actually packaged in the war. -->
  <!-- This can be used to resolve ClassCastExceptions if the same class is in multiple sub
deployments-->
  <local-last value="true" />
</sub-deployment>
<!-- Now we are going to define two additional modules -->
<!-- This one is a different version of javassist that we have packaged -->
<module name="deployment.myjavassist" >
  <resources>
    <resource-root path="javassist.jar" >
      <!-- We want to use the servers version of javassist.util.proxy.* so we filter it out-->
      <filter>
        <exclude path="javassist/util/proxy" />
      </filter>
    </resource-root>
  </resources>
```



```
</module>
<!-- This is a module that re-exports the containers version of javassist.util.proxy -->
<!-- This means that there is only one version of the Proxy classes defined -->
<module name="deployment.javassist.proxy" >
  <dependencies>
    <module name="org.javassist" >
      <imports>
        <include path="javassist/util/proxy" />
        <exclude path="/**" />
      </imports>
    </module>
  </dependencies>
</module>
</jboss-deployment-structure>
```

 The xsd for `jboss-deployment-structure.xml` is available at <https://github.com/wildfly/wildfly/blob/master/build/src/main/resources/docs/schema/jboss-deployment-structure.xsd>

2.8 Accessing JDK classes

Not all JDK classes are exposed to a deployment by default. If your deployment uses JDK classes that are not exposed you can get access to them using `jboss-deployment-structure.xml` with system dependencies:

Using `jboss-deployment-structure.xml` to access JDK classes

```
<jboss-deployment-structure xmlns="urn:jboss:deployment-structure:1.1">
  <deployment>
    <dependencies>
      <system export="true">
        <paths>
          <path name="com/sun/corba/se/spi/legacy/connection"/>
        </paths>
      </system>
    </dependencies>
  </deployment>
</jboss-deployment-structure>
```

2.9 The "jboss.api" property and application use of modules shipped with WildFly

The WildFly distribution includes a large number of modules, a great many of which are included for use by WildFly internals, with no testing of the appropriateness of their direct use by applications or any commitment to continue to ship those modules in future releases if they are no longer needed by the internals. So how can a user know whether it is advisable for their application to specify an explicit dependency on a module WildFly ships? The "jboss.api" property specified in the module's `module.xml` file can tell you:

**Example declaration of the jboss.api property**

```
<module xmlns="urn:jboss:module:1.3" name="com.google.guava">
  <properties>
    <property name="jboss.api" value="private"/>
  </properties>
</module>
```

If a module does not have a property element like the above, then it's equivalent to one with a value of "public".

Following are the meanings of the various values you may see for the jboss.api property:

Value	Meaning
public	May be explicitly depended upon by end user applications. Will continue to be available in future releases within the same major series and should not have incompatible API changes in future releases within the same minor series, and ideally not within the same major series.
private	Intended for internal use only. Only tested according to internal usage. May not be safe for end user applications to use directly. Could change significantly or be removed in a future release without notice.
unsupported	If you see this value in a module.xml in a WildFly release, please file a bug report, as it is not applicable in WildFly. In EAP it has a meaning equivalent to "private" but that does not mean the module is "private" in WildFly; it could very easily be "public".
preview	May be explicitly depended upon by end user applications, but there are no guarantees of continued availability in future releases or that there will not be incompatible API changes. This is not a common classification in WildFly. It is not used in WildFly 10.
deprecated	May be explicitly depended upon by end user applications. Stable and reliable but an alternative should be sought. Will be removed in a future major release.

Note that these definitions are only applicable to WildFly. In EAP and other Red Hat products based on WildFly the same classifiers are used, with generally similar meaning, but the precise meaning is per the definitions on the Red Hat customer support portal.

If an application declares a direct dependency on a module marked "private", "unsupported" or "deprecated", during deployment a WARN message will be logged. The logging will be in log categories "org.jboss.as.dependency.private", "org.jboss.as.dependency.unsupported" and "org.jboss.as.dependency.deprecated" respectively. These categories are not used for other purposes, so once you feel sufficiently warned the logging can be safely suppressed by turning the log level for the relevant category to ERROR or higher.

Other than the WARN messages noted above, declaring a direct dependency on a non-public module has no impact on how WildFly processes the deployment.



3 Implicit module dependencies for deployments

As explained in the [Class Loading in WildFly](#) article, WildFly 8 is based on module classloading. A class within a module B isn't visible to a class within a module A, unless module B adds a dependency on module A. Module dependencies can be explicitly (as explained in that classloading article) or can be "implicit". This article will explain what implicit module dependencies mean and how, when and which modules are added as implicit dependencies.

3.1 What's an implicit module dependency?

Consider an application deployment which contains EJBs. EJBs typically need access to classes from the `javax.ejb.*` package and other Java EE API packages. The jars containing these packages are already shipped in WildFly and are available as "modules". The module which contains the `javax.ejb.*` classes has a specific name and so does the module which contains all the Java EE API classes. For an application to be able to use these classes, it has to add a dependency on the relevant modules. Forcing the application developers to add module dependencies like these (i.e. dependencies which can be "inferred") isn't a productive approach. Hence, whenever an application is being deployed, the deployers within the server, which are processing this deployment "implicitly" add these module dependencies to the deployment so that these classes are visible to the deployment at runtime. This way the application developer doesn't have to worry about adding them explicitly. How and when these implicit dependencies are added is explained in the next section.



3.2 How and when is an implicit module dependency added?

When a deployment is being processed by the server, it goes through a chain of "deployment processors". Each of these processors will have a way to check if the deployment meets a certain criteria and if it does, the deployment processor adds a implicit module dependency to that deployment. Let's take an example - Consider (again) an EJB3 deployment which has the following class:

MySuperDuperBean.java

```
@Stateless
public class MySuperDuperBean {
    ...
}
```

As can be seen, we have a simple @Stateless EJB. When the deployment containing this class is being processed, the EJB deployment processor will see that the deployment contains a class with the @Stateless annotation and thus identifies this as a EJB deployment. **This is just one of the several ways, various deployment processors can identify a deployment of some specific type.** The EJB deployment processor will then add an implicit dependency on the Java EE API module, so that all the Java EE API classes are visible to the deployment.

Some subsystems will always add a API classes, even if the trigger condition is not met. These are listed separately below.

In the next section, we'll list down the implicit module dependencies that are added to a deployment, by various deployers within WildFly.

3.3 Which are the implicit module dependencies?

Subsystem responsible for adding the implicit dependency	Dependencies that are always added	Dependencies that are added if a trigger condition is met
Core Server	<ul style="list-style-type: none">• javax.api• sun.jdk• org.jboss.vfs	



Batch Subsystem	<ul style="list-style-type: none">• javax.batch.api	
EE Subsystem	<ul style="list-style-type: none">• javaee.api	
EJB3 subsystem		<ul style="list-style-type: none">• javaee.api
JAX-RS (Resteasy) subsystem	<ul style="list-style-type: none">• javax.xml.bind.api	<ul style="list-style-type: none">• org.jboss.resteasy.resteasy-atom-provider• org.jboss.resteasy.resteasy-cdi• org.jboss.resteasy.resteasy-jaxrs• org.jboss.resteasy.resteasy-jaxb-provider• org.jboss.resteasy.resteasy-jackson-provider• org.jboss.resteasy.resteasy-jsapi• org.jboss.resteasy.resteasy-multipart-provider• org.jboss.resteasy.async-http-servlet-30
JCA subsystem	<ul style="list-style-type: none">• javax.resource.api	<ul style="list-style-type: none">• javax.jms.api• javax.validation.api• org.jboss.logging• org.jboss.ironjacamar.api• org.jboss.ironjacamar.impl• org.hibernate.validator
JPA (Hibernate) subsystem	<ul style="list-style-type: none">• javax.persistence.api	<ul style="list-style-type: none">• javaee.api• org.jboss.as.jpa• org.hibernate



Logging Subsystem	<ul style="list-style-type: none">• org.jboss.logging• org.apache.commons.logging• org.apache.log4j• org.slf4j• org.jboss.logging.jul-to-slf4j-stub	
SAR Subsystem		<ul style="list-style-type: none">• org.jboss.logging• org.jboss.modules
Security Subsystem	<ul style="list-style-type: none">• org.picketbox	
Web Subsystem		<ul style="list-style-type: none">• javax.ee.api• com.sun.jsf-impl• org.hibernate.validator• org.jboss.as.web• org.jboss.logging
Web Services Subsystem	<ul style="list-style-type: none">• org.jboss.ws.api• org.jboss.ws.spi	
Weld (CDI) Subsystem		<ul style="list-style-type: none">• javax.persistence.api• javax.ee.api• org.javassist• org.jboss.interceptor• org.jboss.as.weld• org.jboss.logging• org.jboss.weld.core• org.jboss.weld.api• org.jboss.weld.spi



4 How do I migrate my application from JBoss AS 5 or AS 6 to WildFly?

Couldn't find a page to include called: How do I migrate my application from AS5 or AS6 to WildFly



5 EJB invocations from a remote standalone client using JNDI

This chapter explains how to invoke EJBs from a remote client by using the JNDI API to first lookup the bean proxy and then invoke on that proxy.

i After you have read this article, do remember to take a look at [Remote EJB invocations via JNDI - EJB client API or remote-naming project](#)

Before getting into the details, we would like the users to know that we have introduced a new EJB client API, which is a WildFly-specific API and allows invocation on remote EJBs. This client API isn't based on JNDI. So remote clients need not rely on JNDI API to invoke on EJBs. A separate document covering the EJB remote client API will be made available. For now, you can refer to the javadocs of the EJB client project at <http://docs.jboss.org/ejbclient/>. In this document, we'll just concentrate on the traditional JNDI based invocation on EJBs. So let's get started:

5.1 Deploying your EJBs on the server side:

⚠ Users who already have EJBs deployed on the server side can just skip to the next section.

As a first step, you'll have to deploy your application containing the EJBs on the Wildfly server. If you want those EJBs to be remotely invocable, then you'll have to expose at least one remote view for that bean. In this example, let's consider a simple Calculator stateless bean which exposes a RemoteCalculator remote business interface. We'll also have a simple stateful CounterBean which exposes a RemoteCounter remote business interface. Here's the code:

```
package org.jboss.as.quickstarts.ejb.remote.stateless;

/**
 * @author Jaikiran Pai
 */
public interface RemoteCalculator {

    int add(int a, int b);

    int subtract(int a, int b);

}
```



```
package org.jboss.as.quickstarts.ejb.remote.stateless;

import javax.ejb.Remote;
import javax.ejb.Stateless;

/**
 * @author Jaikiran Pai
 */
@Stateless
@Remote(RemoteCalculator.class)
public class CalculatorBean implements RemoteCalculator {

    @Override
    public int add(int a, int b) {
        return a + b;
    }

    @Override
    public int subtract(int a, int b) {
        return a - b;
    }
}
```

```
package org.jboss.as.quickstarts.ejb.remote.stateful;

/**
 * @author Jaikiran Pai
 */
public interface RemoteCounter {

    void increment();

    void decrement();

    int getCount();
}
```



```
package org.jboss.as.quickstarts.ejb.remote.stateful;

import javax.ejb.Remote;
import javax.ejb.Stateful;

/**
 * @author Jaikiran Pai
 */
@Stateful
@Remote(RemoteCounter.class)
public class CounterBean implements RemoteCounter {

    private int count = 0;

    @Override
    public void increment() {
        this.count++;
    }

    @Override
    public void decrement() {
        this.count--;
    }

    @Override
    public int getCount() {
        return this.count;
    }
}
```

Let's package this in a jar (how you package it in a jar is out of scope of this chapter) named "jboss-as-ejb-remote-app.jar" and deploy it to the server. Make sure that your deployment has been processed successfully and there aren't any errors.

5.2 Writing a remote client application for accessing and invoking the EJBs deployed on the server

The next step is to write an application which will invoke the EJBs that you deployed on the server. In WildFly, you can either choose to use the WildFly specific EJB client API to do the invocation or use JNDI to lookup a proxy for your bean and invoke on that returned proxy. In this chapter we will concentrate on the JNDI lookup and invocation and will leave the EJB client API for a separate chapter.

So let's take a look at what the client code looks like for looking up the JNDI proxy and invoking on it. Here's the entire client code which invokes on a stateless bean:

```
package org.jboss.as.quickstarts.ejb.remote.client;

import javax.naming.Context;
import javax.naming.InitialContext;
```



```
import javax.naming.NamingException;
import java.security.Security;
import java.util.Hashtable;

import org.jboss.as.quickstarts.ejb.remote.stateful.CounterBean;
import org.jboss.as.quickstarts.ejb.remote.stateful.RemoteCounter;
import org.jboss.as.quickstarts.ejb.remote.stateless.CalculatorBean;
import org.jboss.as.quickstarts.ejb.remote.stateless.RemoteCalculator;
import org.jboss.sasl.JBossSaslProvider;

/**
 * A sample program which acts a remote client for a EJB deployed on Wildfly 10 server.
 * This program shows how to lookup stateful and stateless beans via JNDI and then invoke on
them
 *
 * @author Jaikiran Pai
 */
public class RemoteEJBClient {

    public static void main(String[] args) throws Exception {
        // Invoke a stateless bean
        invokeStatelessBean();

        // Invoke a stateful bean
        invokeStatefulBean();
    }

    /**
     * Looks up a stateless bean and invokes on it
     *
     * @throws NamingException
     */
    private static void invokeStatelessBean() throws NamingException {
        // Let's lookup the remote stateless calculator
        final RemoteCalculator statelessRemoteCalculator = lookupRemoteStatelessCalculator();
        System.out.println("Obtained a remote stateless calculator for invocation");
        // invoke on the remote calculator
        int a = 204;
        int b = 340;
        System.out.println("Adding " + a + " and " + b + " via the remote stateless calculator
deployed on the server");
        int sum = statelessRemoteCalculator.add(a, b);
        System.out.println("Remote calculator returned sum = " + sum);
        if (sum != a + b) {
            throw new RuntimeException("Remote stateless calculator returned an incorrect sum "
+ sum + " ,expected sum was " + (a + b));
        }
        // try one more invocation, this time for subtraction
        int num1 = 3434;
        int num2 = 2332;
        System.out.println("Subtracting " + num2 + " from " + num1 + " via the remote stateless
calculator deployed on the server");
        int difference = statelessRemoteCalculator.subtract(num1, num2);
        System.out.println("Remote calculator returned difference = " + difference);
        if (difference != num1 - num2) {
            throw new RuntimeException("Remote stateless calculator returned an incorrect
difference " + difference + " ,expected difference was " + (num1 - num2));
        }
    }
}
```




```
}

/**
 * Looks up a stateful bean and invokes on it
 *
 * @throws NamingException
 */
private static void invokeStatefulBean() throws NamingException {
    // Let's lookup the remote stateful counter
    final RemoteCounter statefulRemoteCounter = lookupRemoteStatefulCounter();
    System.out.println("Obtained a remote stateful counter for invocation");
    // invoke on the remote counter bean
    final int NUM_TIMES = 20;
    System.out.println("Counter will now be incremented " + NUM_TIMES + " times");
    for (int i = 0; i < NUM_TIMES; i++) {
        System.out.println("Incrementing counter");
        statefulRemoteCounter.increment();
        System.out.println("Count after increment is " + statefulRemoteCounter.getCount());
    }
    // now decrementing
    System.out.println("Counter will now be decremented " + NUM_TIMES + " times");
    for (int i = NUM_TIMES; i > 0; i--) {
        System.out.println("Decrementing counter");
        statefulRemoteCounter.decrement();
        System.out.println("Count after decrement is " + statefulRemoteCounter.getCount());
    }
}

/**
 * Looks up and returns the proxy to remote stateless calculator bean
 *
 * @return
 * @throws NamingException
 */
private static RemoteCalculator lookupRemoteStatelessCalculator() throws NamingException {
    final Hashtable jndiProperties = new Hashtable();
    jndiProperties.put(Context.URL_PKG_PREFIXES, "org.jboss.ejb.client.naming");
    final Context context = new InitialContext(jndiProperties);
    // The app name is the application name of the deployed EJBs. This is typically the ear
name
    // without the .ear suffix. However, the application name could be overridden in the
application.xml of the
    // EJB deployment on the server.
    // Since we haven't deployed the application as a .ear, the app name for us will be an
empty string
    final String appName = "";
    // This is the module name of the deployed EJBs on the server. This is typically the jar
name of the
    // EJB deployment, without the .jar suffix, but can be overridden via the ejb-jar.xml
    // In this example, we have deployed the EJBs in a jboss-as-remote-app.jar, so the
module name is
    // jboss-as-remote-app
    final String moduleName = "jboss-as-remote-app";
    // AS7 allows each deployment to have an (optional) distinct name. We haven't specified
a distinct name for
    // our EJB deployment, so this is an empty string
    final String distinctName = "";
    // The EJB name which by default is the simple class name of the bean implementation
```



```
class
    final String beanName = CalculatorBean.class.getSimpleName();
    // the remote view fully qualified class name
    final String viewClassName = RemoteCalculator.class.getName();
    // let's do the lookup
    return (RemoteCalculator) context.lookup("ejb:" + appName + "/" + moduleName + "/" +
distinctName + "/" + beanName + "!" + viewClassName);
}

/**
 * Looks up and returns the proxy to remote stateful counter bean
 *
 * @return
 * @throws NamingException
 */
private static RemoteCounter lookupRemoteStatefulCounter() throws NamingException {
    final Hashtable jndiProperties = new Hashtable();
    jndiProperties.put(Context.URL_PKG_PREFIXES, "org.jboss.ejb.client.naming");
    final Context context = new InitialContext(jndiProperties);
    // The app name is the application name of the deployed EJBs. This is typically the ear
name
    // without the .ear suffix. However, the application name could be overridden in the
application.xml of the
    // EJB deployment on the server.
    // Since we haven't deployed the application as a .ear, the app name for us will be an
empty string
    final String appName = "";
    // This is the module name of the deployed EJBs on the server. This is typically the jar
name of the
    // EJB deployment, without the .jar suffix, but can be overridden via the ejb-jar.xml
    // In this example, we have deployed the EJBs in a jboss-as-ejb-remote-app.jar, so the
module name is
    // jboss-as-ejb-remote-app
    final String moduleName = "jboss-as-ejb-remote-app";
    // AS7 allows each deployment to have an (optional) distinct name. We haven't specified
a distinct name for
    // our EJB deployment, so this is an empty string
    final String distinctName = "";
    // The EJB name which by default is the simple class name of the bean implementation
class
    final String beanName = CounterBean.class.getSimpleName();
    // the remote view fully qualified class name
    final String viewClassName = RemoteCounter.class.getName();
    // let's do the lookup (notice the ?stateful string as the last part of the jndi name
for stateful bean lookup)
    return (RemoteCounter) context.lookup("ejb:" + appName + "/" + moduleName + "/" +
distinctName + "/" + beanName + "!" + viewClassName + "?stateful");
}
}
```



The entire server side and client side code is hosted at the github repo here [ejb-remote](#)



The code has some comments which will help you understand each of those lines. But we'll explain here in more detail what the code does. As a first step in the client code, we'll do a lookup of the EJB using a JNDI name. In AS7, for remote access to EJBs, you use the `ejb:` namespace with the following syntax:

For stateless beans:

```
ejb:<app-name>/<module-name>/<distinct-name>/<bean-name>!<fully-qualified-classname-of-the-remote-
```

For stateful beans:

```
ejb:<app-name>/<module-name>/<distinct-name>/<bean-name>!<fully-qualified-classname-of-the-remote-
```

The `ejb:` namespace identifies it as a EJB lookup and is a constant (i.e. doesn't change) for doing EJB lookups. The rest of the parts in the jndi name are as follows:

app-name : This is the name of the `.ear` (without the `.ear` suffix) that you have deployed on the server and contains your EJBs.

- Java EE 6 allows you to override the application name, to a name of your choice by setting it in the `application.xml`. If the deployment uses such an override then the `app-name` used in the JNDI name should match that name.
- EJBs can also be deployed in a `.war` or a plain `.jar` (like we did in step 1). In such cases where the deployment isn't an `.ear` file, then the `app-name` must be an empty string, while doing the lookup.

module-name : This is the name of the `.jar` (without the `.jar` suffix) that you have deployed on the server and the contains your EJBs. If the EJBs are deployed in a `.war` then the module name is the `.war` name (without the `.war` suffix).

- Java EE 6 allows you to override the module name, by setting it in the `ejb-jar.xml/web.xml` of your deployment. If the deployment uses such an override then the `module-name` used in the JNDI name should match that name.
- Module name part cannot be an empty string in the JNDI name

distinct-name : This is a WildFly-specific name which can be optionally assigned to the deployments that are deployed on the server. More about the purpose and usage of this will be explained in a separate chapter. If a deployment doesn't use `distinct-name` then, use an empty string in the JNDI name, for `distinct-name`

bean-name : This is the name of the bean for which you are doing the lookup. The bean name is typically the unqualified classname of the bean implementation class, but can be overridden through either `ejb-jar.xml` or via annotations. The bean name part cannot be an empty string in the JNDI name.

fully-qualified-classname-of-the-remote-interface : This is the fully qualified class name of the interface for which you are doing the lookup. The interface should be one of the remote interfaces exposed by the bean on the server. The fully qualified class name part cannot be an empty string in the JNDI name.



For stateful beans, the JNDI name expects an additional "?stateful" to be appended after the fully qualified interface name part. This is because for stateful beans, a new session gets created on JNDI lookup and the EJB client API implementation doesn't contact the server during the JNDI lookup to know what kind of a bean the JNDI name represents (we'll come to this in a while). So the JNDI name itself is expected to indicate that the client is looking up a stateful bean, so that an appropriate session can be created.

Now that we know the syntax, let's see our code and check what JNDI name it uses. Since our stateless EJB named CalculatorBean is deployed in a jboss-as-ejb-remote-app.jar (without any ear) and since we are looking up the org.jboss.as.quickstarts.ejb.remote.stateless.RemoteCalculator remote interface, our JNDI name will be:

```
ejb:/jboss-as-ejb-remote-app//CalculatorBean!org.jboss.as.quickstarts.ejb.remote.stateless.RemoteC
```

That's what the lookupRemoteStatelessCalculator() method in the above client code uses.

For the stateful EJB named CounterBean which is deployed in the same jboss-as-ejb-remote-app.jar and which exposes the org.jboss.as.quickstarts.ejb.remote.stateful.RemoteCounter, the JNDI name will be:

```
ejb:/jboss-as-ejb-remote-app//CounterBean!org.jboss.as.quickstarts.ejb.remote.stateful.RemoteCount
```

That's what the lookupRemoteStatefulCounter() method in the above client code uses.

Now that we know of the JNDI name, let's take a look at the following piece of code in the lookupRemoteStatelessCalculator():

```
final Hashtable jndiProperties = new Hashtable();
jndiProperties.put(Context.URL_PKG_PREFIXES, "org.jboss.ejb.client.naming");
final Context context = new InitialContext(jndiProperties);
```

Here we are creating a JNDI InitialContext object by passing it some JNDI properties. The Context.URL_PKG_PREFIXES is set to org.jboss.ejb.client.naming. This is necessary because we should let the JNDI API know what handles the ejb: namespace that we use in our JNDI names for lookup. The "org.jboss.ejb.client.naming" has a URLContextFactory implementation which will be used by the JNDI APIs to parse and return an object for ejb: namespace lookups. You can either pass these properties to the constructor of the InitialContext class or have a jndi.properties file in the classpath of the client application, which (atleast) contains the following property:

```
java.naming.factory.url.pkgs=org.jboss.ejb.client.naming
```

So at this point, we have setup the InitialContext and also have the JNDI name ready to do the lookup. You can now do the lookup and the appropriate proxy which will be castable to the remote interface that you used as the fully qualified class name in the JNDI name, will be returned. Some of you might be wondering, how the JNDI implementation knew which server address to look, for your deployed EJBs. The answer is in AS7, the proxies returned via JNDI name lookup for ejb: namespace do not connect to the server unless an invocation on those proxies is done.



Now let's get to the point where we invoke on this returned proxy:

```
// Let's lookup the remote stateless calculator
    final RemoteCalculator statelessRemoteCalculator = lookupRemoteStatelessCalculator();
    System.out.println("Obtained a remote stateless calculator for invocation");
    // invoke on the remote calculator
    int a = 204;
    int b = 340;
    System.out.println("Adding " + a + " and " + b + " via the remote stateless calculator
    deployed on the server");
    int sum = statelessRemoteCalculator.add(a, b);
```

We can see here that the proxy returned after the lookup is used to invoke the `add(...)` method of the bean. It's at this point that the JNDI implementation (which is backed by the EJB client API) needs to know the server details. So let's now get to the important part of setting up the EJB client context properties.

5.3 Setting up EJB client context properties

A EJB client context is a context which contains contextual information for carrying out remote invocations on EJBs. This is a WildFly-specific API. The EJB client context can be associated with multiple EJB receivers. Each EJB receiver is capable of handling invocations on different EJBs. For example, an EJB receiver "Foo" might be able to handle invocation on a bean identified by `app-A/module-A/distinctinctName-A/Bar!RemoteBar`, whereas a EJB receiver named "Blah" might be able to handle invocation on a bean identified by `app-B/module-B/distinctName-B/BeanB!RemoteBean`. Each such EJB receiver knows about what set of EJBs it can handle and each of the EJB receiver knows which server target to use for handling the invocations on the bean. For example, if you have a AS7 server at 10.20.30.40 IP address which has its remoting port opened at 4447 and if that's the server on which you deployed that `CalculatorBean`, then you can setup a EJB receiver which knows its target address is 10.20.30.40:4447. Such an EJB receiver will be capable enough to communicate to the server via the JBoss specific EJB remote client protocol (details of which will be explained in-depth in a separate chapter).

Now that we know what a EJB client context is and what a EJB receiver is, let's see how we can setup a client context with 1 EJB receiver which can connect to 10.20.30.40 IP address at port 4447. That EJB client context will then be used (internally) by the JNDI implementation to handle invocations on the bean proxy.

The client will have to place a `jboss-ejb-client.properties` file in the classpath of the application. The `jboss-ejb-client.properties` can contain the following properties:




```
endpoint.name=client-endpoint
remote.connectionprovider.create.options.org.xnio.Options.SSL_ENABLED=false


remote.connections=default

remote.connection.default.host=10.20.30.40
remote.connection.default.port = 8080
remote.connection.default.connect.options.org.xnio.Options.SASL_POLICY_NOANONYMOUS=false

remote.connection.default.username=appuser
remote.connection.default.password=apppassword
```

 This file includes a reference to a default password. Be sure to change this as soon as possible.

The above properties file is just an example. The actual file that was used for this sample program is available here for reference [jboss-ejb-client.properties](#)

 We'll see what each of it means.

First the `endpoint.name` property. We mentioned earlier that the EJB receivers will communicate with the server for EJB invocations. Internally, they use JBoss Remoting project to carry out the communication. The `endpoint.name` property represents the name that will be used to create the client side of the endpoint. The `endpoint.name` property is optional and if not specified in the `jboss-ejb-client.properties` file, it will default to "config-based-ejb-client-endpoint" name.

Next is the `remote.connectionprovider.create.options.<...>` properties:

```
remote.connectionprovider.create.options.org.xnio.Options.SSL_ENABLED=false
```

The "remote.connectionprovider.create.options." property prefix can be used to pass the options that will be used while create the connection provider which will handle the "remote:" protocol. In this example we use the "remote.connectionprovider.create.options." property prefix to pass the "org.xnio.Options.SSL_ENABLED" property value as false. That property will then be used during the connection provider creation. Similarly other properties can be passed too, just append it to the "remote.connectionprovider.create.options." prefix

Next we'll see:

```
remote.connections=default
```



This is where you define the connections that you want to setup for communication with the remote server. The "remote.connections" property uses a comma separated value of connection "names". The connection names are just logical and are used grouping together the connection configuration properties later on in the properties file. The example above sets up a single remote connection named "default". There can be more than one connections that are configured. For example:

```
remote.connections=one, two
```

Here we are listing 2 connections named "one" and "two". Ultimately, each of the connections will map to a EJB receiver. So if you have 2 connections, that will setup 2 EJB receivers that will be added to the EJB client context. Each of these connections will be configured with the connection specific properties as follows:

```
remote.connection.default.host=10.20.30.40
remote.connection.default.port = 8080
remote.connection.default.connect.options.org.xnio.Options.SASL_POLICY_NOANONYMOUS=false
```

As you can see we are using the "remote.connection.<connection-name>." prefix for specifying the connection specific property. The connection name here is "default" and we are setting the "host" property of that connection to point to 10.20.30.40. Similarly we set the "port" for that connection to 4447.

By default WildFly uses 8080 as the remoting port. The EJB client API uses the http port, with the http-upgrade functionality, for communicating with the server for remote invocations, so that's the port we use in our client programs (unless the server is configured for some other http port)



```
remote.connection.default.username=appuser
remote.connection.default.password=apppassword
```

The given user/password must be set by using the command bin/add-user.sh (or.bat). The user and password must be set because the security-realm is enabled for the subsystem remoting (see standalone*.xml or domain.xml) by default. If you do not need the security for remoting you might remove the attribute security-realm in the configuration.

security-realm is enabled by default.



We then use the "remote.connection.<connection-name>.connect.options." property prefix to setup options that will be used during the connection creation.

Here's an example of setting up multiple connections with different properties for each of those:

```
remote.connectionprovider.create.options.org.xnio.Options.SSL_ENABLED=false

remote.connections=one, two

remote.connection.one.host=localhost
remote.connection.one.port=6999
remote.connection.one.connect.options.org.xnio.Options.SASL_POLICY_NOANONYMOUS=false

remote.connection.two.host=localhost
remote.connection.two.port=7999
remote.connection.two.connect.options.org.xnio.Options.SASL_POLICY_NOANONYMOUS=false
```

As you can see we setup 2 connections "one" and "two" which both point to "localhost" as the "host" but different ports. Each of these connections will internally be used to create the EJB receivers in the EJB client context.

So that's how the `jboss-ejb-client.properties` file can be setup and placed in the classpath.

Using a different file for setting up EJB client context

The EJB client code will by default look for `jboss-ejb-client.properties` in the classpath. However, you can specify a different file of your choice by setting the "jboss.ejb.client.properties.file.path" system property which points to a properties file on your filesystem, containing the client context configurations. An example for that would be

```
"-Djboss.ejb.client.properties.file.path=/home/me/my-client/custom-jboss-ejb-client.properties"
```

Setting up the client classpath with the jars that are required to run the client application

A `jboss-client` jar is shipped in the distribution. It's available at `WILDFLY_HOME/bin/client/jboss-client.jar`. Place this jar in the classpath of your client application.

If you are using Maven to build the client application, then please follow the instructions in the `WILDFLY_HOME/bin/client/README.txt` to add this jar as a Maven dependency.





5.4 Summary

In the above examples, we saw what it takes to invoke a EJB from a remote client. To summarize:


- On the server side you need to deploy EJBs which expose the remote views.
- On the client side you need a client program which:
 - Has a `jboss-ejb-client.properties` in its classpath to setup the server connection information
 - Either has a `jndi.properties` to specify the `java.naming.factory.url.pkgs` property or passes that as a property to the `InitialContext` constructor
 - Setup the client classpath to include the `jboss-client` jar that's required for remote invocation of the EJBs. The location of the jar is mentioned above. You'll also need to have your application's bean interface jars and other jars that are required by your application, in the client classpath



6 EJB invocations from a remote server

The purpose of this chapter is to demonstrate how to lookup and invoke on EJBs deployed on an WildFly server instance **from another** WildFly server instance. This is different from invoking the EJBs [from a remote standalone client](#)


Let's call the server, from which the invocation happens to the EJB, as "Client Server" and the server on which the bean is deployed as the "Destination Server".

 Note that this chapter deals with the case where the bean is deployed on the "Destination Server" but **not** on the "Client Server".

6.1 Application packaging

In this example, we'll consider a EJB which is packaged in a `myejb.jar` which is within a `myapp.ear`. Here's how it would look like:

```
myapp.ear
|
|---- myejb.jar
|      |
|      |---- <org.myapp.ejb.*> // EJB classes
```

 Note that packaging itself isn't really important in the context of this article. You can deploy the EJBs in any standard way (`.ear`, `.war` or `.jar`).



6.2 Beans

In our example, we'll consider a simple stateless session bean which is as follows:

```
package org.myapp.ejb;

public interface Greeter {

    String greet(String user);
}
```

```
package org.myapp.ejb;

import javax.ejb.Remote;
import javax.ejb.Stateless;

@Stateless
@Remote (Greeter.class)
public class GreeterBean implements Greeter {

    @Override
    public String greet(String user) {
        return "Hello " + user + ", have a pleasant day!";
    }
}
```

6.3 Security

WildFly 8 is secure by default. What this means is that no communication can happen with an WildFly instance from a remote client (irrespective of whether it is a standalone client or another server instance) without passing the appropriate credentials. Remember that in this example, our "client server" will be communicating with the "destination server". So in order to allow this communication to happen successfully, we'll have to configure user credentials which we will be using during this communication. So let's start with the necessary configurations for this.



6.4 Configuring a user on the "Destination Server"

As a first step we'll configure a user on the destination server who will be allowed to access the destination server. We create the user using the `add-user` script that's available in the `JBOSS_HOME/bin` folder. In this example, we'll be configuring a `Application` User named `ejb` and with a password `test` in the `ApplicationRealm`. Running the `add-user` script is an interactive process and you will see questions/output as follows:


add-user


```
jpai@jpai-laptop:bin$ ./add-user.sh

What type of user do you wish to add?
&nbsp;a) Management User (mgmt-users.properties)
&nbsp;b) Application User (application-users.properties)
(a): b

Enter the details of the new user to add.
Realm (ApplicationRealm) :
Username : ejb
Password :
Re-enter Password :
What roles do you want this user to belong to? (Please enter a comma separated list, or leave
blank for none)\[&nbsp; \]:
About to add user 'ejb' for realm 'ApplicationRealm'
Is this correct yes/no? yes
Added user 'ejb' to file
'/jboss-as-7.1.1.Final/standalone/configuration/application-users.properties'
Added user 'ejb' to file
'/jboss-as-7.1.1.Final/domain/configuration/application-users.properties'
Added user 'ejb' with roles to file
'/jboss-as-7.1.1.Final/standalone/configuration/application-roles.properties'
Added user 'ejb' with roles to file
'/jboss-as-7.1.1.Final/domain/configuration/application-roles.properties'
```

As you can see in the output above we have now configured a user on the destination server who'll be allowed to access this server. We'll use this user credentials later on in the client server for communicating with this server. The important bits to remember are the user we have created in this example is `ejb` and the password is `test`.

 Note that you can use any username and password combination you want to.

 You do **not** require the server to be started to add a user using the `add-user` script.



6.5 Start the "Destination Server"

As a next step towards running this example, we'll start the "Destination Server". In this example, we'll use the standalone server and use the *standalone-full.xml* configuration. The startup command will look like:

```
./standalone.sh -server-config=standalone-full.xml
```

Ensure that the server has started without any errors.



It's very important to note that if you are starting both the server instances on the same machine, then each of those server instances **must** have a unique `jboss.node.name` system property. You can do that by passing an appropriate value for `-Djboss.node.name` system property to the startup script:

```
./standalone.sh -server-config=standalone-full.xml -Djboss.node.name=<add appropriate value here>
```

6.6 Deploying the application

The application (*myapp.ear* in our case) will be deployed to "Destination Server". The process of deploying the application is out of scope of this chapter. You can either use the Command Line Interface or the Admin console or any IDE or manually copy it to `JBOSS_HOME/standalone/deployments` folder (for standalone server). Just ensure that the application has been deployed successfully.

So far, we have built a EJB application and deployed it on the "Destination Server". Now let's move to the "Client Server" which acts as the client for the deployed EJBs on the "Destination Server".

6.7 Configuring the "Client Server" to point to the EJB remoting connector on the "Destination Server"

As a first step on the "Client Server", we need to let the server know about the "Destination Server"'s EJB remoting connector, over which it can communicate during the EJB invocations. To do that, we'll have to add a *remote-outbound-connection* to the remoting subsystem on the "Client Server". The *remote-outbound-connection* configuration indicates that a outbound connection will be created to a remote server instance from that server. The *remote-outbound-connection* will be backed by a *outbound-socket-binding* which will point to a remote host and a remote port (of the "Destination Server"). So let's see how we create these configurations.



6.8 Start the "Client Server"

In this example, we'll start the "Client Server" on the same machine as the "Destination Server". We have copied the entire server installation to a different folder and while starting the "Client Server" we'll use a port-offset (of 100 in this example) to avoid port conflicts:

```
./standalone.sh -server-config=standalone-full.xml -Djboss.socket.binding.port-offset=100
```

6.9 Create a security realm on the client server

Remember that we need to communicate with a secure destination server. In order to do that the client server has to pass the user credentials to the destination server. Earlier we created a user on the destination server who'll be allowed to communicate with that server. Now on the "client server" we'll create a security-realm which will be used to pass the user information.

In this example we'll use a security realm which stores a Base64 encoded password and then passes on that credentials when asked for. Earlier we created a user named `ejb` and password `test`. So our first task here would be to create the base64 encoded version of the password `test`. You can use any utility which generates you a base64 version for a string. I used [this online site](#) which generates the base64 encoded string. So for the `test` password, the base64 encoded version is `dGVzdA==`

- ✔ While generating the base64 encoded string make sure that you don't have any trailing or leading spaces for the original password. That can lead to incorrect encoded versions being generated.

- ✔ With new versions the add-user script will show the base64 password if you type 'y' if you've been ask


```
Is this new user going to be used for one AS process to connect to another AS process  
e.g. slave domain controller?
```

Now that we have generated that base64 encoded password, let's use in the in the security realm that we are going to configure on the "client server". I'll first shutdown the client server and edit the `standalone-full.xml` file to add the following in the `<management>` section

Now let's create a "*security-realm*" for the base64 encoded password.

```
/core-service=management/security-realm=ejb-security-realm:add()  
/core-service=management/security-realm=ejb-security-realm/server-identity=secret:add(value=dGVzdA==)
```



 Notice that the CLI show the message *"process-state" => "reload-required"*, so you have to restart the server before you can use this change.

upon successful invocation of this command, the following configuration will be created in the *management* section:

standalone-full.xml

```
<management>
  <security-realms>
    ...
    <security-realm name="ejb-security-realm">
      <server-identities>
        <secret value="dGVzdA==" />
      </server-identities>
    </security-realm>
  </security-realms>
  ...

```

As you can see I have created a security realm named "ejb-security-realm" (you can name it anything) with the base64 encoded password. So that completes the security realm configuration for the client server. Now let's move on to the next step.



6.10 Create a outbound-socket-binding on the "Client Server"

Let's first create a *outbound-socket-binding* which points the "Destination Server"'s host and port. We'll use the CLI to create this configuration:

```
/socket-binding-group=standard-sockets/remote-destination-outbound-socket-binding=remote-ejb:add(host=localhost, port=8080)
```

The above command will create a outbound-socket-binding named "*remote-ejb*" (we can name it anything) which points to "localhost" as the host and port 8080 as the destination port. Note that the host information should match the host/IP of the "Destination Server" (in this example we are running on the same machine so we use "localhost") and the port information should match the http-remoting connector port used by the EJB subsystem (by default it's 8080). When this command is run successfully, we'll see that the standalone-full.xml (the file which we used to start the server) was updated with the following outbound-socket-binding in the socket-binding-group:

```
<socket-binding-group name="standard-sockets" default-interface="public"
port-offset="{jboss.socket.binding.port-offset:0}">
  ...
  <outbound-socket-binding name="remote-ejb">
    <remote-destination host="localhost" port="8080"/>
  </outbound-socket-binding>
</socket-binding-group>
```

6.11 Create a "remote-outbound-connection" which uses this newly created "outbound-socket-binding"

Now let's create a "*remote-outbound-connection*" which will use the newly created outbound-socket-binding (pointing to the EJB remoting connector of the "Destination Server"). We'll continue to use the CLI to create this configuration:

```
/subsystem=remoting/remote-outbound-connection=remote-ejb-connection:add(outbound-socket-binding-ref=remote-ejb, protocol=http-remoting, security-realm=ejb-security-realm, username=ejb)
```

The above command creates a remote-outbound-connection, named "*remote-ejb-connection*" (we can name it anything), in the remoting subsystem and uses the previously created "*remote-ejb*" outbound-socket-binding (notice the outbound-socket-binding-ref in that command) with the http-remoting protocol. Furthermore, we also set the security-realm attribute to point to the security-realm that we created in the previous step. Also notice that we have set the username attribute to use the user name who is allowed to communicate with the destination server.



What this step does is, it creates a outbound connection, on the client server, to the remote destination server and sets up the username to the user who allowed to communicate with that destination server and also sets up the security-realm to a pre-configured security-realm capable of passing along the user credentials (in this case the password). This way when a connection has to be established from the client server to the destination server, the connection creation logic will have the necessary security credentials to pass along and setup a successful secured connection.

Now let's run the following two operations to set some default connection creation options for the outbound connection:

```
/subsystem=remoting/remote-outbound-connection=remote-ejb-connection/property=SASL_POLICY_NOANONYM
```

```
/subsystem=remoting/remote-outbound-connection=remote-ejb-connection/property=SSL_ENABLED:add(valu
```

Ultimately, upon successful invocation of this command, the following configuration will be created in the remoting subsystem:

```
<subsystem xmlns="urn:jboss:domain:remoting:1.1">
  ....
  <outbound-connections>
    <remote-outbound-connection name="remote-ejb-connection"
outbound-socket-binding-ref="remote-ejb" protocol="http-remoting"
security-realm="ejb-security-realm" username="ejb">
      <properties>
        <property name="SASL_POLICY_NOANONYMOUS" value="false"/>
        <property name="SSL_ENABLED" value="false"/>
      </properties>
    </remote-outbound-connection>
  </outbound-connections>
</subsystem>
```

From a server configuration point of view, that's all we need on the "Client Server". Our next step is to deploy an application on the "Client Server" which will invoke on the bean deployed on the "Destination Server".




6.12 Packaging the client application on the "Client Server"

Like on the "Destination Server", we'll use .ear packaging for the client application too. But like previously mentioned, that's not mandatory. You can even use a .war or .jar deployments. Here's how our client application packaging will look like:

```
client-app.ear
|
|--- META-INF
|       |
|       |--- jboss-ejb-client.xml
|
|--- web.war
|       |
|       |--- WEB-INF/classes
|               |
|               |---- <org.myapp.FooServlet> // classes in the web app
```

In the client application we'll use a servlet which invokes on the bean deployed on the "Destination Server". We can even invoke the bean on the "Destination Server" from a EJB on the "Client Server". The code remains the same (JNDI lookup, followed by invocation on the proxy). The important part to notice in this client application is the file *jboss-ejb-client.xml* which is packaged in the META-INF folder of a top level deployment (in this case our client-app.ear). This *jboss-ejb-client.xml* contains the EJB client configurations which will be used during the EJB invocations for finding the appropriate destinations (also known as, EJB receivers). The contents of the *jboss-ejb-client.xml* are explained next.

 If your application is deployed as a top level .war deployment, then the *jboss-ejb-client.xml* is expected to be placed in .war/WEB-INF/ folder (i.e. the same location where you place any web.xml file).



6.13 Contents on jboss-ejb-client.xml

The jboss-ejb-client.xml will look like:

```
<jboss-ejb-client xmlns="urn:jboss:ejb-client:1.0">
  <client-context>
    <ejb-receivers>
      <remoting-ejb-receiver outbound-connection-ref="remote-ejb-connection"/>
    </ejb-receivers>
  </client-context>
</jboss-ejb-client>
```

You'll notice that we have configured the EJB client context (for this application) to use a `remoting-ejb-receiver` which points to our earlier created `remote-outbound-connection` named `remote-ejb-connection`. This links the EJB client context to use the `remote-ejb-connection` which ultimately points to the EJB remoting connector on the "Destination Server".

6.14 Deploy the client application

Let's deploy the client application on the "Client Server". The process of deploying the application is out of scope, of this chapter. You can use either the CLI or the admin console or a IDE or deploy manually to `JBOSS_HOME/standalone/deployments` folder. Just ensure that the application is deployed successfully.



6.15 Client code invoking the bean

We mentioned that we'll be using a servlet to invoke on the bean, but the code to invoke the bean isn't servlet specific and can be used in other components (like EJB) too. So let's see how it looks like:

```
import javax.naming.Context;
import java.util.Hashtable;
import javax.naming.InitialContext;

...
public void invokeOnBean() {
    try {
        final Hashtable props = new Hashtable();
        // setup the ejb: namespace URL factory
        props.put(Context.URL_PKG_PREFIXES, "org.jboss.ejb.client.naming");
        // create the InitialContext
        final Context context = new javax.naming.InitialContext(props);

        // Lookup the Greeter bean using the ejb: namespace syntax which is explained here
https://docs.jboss.org/author/display/AS71/EJB+invocations+from+a+remote+client+using+JNDI
        final Greeter bean = (Greeter) context.lookup("ejb:" + "myapp" + "/" + "myejb" + "/"
+ "" + "/" + "GreeterBean" + "!" + org.myapp.ejb.Greeter.class.getName());

        // invoke on the bean
        final String greeting = bean.greet("Tom");

        System.out.println("Received greeting: " + greeting);

    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

That's it! The above code will invoke on the bean deployed on the "Destination Server" and return the result.



7 Remote EJB invocations via JNDI - Which approach to use?

Couldn't find a page to include called: Remote EJB invocations via JNDI - EJB client API or remote-naming project?



8 JBoss EJB 3 reference guide

This chapter details the extensions that are available when developing Enterprise Java Beans™ on WildFly 8.

Currently there is no support for configuring the extensions using an implementation specific descriptor file.

8.1 Resource Adapter for Message Driven Beans

Each Message Driven Bean must be connected to a resource adapter.

8.1.1 Specification of Resource Adapter using Metadata Annotations

The `ResourceAdapter` annotation is used to specify the resource adapter with which the MDB should connect.

The `value` of the annotation is the name of the deployment unit containing the resource adapter. For example `jms-ra.rar`.

For example:

```
@MessageDriven(messageListenerInterface = PostmanPat.class)
@ResourceAdapter("ejb3-rar.rar")
```



8.2 as Principal

Whenever a run-as role is specified for a given method invocation the default anonymous principal is used as the caller principal. This principal can be overridden by specifying a run-as principal.

8.2.1 Specification of Run-as Principal using Metadata Annotations

The `RunAsPrincipal` annotation is used to specify the run-as principal to use for a given method invocation.

The value of the annotation specifies the name of the principal to use. The actual type of the principal is undefined and should not be relied upon.

Using this annotation without specifying a run-as role is considered an error.

For example:

```
@RunAs("admin")
@RunAsPrincipal("MyBean")
```

8.3 Security Domain

Each Enterprise Java Bean [™] can be associated with a security domain. Only when an EJB is associated with a security domain will authentication and authorization be enforced.

8.3.1 Specification of Security Domain using Metadata Annotations

The `SecurityDomain` annotation is used to specify the security domain to associate with the EJB.

The value of the annotation is the name of the security domain to be used.

For example:

```
@SecurityDomain("other")
```

8.4 Transaction Timeout

For any newly started transaction a transaction timeout can be specified in seconds.



When a transaction timeout of 0 is used, then the actual transaction timeout will default to the domain configured default.

TODO: add link to tx subsystem

Although this is only applicable when using transaction attribute `REQUIRED` or `REQUIRES_NEW` the application server will not detect invalid setups.



New Transactions

Take care that even when transaction attribute `REQUIRED` is specified, the timeout will only be applicable if a **new** transaction is started.

8.4.1 Specification of Transaction Timeout with Metadata Annotations

The `TransactionTimeout` annotation is used to specify the transaction timeout for a given method.

The `value` of the annotation is the timeout used in the given `unit` granularity. It must be a positive integer or 0. Whenever 0 is specified the default domain configured timeout is used.

The `unit` specifies the granularity of the `value`. The actual value used is converted to seconds. Specifying a granularity lower than `SECONDS` is considered an error, even when the computed value will result in an even amount of seconds.

For example: `@TransactionTimeout(value = 10, unit = TimeUnit.SECONDS)`



8.4.2 Specification of Transaction Timeout in the Deployment Descriptor

The `trans-timeout` element is used to define the transaction timeout for business, home, component, and message-listener interface methods; no-interface view methods; web service endpoint methods; and timeout callback methods.

The `trans-timeout` element resides in the `urn:trans-timeout` namespace and is part of the standard `container-transaction` element as defined in the `jboss` namespace.

For the rules when a `container-transaction` is applicable please refer to EJB 3.1 FR 13.3.7.2.1.

Example of `trans-timeout`

`jboss-ejb3.xml`

```
<jboss:ejb-jar xmlns:jboss="http://www.jboss.com/xml/ns/javaee"
              xmlns="http://java.sun.com/xml/ns/javaee"
              xmlns:tx="urn:trans-timeout"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xsi:schemaLocation="http://www.jboss.com/xml/ns/javaee
http://www.jboss.org/j2ee/schema/jboss-ejb3-2_0.xsd
http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/ejb-jar_3_1.xsd
urn:trans-timeout http://www.jboss.org/j2ee/schema/trans-timeout-1_0.xsd"
              version="3.1"
              impl-version="2.0">
  <assembly-descriptor>
    <container-transaction>
      <method>
        <ejb-name>BeanWithTimeoutValue</ejb-name>
        <method-name>*</method-name>
        <method-intf>Local</method-intf>
      </method>
      <tx:trans-timeout>
        <tx:timeout>10</tx:timeout>
        <tx:unit>Seconds</tx:unit>
      </tx:trans-timeout>
    </container-transaction>
  </assembly-descriptor>
</jboss:ejb-jar>
```

8.5 Timer service

The service is responsible to call the registered timeout methods of the different session beans.



i A persistent timer will be identified by the name of the EAR, the name of the sub-deployment JAR and the Bean's name.
If one of those names are changed (e.g. EAR name contain a version) the timer entry became orphaned and the timer event will not longer be fired.

8.5.1 Single event timer

The timer is will be started once at the specified time.

In case of a server restart the timeout method of a persistent timer will only be called directly if the specified time is elapsed.

If the timer is not persistent (since EJB3.1 see 18.2.3) it will be not longer available if JBoss is restarted or the application is redeployed.

8.5.2 Recurring timer

The timer will be started at the specified first occurrence and after that point at each time if the interval is elapsed.

If the timer will be started during the last execution is not finished the execution will be suppressed with a warning to avoid concurrent execution.

In case of server downtime for a persistent timer, the timeout method will be called only once if one, or more than one, interval is elapsed.

If the timer is not persistent (since EJB3.1 see 18.2.3) it will not longer be active after the server is restarted or the application is redeployed.



8.5.3 Calendar timer

The timer will be started if the schedule expression match. It will be automatically deactivated and removed if there will be no next expiration possible, i.e. If you set a specific year.

For example:

```
@Schedule( ... dayOfMonth="1", month="1", year="2012")  
// start once at 01-01-2012 00:00:00
```

Programmatic calendar timer

If the timer is persistent it will be fetched at server start and the missed timeouts are called concurrent.

If a persistent timer contains an end date it will be executed once nevertheless how many times the execution was missed. Also a retry will be suppressed if the timeout method throw an Exception.

In case of such expired timer access to the given Timer object might throw a NoMoreTimeoutException or NoSuchObjectException.

If the timer is non persistent it will not longer be active after the server is restarted or the application is redeployed.

TODO: clarify whether this should happen concurrently/blocked or even fired only once like a recurring timer!

Annotated calendar timer

If the timer is non persistent it will not activated for missed events during the server is down. In case of server start the timer is scheduled based on the @Schedule annotation.

If the timer is persistent (default if not deactivated by annotation) all missed events are fetched at server start and the annotated timeout method is called concurrent.

TODO: clarify whether this should happen concurrently/blocked or even fired only once like a recurring timer!



9 JPA reference guide

- [Introduction](#)
- [Update your Persistence.xml for Hibernate 5.0](#)
- [Entity manager](#)
- [Application-managed entity manager](#)
- [Container-managed entity manager](#)
- [Persistence Context](#)
- [Transaction-scoped Persistence Context](#)
- [Extended Persistence Context](#)
 - [Extended Persistence Context Inheritance](#)
- [Entities](#)
- [Deployment](#)
- [Troubleshooting](#)
- [Using the Hibernate 5.x JPA persistence provider](#)
- [Hibernate ORM 3.x integration is not included](#)
- [Using the Infinispan second level cache](#)
- [Replacing the current Hibernate 5.x jars with a newer version](#)
- [Using Hibernate Search](#)
- [Packaging the Hibernate JPA persistence provider with your application](#)
- [Using OpenJPA](#)
- [Using EclipseLink](#)
- [Using DataNucleus](#)
- [Native Hibernate use](#)
- [Injection of Hibernate Session and SessionFactory](#)
- [Injection of Hibernate Session and SessionFactory](#)
- [Hibernate properties](#)
- [Persistence unit properties](#)
- [Determine the persistence provider module](#)
- [Binding EntityManagerFactory/EntityManager to JNDI](#)
- [Community](#)
 - [People who have contributed to the WildFly JPA layer:](#)



9.1 Introduction

The WildFly JPA subsystem implements the JPA 2.1 container-managed requirements. Deploys the persistence unit definitions, the persistence unit/context annotations and persistence unit/context references in the deployment descriptor. JPA Applications use the Hibernate (version 5) persistence provider, which is included with WildFly. The JPA subsystem uses the standard SPI (`javax.persistence.spi.PersistenceProvider`) to access the Hibernate persistence provider and some additional extensions as well.

During application deployment, JPA use is detected (e.g. `persistence.xml` or `@PersistenceContext/Unit` annotations) and injects Hibernate dependencies into the application deployment. This makes it easy to deploy JPA applications.

In the remainder of this documentation, "entity manager" refers to an instance of the `javax.persistence.EntityManager` class. [Javadoc for the JPA interfaces](#) and [JPA 2.1 specification](#).

The index of the Hibernate documentation is at <http://hibernate.org/orm/documentation/5.0/>.

9.2 Update your Persistence.xml for Hibernate 5.0

The persistence provider class name in Hibernate 4.3.0 (and greater) is `org.hibernate.jpa.HibernatePersistenceProvider`.

Instead of specifying:

```
<provider>org.hibernate.ejb.HibernatePersistence</provider>
```

Switch to:

```
<provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
```

Or remove the persistence provider class name from your `persistence.xml` (so the default provider will be used).

9.3 Entity manager

The entity manager is similar to the Hibernate Session class; applications use it to create/read/update/delete data (and related operations). Applications can use application-managed or container-managed entity managers. Keep in mind that the entity manager is not expected to be thread safe (don't inject it into a servlet class variable which is visible to multiple threads).



9.4 Application-managed entity manager

Application-managed entity managers provide direct access to the underlying persistence provider (`org.hibernate.ejb.HibernatePersistence`). The scope of the application-managed entity manager is from when the application creates it and lasts until the app closes it. Use the `@PersistenceUnit` annotation to inject a persistence unit into a `javax.persistence.EntityManagerFactory`. The `EntityManagerFactory` can return an application-managed entity manager.

9.5 Container-managed entity manager

Container-managed entity managers auto-magically manage the underlying persistence provider for the application. Container-managed entity managers may use transaction-scoped persistence contexts or extended persistence contexts. The container-managed entity manager will create instances of the underlying persistence provider as needed. Every time that a new underlying persistence provider (`org.hibernate.ejb.HibernatePersistence`) instance is created, a new persistence context is also created (as an implementation detail of the underlying persistence provider).

9.6 Persistence Context

The JPA persistence context contains the entities managed by the persistence provider. The persistence context acts like a first level (transactional) cache for interacting with the datasource. Loaded entities are placed into the persistence context before being returned to the application. Entities changes are also placed into the persistence context (to be saved in the database when the transaction commits).



9.7 Transaction-scoped Persistence Context

The transaction-scoped persistence context coordinates with the (active) JTA transaction. When the transaction commits, the persistence context is flushed to the datasource (entity objects are detached but may still be referenced by application code). All entity changes that are expected to be saved to the datasource, must be made during a transaction. Entities read outside of a transaction will be detached when the entity manager invocation completes. Example transaction-scoped persistence context is below.

```
@Stateful // will use container managed transactions
public class CustomerManager {
    @PersistenceContext(unitName = "customerPU") // default type is
    PersistenceContextType.TRANSACTION
    EntityManager em;
    public customer createCustomer(String name, String address) {
        Customer customer = new Customer(name, address);
        em.persist(customer); // persist new Customer when JTA transaction completes (when method
ends).
        // internally:
        // 1. Look for existing "customerPU" persistence context in active
JTA transaction and use if found.
        // 2. Else create new "customerPU" persistence context (e.g.
instance of org.hibernate.ejb.HibernatePersistence)
        // and put in current active JTA transaction.
        return customer; // return Customer entity (will be detached from the persistence
context when caller gets control)
    } // Transaction.commit will be called, Customer entity will be persisted to the database and
"customerPU" persistence context closed
}
```

9.8 Extended Persistence Context

The (ee container managed) extended persistence context can span multiple transactions and allows data modifications to be queued up (like a shopping cart), without an active JTA transaction (to be applied during the next JTA TX). The Container-managed extended persistence context can only be injected into a stateful session bean.

```
@PersistenceContext(type = PersistenceContextType.EXTENDED, unitName = "inventoryPU")
EntityManager em;
```



9.8.1 Extended Persistence Context Inheritance

JPA 2.0 specification section 7.6.2.1

If a stateful session bean instantiates a stateful session bean (executing in the same EJB container instance) which also has such an extended persistence context, the extended persistence context of the first stateful session bean is inherited by the second stateful session bean and bound to it, and this rule recursively applies—independently of whether transactions are active or not at the point of the creation of the stateful session beans.

By default, the current stateful session bean being created, will (**deeply**) inherit the extended persistence context from any stateful session bean executing in the current Java thread. The **deep** inheritance of extended persistence context includes walking multiple levels up the stateful bean call stack (inheriting from parent beans). The **deep** inheritance of extended persistence context includes sibling beans. For example, parentA references child beans beanBwithXPC & beanCwithXPC. Even though parentA doesn't have an extended persistence context, beanBwithXPC & beanCwithXPC will share the same extended persistence context.

Some other EE application servers, use **shallow** inheritance, where stateful session bean only inherit from the parent stateful session bean (if there is a parent bean). Sibling beans do not share the same extended persistence context unless their (common) parent bean also has the same extended persistence context.

Applications can include a (top-level) **jboss-all.xml** deployment descriptor that specifies either the (default) **DEEP** extended persistence context inheritance or **SHALLOW**.

The WF/docs/schema/jboss-jpa_1_0.xsd describes the **jboss-jpa** deployment descriptor that may be included in the **jboss-all.xml**. Below is an example of using **SHALLOW** extended persistence context inheritance:

```
<jboss>
  <jboss-jpa xmlns="http://www.jboss.com/xml/ns/javaee">
    <extended-persistence inheritance="SHALLOW"/>
  </jboss-jpa>
</jboss>
```

Below is an example of using **DEEP** extended persistence inheritance:

```
<jboss>
  <jboss-jpa xmlns="http://www.jboss.com/xml/ns/javaee">
    <extended-persistence inheritance="DEEP"/>
  </jboss-jpa>
</jboss>
```




The AS console/cli can change the **default** extended persistence context setting (DEEP or SHALLOW). The following cli commands will read the current JPA settings and enable SHALLOW extended persistence context inheritance for applications that do not include the **jboss-jpa** deployment descriptor:

```
./jboss-cli.sh
cd subsystem=jpa
:read-resource
:write-attribute(name=default-extended-persistence-inheritance,value="SHALLOW")
```

9.9 Entities

JPA allows use of your (pojo) plain old Java class to represent a database table row.

```
@PersistenceContext EntityManager em;
Integer bomPk = getIndexKeyValue();
BillOfMaterials bom = em.find(BillOfMaterials.class, bomPk); // read existing table row into
BillOfMaterials class

BillOfMaterials createdBom = new BillOfMaterials("..."); // create new entity
em.persist(createdBom); // createdBom is now managed and will be saved to database when the
current JTA transaction completes
```

The entity lifecycle is managed by the underlying persistence provider.

- **New (transient):** an entity is new if it has just been instantiated using the new operator, and it is not associated with a persistence context. It has no persistent representation in the database and no identifier value has been assigned.
- **Managed (persistent):** a managed entity instance is an instance with a persistent identity that is currently associated with a persistence context.
- **Detached:** the entity instance is an instance with a persistent identity that is no longer associated with a persistence context, usually because the persistence context was closed or the instance was evicted from the context.
- **Removed:** a removed entity instance is an instance with a persistent identity, associated with a persistence context, but scheduled for removal from the database.



9.10 Deployment

The persistence.xml contains the persistence unit configuration (e.g. datasource name) and as described in the JPA 2.0 spec (section 8.2), the jar file or directory whose META-INF directory contains the persistence.xml file is termed the root of the persistence unit. In Java EE environments, the root of a persistence unit must be one of the following (quoted directly from the JPA 2.0 specification):

"

- an EJB-JAR file
- the WEB-INF/classes directory of a WAR file
- a jar file in the WEB-INF/lib directory of a WAR file
- a jar file in the EAR library directory
- an application client jar file

The persistence.xml can specify either a JTA datasource or a non-JTA datasource. The JTA datasource is expected to be used within the EE environment (even when reading data without an active transaction). If a datasource is not specified, the default-datasource will instead be used (must be configured).

NOTE: Java Persistence 1.0 supported use of a jar file in the root of the EAR as the root of a persistence unit. This use is no longer supported. Portable applications should use the EAR library directory for this case instead.

"

Question: Can you have a EAR/META-INF/persistence.xml?

Answer: No, the above may deploy but it could include other archives also in the EAR, so you may have deployment issues for other reasons. Better to put the persistence.xml in an EAR/lib/somePuJar.jar.

9.11 Troubleshooting

The **org.jboss.as.jpa** logging can be enabled to get the following information:

- INFO - when persistence.xml has been parsed, starting of persistence unit service (per deployed persistence.xml), stopping of persistence unit service
- DEBUG - informs about entity managers being injected, creating/reusing transaction scoped entity manager for active transaction
- TRACE - shows how long each entity manager operation took in milliseconds, application searches for a persistence unit, parsing of persistence.xml

To enable TRACE, open the as/standalone/configuration/standalone.xml (or as/domain/configuration/domain.xml) file. Search for **<subsystem xmlns="urn:jboss:domain:logging:1.0">** and add the **org.jboss.as.jpa** category. You need to change the console-handler level from **INFO** to **TRACE**.



```
<subsystem xmlns="urn:jboss:domain:logging:1.0">
  <console-handler name="CONSOLE">
    <level name="TRACE" />
    ...
  </console-handler>

  </periodic-rotating-file-handler>
  <logger category="com.arjuna">
    <level name="WARN" />
  </logger>

  <logger category="org.jboss.as.jpa">
    <level name="TRACE" />
  </logger>

  <logger category="org.apache.tomcat.util.modeler">
    <level name="WARN" />
  </logger>
  ...
</subsystem>
```

To see what is going on at the JDBC level, enable **jboss.jdbc.spy** TRACE and add `spy="true"` to the `datasource`.

```
<datasource jndi-name="java:jboss/datasources/..." pool-name="..." enabled="true" spy="true">
  <logger category="jboss.jdbc.spy">
    <level name="TRACE" />
  </logger>
</datasource>
```

To troubleshoot issues with the Hibernate second level cache, try enabling trace for **org.hibernate.SQL + org.hibernate.cache.infinispan + org.infinispan:**



```
<subsystem xmlns="urn:jboss:domain:logging:1.0">
  <console-handler name="CONSOLE">
    <level name="TRACE" />
    ...
  </console-handler>

  </periodic-rotating-file-handler>
  <logger category="com.arjuna">
    <level name="WARN" />
  </logger>

  <logger category="org.hibernate.SQL">
    <level name="TRACE" />
  </logger>

  <logger category="org.hibernate">
    <level name="TRACE" />
  </logger>
  <logger category="org.infinispan">
    <level name="TRACE" />
  </logger>

  <logger category="org.apache.tomcat.util.modeler">
    <level name="WARN" />
  </logger>
  ...
</subsystem>
```

9.12 Using the Hibernate 5.x JPA persistence provider

Hibernate 5.x is packaged with WildFly and is the default persistence provider.

9.13 Hibernate ORM 3.x integration is not included

The Hibernate 3.x integration is removed from WildFly, please use a newer version of Hibernate.

9.14 Using the Infinispan second level cache

To enable the second level cache with Hibernate 5.x, just set the **hibernate.cache.use_second_level_cache** property to true, as is done in the following example (also set the [shared-cache-mode](#) accordingly). By default the application server uses Infinispan as the cache provider for **JPA applications**, so you don't need specify anything on top of that. The Infinispan version that is included in WildFly is expected to work with the Hibernate version that is included with WildFly. Example persistence.xml settings:



```
<?xml version="1.0" encoding="UTF-8"?><persistence
xmlns="http://java.sun.com/xml/ns/persistence" version="1.0">
<persistence-unit name="2lc_example_pu">
  <description>example of enabling the second level cache.</description>
  <jta-data-source>java:jboss/datasources/mydatasource</jta-data-source>
  <shared-cache-mode>ENABLE_SELECTIVE</shared-cache-mode>
  <properties>
    <property name="hibernate.cache.use_second_level_cache" value="true"/>
  </properties>
</persistence-unit>
</persistence>
```

Here is an example of enabling the second level cache for a Hibernate native API hibernate.cfg.xml file:

```
<property name="hibernate.cache.region.factory_class"
value="org.jboss.as.jpa.hibernate5.infinispan.InfinispanRegionFactory"/>
<property name="hibernate.cache.infinispan.cachemanager"
value="java:jboss/infinispan/container/hibernate"/>
<property name="hibernate.transaction.manager_lookup_class"
value="org.hibernate.transaction.JBossTransactionManagerLookup"/>
<property name="hibernate.cache.use_second_level_cache" value="true"/>
```

The Hibernate native API application will also need a MANIFEST.MF:

```
Dependencies: org.infinispan,org.hibernate
```

[Infinispan Hibernate/JPA second level cache provider documentation](#) contains advanced configuration information but you should bear in mind that when Hibernate runs within WildFly 8, some of those configuration options, such as region factory, are not needed. Moreover, the application server providers you with option of selecting a different cache container for Infinispan via **hibernate.cache.infinispan.container** persistence property. To reiterate, this property is not mandatory and a default container is already deployed for by the application server to host the second level cache.

Here is an example of what the Hibernate cache settings may currently be in your standalone.xml:

```
<cache-container name="hibernate" default-cache="local-query" module="org.hibernate.infinispan">
  <local-cache name="entity">
    <transaction mode="NON_XA"/>
    <eviction strategy="LRU" max-entries="10000"/>
    <expiration max-idle="100000"/>
  </local-cache>
  <local-cache name="local-query">
    <eviction strategy="LRU" max-entries="10000"/>
    <expiration max-idle="100000"/>
  </local-cache>
  <local-cache name="timestamps"/>
</cache-container>
```



Below is an example of customizing the "entity", "immutable-entity", "local-query", "pending-puts", "timestamps" cache configuration may look like:

```
<cache-container name="hibernate" module="org.hibernate.infinispan"
default-cache="immutable-entity">
  <local-cache name="entity">
    <transaction mode="NONE" />
    <eviction max-entries="-1" />
    <expiration max-idle="120000" />
  </local-cache>
  <local-cache name="immutable-entity">
    <transaction mode="NONE" />
    <eviction max-entries="-1" />
    <expiration max-idle="120000" />
  </local-cache>
  <local-cache name="local-query">
    <eviction max-entries="-1" />
    <expiration max-idle="300000" />
  </local-cache>
  <local-cache name="pending-puts">
    <transaction mode="NONE" />
    <eviction strategy="NONE" />
    <expiration max-idle="60000" />
  </local-cache>
  <local-cache name="timestamps">
    <transaction mode="NONE" />
    <eviction strategy="NONE" />
  </local-cache>
</cache-container>
```

Persistence.xml to use the above custom settings:

```
<properties>
  <property name="hibernate.cache.use_second_level_cache" value="true" />
  <property name="hibernate.cache.use_query_cache" value="true" />
  <property name="hibernate.cache.infinispan.immutable-entity.cfg" value="immutable-entity" />
  <property name="hibernate.cache.infinispan.timestamps.cfg" value="timestamps" />
  <property name="hibernate.cache.infinispan.pending-puts.cfg" value="pending-puts" />
</properties>
```



9.15 Replacing the current Hibernate 5.x jars with a newer version

Just update the current `wildfly/modules/system/layers/base/org/hibernate/main` folder to contain the newer version (after stopping your WildFly server instance).

1. Delete *.index files in `wildfly/modules/system/layers/base/org/hibernate/main` and `wildfly/modules/system/layers/base/org/hibernate/envers/main` folders.
2. Backup the current contents of `wildfly/modules/system/layers/base/org/hibernate` in case you make a mistake.
3. Remove the older jars and copy new Hibernate jars into `wildfly/modules/system/layers/base/org/hibernate/main` + `wildfly/modules/system/layers/base/org/hibernate/envers/main`.
4. Update the `wildfly/modules/system/layers/base/org/hibernate/main/module.xml` + `wildfly/modules/system/layers/base/org/hibernate/envers/main/module.xml` to name the jars that you copied in.
5. Also update the hibernate-infinispan jars in `wildfly/modules/system/layers/base/org/hibernate/infinispan`.

9.16 Using Hibernate Search

WildFly 10 includes Hibernate Search. If you want to use the bundled version of Hibernate Search - which requires to use the default Hibernate ORM 5 persistence provider - this will be automatically enabled. Having this enabled means that, provided your application includes any entity which is annotated with **org.hibernate.search.annotations.Indexed**, the module **org.hibernate.search.orm:main** will be made available to your deployment; this will also include the required version of Apache Lucene.

If you do not want this module to be exposed to your deployment, set the persistence property **wildfly.jpa.hibernate.search.module** to either **none** to not automatically inject any Hibernate Search module, or to any other module identifier to inject a different module.

For example you could set **wildfly.jpa.hibernate.search.module=org.hibernate.search.orm:5.4.0.Alpha1** to use the experimental version 5.4.0.Alpha1 instead of the provided module; in this case you'll have to download and add the custom modules to the application server as other versions are not included.

When setting **wildfly.jpa.hibernate.search.module=none** you might also opt to include Hibernate Search and its dependencies within your application but we highly recommend the modules approach.



9.17 Packaging the Hibernate JPA persistence provider with your application

WildFly 8 allows the packaging of Hibernate 4.x (or greater) persistence provider jars with the application. The JPA deployer will detect the presence of a persistence provider in the application and

jboss.as.jpa.providerModule needs to be set to **application**.<?xml version="1.0" encoding="UTF-8"?>

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence" version="1.0">
```

```
<persistence-unit name="myOwnORMVersion_pu">
```

```
<description>Hibernate 4 Persistence Unit.</description>
```

```
<jta-data-source>java:jboss/datasources/PlannerDS</jta-data-source>
```

```
<properties>
```

```
  <property name="jboss.as.jpa.providerModule" value="application" />
```

```
</properties>
```

```
</persistence-unit>
```

```
</persistence>
```





9.18 Using OpenJPA

You need to copy the OpenJPA jars (e.g. openjpa-all.jar serp.jar) into the WildFly modules/system/layers/base/org/apache/openjpa/main folder and update modules/system/layers/base/org/apache/openjpa/main/module.xml to include the same jar file names that you copied in.

```
<module xmlns="urn:jboss:module:1.1" name="org.apache.openjpa">
  <resources>
    <resource-root path="jipijapa-openjpa-1.0.1.Final.jar" />
    <resource-root path="openjpa-all.jar">
      <filter>
        <exclude path="javax/**" />
      </filter>
    </resource-root>
    <resource-root path="serp.jar" />
  </resources>

  <dependencies>
    <module name="javax.api" />
    <module name="javax.annotation.api" />
    <module name="javax.enterprise.api" />
    <module name="javax.persistence.api" />
    <module name="javax.transaction.api" />
    <module name="javax.validation.api" />
    <module name="javax.xml.bind.api" />
    <module name="org.apache.commons.collections" />
    <module name="org.apache.commons.lang" />
    <module name="org.jboss.as.jpa.spi" />
    <module name="org.jboss.logging" />
    <module name="org.jboss.vfs" />
    <module name="org.jboss.jandex" />
  </dependencies>
</module>
```

9.19 Using EclipseLink

You need to copy the EclipseLink jar (e.g. eclipselink-2.6.0.jar or eclipselink.jar as in the example below) into the WildFly modules/system/layers/base/org/eclipse/persistence/main folder and update modules/system/layers/base/org/eclipse/persistence/main/module.xml to include the EclipseLink jar (take care to use the jar name that you copied in). If you happen to leave the EclipseLink version number in the jar name, the module.xml should reflect that.



```
<module xmlns="urn:jboss:module:1.1" name="org.eclipse.persistence">
  <resources>
    <resource-root path="jipijapa-eclipselink-10.0.0.Final.jar" />
    <resource-root path="eclipselink.jar">
      <filter>
        <exclude path="javax/**" />
      </filter>
    </resource-root>
  </resources>

  <dependencies>
    <module name="asm.asm" />
    <module name="javax.api" />
    <module name="javax.annotation.api" />
    <module name="javax.enterprise.api" />
    <module name="javax.persistence.api" />
    <module name="javax.transaction.api" />
    <module name="javax.validation.api" />
    <module name="javax.xml.bind.api" />
    <module name="javax.ws.rs.api" />
    <module name="org.antlr" />
    <module name="org.apache.commons.collections" />
    <module name="org.dom4j" />
    <module name="org.jboss.as.jpa.spi" />
    <module name="org.jboss.logging" />
    <module name="org.jboss.vfs" />
  </dependencies>
</module>
```

As a workaround for [issue id=414974](#), set (WildFly) system property "eclipselink.archive.factory" to value "org.jipijapa.eclipselink.JBossArchiveFactoryImpl" via `jboss-cli.sh` command (WildFly server needs to be running when this command is issued):

```
jboss-cli.sh --connect
'/system-property=eclipselink.archive.factory:add(value=org.jipijapa.eclipselink.JBossArchiveFacto
```

. The following shows what the `standalone.xml` (or your WildFly configuration you are using) file might look like after updating the system properties:

```
<system-properties>
  ...
  <property name="eclipselink.archive.factory"
value="org.jipijapa.eclipselink.JBossArchiveFactoryImpl" />
</system-properties>
```

You should then be able to deploy applications with `persistence.xml` that include;

```
<provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
```



Also refer to page [how to use EclipseLink with WildFly guide here](#).

9.20 Using DataNucleus

Read the [how to use DataNucleus with WildFly guide here](#).

9.21 Native Hibernate use

Applications that use the Hibernate API directly, are referred to here as native Hibernate applications. Native Hibernate applications, can choose to use the Hibernate jars included with WildFly or they can package their own copy of the Hibernate jars. Applications that utilize JPA will automatically have the Hibernate classes injected onto the application deployment classpath. Meaning that JPA applications, should expect to use the Hibernate jars included in WildFly.

Example MANIFEST.MF entry to add dependency for Hibernate native applications:

```
Manifest-Version: 1.0
...
Dependencies: org.hibernate
```

If you use the Hibernate native api in your application and also use the JPA api to access the same entities (from the same Hibernate session/EntityManager), you could get surprising results (e.g. `HibernateSession.saveOrUpdate(entity)` is different than `EntityManager.merge(entity)`). Each entity should be managed by either Hibernate native API or JPA code.

9.22 Injection of Hibernate Session and SessionFactoryInjection of Hibernate Session and SessionFactory

You can inject a `org.hibernate.Session` and `org.hibernate.SessionFactory` directly, just as you can do with `EntityManagers` and `EntityManagerFactories`.

```
import org.hibernate.Session;
import org.hibernate.SessionFactory;
@Stateful public class MyStatefulBean ... {
    @PersistenceContext(unitName="crm") Session session1;
    @PersistenceContext(unitName="crm2", type=EXTENDED) Session extendedpc;
    @PersistenceUnit(unitName="crm") SessionFactory factory;
}
```



9.23 Hibernate properties

WildFly automatically sets the following Hibernate (5.x) properties (if not already set in persistence unit definition):

Property	Purpose
hibernate.id.new_generator_mappings =true	New applications should let the default to true, older applications with existing data might need set to false (see note below). It really depends on whether your application uses the <code>@GeneratedValue(AUTO)</code> which will generate new key values for newly created entities. The application can override this value (in the persistence.xml)
hibernate.transaction.jta.platform = instance of <code>org.hibernate.service.jta.platform.spi.JtaPlatform</code> interface	The transaction manager, used for transaction and transaction synchronization registry is passed into Hibernate via this class.
hibernate.ejb.resource_scanner = instance of <code>org.hibernate.ejb.packaging.Scanner</code> interface	Instance of entity scanning class is passed in that knows how to use the AS annotation indexes (for faster deployment).
hibernate.transaction.manager_lookup_class	This property is removed if found in the persistence.xml (could conflict with JtaPlatform)
hibernate.session_factory_name = qualified persistence unit name	Is set to the application name or persistence unit name (application can specify a different value but it needs to be unique across all application deployments on the AS instance).
hibernate.session_factory_name_is_jndi = false	only set if the application didn't specify a value for <code>hibernate.session_factory_name</code>



hibernate.ejb.entitymanager_factory_name = qualified persistence unit name	Is set to the application name persistence unit name (application can specify a different value but it needs to be unique across all application deployments on the AS instance).
hibernate.query.jpql_strict_compliance =true	
hibernate.auto_quote_keyword =false	
hibernate.implicit_naming_strategy =org.hibernate.boot.model.naming.ImplicitNamingStrategyJpaCompliantImpl	

In Hibernate 4.x (and greater), if **new_generator_mappings** is **true**:

- @GeneratedValue(AUTO) maps to org.hibernate.id.enhanced.SequenceStyleGenerator
- @GeneratedValue(TABLE) maps to org.hibernate.id.enhanced.TableGenerator
- @GeneratedValue(SEQUENCE) maps to org.hibernate.id.enhanced.SequenceStyleGenerator

In Hibernate 4.x (and greater), if **new_generator_mappings** is **false**:

- @GeneratedValue(AUTO) maps to Hibernate "native"
- @GeneratedValue(TABLE) maps to org.hibernate.id.MultipleHiLoPerTableGenerator
- @GeneratedValue(SEQUENCE) to Hibernate "seqhilo"

9.24 Persistence unit properties

The following properties are supported in the persistence unit definition (in the persistence.xml file):

Property	Purpose
jboss.as.jpa.providerModule	name of the persistence provider module (default is org.hibernate). Should be application , if a persistence provider is packaged with the application. See note below about some module names that are built in (based on the provider).
jboss.as.jpa.adapterModule	name of the integration classes that help WildFly to work with the persistence provider.
jboss.as.jpa.adapterClass	class name of the integration adapter.
jboss.as.jpa.managed	set to false to disable container managed JPA access to the persistence unit. The default is true , which enables container managed JPA access to the persistence unit. This is typically set to false for Seam 2.x + Spring applications.



jboss.as.jpa.classtransformer	set to false to disable class transformers for the persistence unit. The default is true , which allows class enhancing/rewriting. Hibernate also needs persistence unit property hibernate.ejb.use_class_enhancer to be true, for class enhancing to be enabled.
wildfly.jpa.default-unit	set to true to choose the default persistence unit in an application. This is useful if you inject a persistence context without specifying the unitName (<code>@PersistenceContext EntityManager em</code>) but have multiple persistence units specified in your persistence.xml.
wildfly.jpa.twophasebootstrap	persistence providers (like Hibernate ORM 4.3.x via <code>EntityManagerFactoryBuilder</code>), allow a two phase persistence unit bootstrap, which improves JPA integration with CDI. Setting the wildfly.jpa.twophasebootstrap hint to false, disables the two phase bootstrap (for the persistence unit that contains the hint).
wildfly.jpa.allowdefaultdatasourceuse	set to false to prevent persistence unit from using the default data source. Defaults to true. This is only important for persistence units that do not specify a datasource.
jboss.as.jpa.deferdetach	Controls whether transaction scoped persistence context used in non-JTA transaction thread, will detach loaded entities after each <code>EntityManager</code> invocation or when the persistence context is closed (e.g. business method ends). Defaults to false (entities are cleared after <code>EntityManager</code> invocation) and if set to true, the detach is deferred until the context is closed.
wildfly.jpa.hibernate.search.module	Controls which version of Hibernate Search to include on classpath. Only makes sense when using Hibernate as JPA implementation. The default is auto ; other valid values are none or a full module identifier to use an alternative version.
jboss.as.jpa.scopedname	Specify the qualified (application scoped) persistence unit name to be used. By default, this is internally set to the application name + persistence unit name. The <code>hibernate.cache.region_prefix</code> will default to whatever you set <code>jboss.as.jpa.scopedname</code> to. Make sure you set the <code>jboss.as.jpa.scopedname</code> value to a value not already in use by other applications deployed on the same application server instance.



9.25 Determine the persistence provider module

As mentioned above, if the `jboss.as.jpa.providerModule` property is not specified, the provider module name is determined by the `provider` name specified in the `persistence.xml`. The mapping is:

Provider Name	Module name
blank	<code>org.hibernate</code>
<code>org.hibernate.ejb.HibernatePersistence</code>	<code>org.hibernate</code>
<code>org.hibernate.ogm.jpa.HibernateOgmPersistence</code>	<code>org.hibernate.ogm</code>
<code>oracle.toplink.essentials.PersistenceProvider</code>	<code>oracle.toplink</code>
<code>oracle.toplink.essentials.ejb.cmp3.EntityManagerFactoryProvider</code>	<code>oracle.toplink</code>
<code>org.eclipse.persistence.jpa.PersistenceProvider</code>	<code>org.eclipse.persistence</code>
<code>org.datanucleus.api.jpa.PersistenceProviderImpl</code>	<code>org.datanucleus</code>
<code>org.datanucleus.store.appengine.jpa.DatastorePersistenceProvider</code>	<code>org.datanucleus:appengine</code>
<code>org.apache.openjpa.persistence.PersistenceProviderImpl</code>	<code>org.apache.openjpa</code>



9.26 Binding EntityManagerFactory/EntityManager to JNDI

By default WildFly does **not** bind the entity manager factory to JNDI. However, you can explicitly configure this in the `persistence.xml` of your application by setting the `jboss.entity.manager.factory.jndi.name` hint. The value of that property should be the JNDI name to which the entity manager factory should be bound.

You can also bind a container managed (transaction scoped) entity manager to JNDI as well, } via hint `jboss.entity.manager.jndi.name`{. As a reminder, a transaction scoped entity manager (persistence context), acts as a proxy that always gets an unique underlying entity manager (at the persistence provider level).

Here's an example:

`persistence.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
  xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="
    http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
  <persistence-unit name="myPU">
    <!-- If you are running in a production environment, add a managed
      data source, the example data source is just for proofs of concept! -->
    <jta-data-source>java:jboss/datasources/ExampleDS</jta-data-source>
    <properties>
      <!-- Bind entity manager factory to JNDI at java:jboss/myEntityManagerFactory -->
      <property name="jboss.entity.manager.factory.jndi.name"
value="java:jboss/myEntityManagerFactory" />
      <property name="jboss.entity.manager.jndi.name" value="java:/myEntityManager"/>
    </properties>
  </persistence-unit>
</persistence>
```

```
@Stateful
public class ExampleSFSB {
  public void createSomeEntityWithTransactionScopedEM(String name) {
    Context context = new InitialContext();
    javax.persistence.EntityManager entityManager = (javax.persistence.EntityManager)
context.lookup("java:/myEntityManager");
    SomeEntity someEntity = new SomeEntity();
    someEntity.setName(name);    entityManager.persist(name);
  }
}
```




9.27 Community

Many thanks to the community, for reporting issues, solutions and code changes. A number of people have been answering Wildfly forum questions related to JPA usage. I would like to thank them for this, as well as those reporting issues. For those of you that haven't downloaded the AS source code and started hacking patches together. I would like to encourage you to start by reading [Hacking on WildFly](#). You will find that it is very easy to find your way around the WildFly/JPA/* source tree and make changes. Also, new for WildFly, is the JipiJapa project that contains additional integration code that makes EE JPA application deployments work better. The following list of contributors should grow over time, I hope to see more of you listed here.

9.27.1 People who have contributed to the WildFly JPA layer:

- [Carlo de Wolf](#) (lead of the EJB3 project)
- [Steve Ebersole](#) (lead of the Hibernate ORM project)
- [Stuart Douglas](#) (lead of the Seam Persistence project, WildFly project team member/commmitter)
- [Jaikiran Pai](#) (Active member of JBoss forums and JBoss EJB3 project team member)
- [Strong Liu](#) (leads the productization effort of Hibernate in the EAP product)
- [Scott Marlow](#) (lead of the WildFly container JPA sub-project)
- [Antti Laisi](#) (**OpenJPA integration changes**)
- [Galder Zamarreño](#) (Infinispan 2lc documentation)
- [Sanne Grinovero](#) (lead of the Hibernate Search project)
- [Paul Ferraro](#) (Infinispan 2lc integration)



10 OSGi developer guide

Couldn't find a page to include called: OSGi Developer Guide



11 JNDI reference guide

11.1 Overview

WildFly offers several mechanisms to retrieve components by name. Every WildFly instance has its own local JNDI namespace (`java:`) which is unique per JVM. The layout of this namespace is primarily governed by the Java EE specification. Applications which share the same WildFly instance can use this namespace to intercommunicate. In addition to local JNDI, a variety of mechanisms exist to access remote components.

- Client JNDI - This is a mechanism by which remote components can be accessed using the JNDI APIs, but **without network round-trips**. This approach is the most efficient, and **removes a potential single point of failure**. For this reason, it is highly recommended to use Client JNDI over traditional remote JNDI access. However, to make this possible, it does require that all names follow a strict layout, so user customizations are not possible. Currently only access to remote EJBs is supported via the `ejb:` namespace. Future revisions will likely add a JMS client JNDI namespace.
- Traditional Remote JNDI - This is a more familiar approach to EE application developers, where the client performs a remote component name lookup against a server, and a proxy/stub to the component is serialized as part of the name lookup and returned to the client. The client then invokes a method on the proxy which results in another remote network call to the underlying service. In a nutshell, traditional remote JNDI involves two calls to invoke an EE component, whereas Client JNDI requires one. It does however allow for customized names, and for a centralised directory for multiple application servers. This centralized directory is, however, *a single point of failure*.
- EE Application Client / Server-To-Server Delegation - This approach is where local names are bound as an *alias* to a remote name using one of the above mechanisms. This is useful in that it allows applications to only ever reference standard portable Java EE names in both code and deployment descriptors. It also allows for the application to be unaware of network topology details/ This can even work with Java SE clients by using the little known EE Application Client feature. This feature allows you to run an extremely minimal AS server around your application, so that you can take advantage of certain core services such as naming and injection.

11.2 Local JNDI


The Java EE platform specification defines the following JNDI contexts:


- `java:comp` - The namespace is scoped to the current component (i.e. EJB)
- `java:module` - Scoped to the current module
- `java:app` - Scoped to the current application
- `java:global` - Scoped to the application server

In addition to the standard namespaces, WildFly also provides the following two global namespaces:



- java:jboss
- java:/

 Only entries within the `java:jboss/exported` context are accessible over remote JNDI.

 For web deployments `java:comp` is aliased to `java:module`, so EJB's deployed in a war do not have their own comp namespace.

11.2.1 Binding entries to JNDI

There are several methods that can be used to bind entries into JNDI in WildFly.

Using a deployment descriptor

For Java EE applications the recommended way is to use a [deployment descriptor](#) to create the binding. For example the following `web.xml` binds the string "Hello World" to `java:global/mystring` and the string "Hello Module" to `java:comp/env/hello` (any non absolute JNDI name is relative to `java:comp/env` context).

```
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
  version="3.1">
  <env-entry>
    <env-entry-name>java:global/mystring</env-entry-name>
    <env-entry-type>java.lang.String</env-entry-type>
    <env-entry-value>Hello World</env-entry-value>
  </env-entry>
  <env-entry>
    <env-entry-name>hello</env-entry-name>
    <env-entry-type>java.lang.String</env-entry-type>
    <env-entry-value>Hello Module</env-entry-value>
  </env-entry>
</web-app>
```

For more details, see the [Java EE Platform Specification](#).




Programmatically

Java EE Applications

Standard Java EE applications may use the standard JNDI API, included with Java SE, to bind entries in the global namespaces (the standard `java:comp`, `java:module` and `java:app` namespaces are read-only, as mandated by the Java EE Platform Specification).


```
InitialContext initialContext = new InitialContext();
initialContext.bind("java:global/a", 100);
```

 There is no need to unbind entries created programmatically, since WildFly tracks which bindings belong to a deployment, and the bindings are automatically removed when the deployment is undeployed.

WildFly Modules and Extensions

With respect to code in WildFly Modules/Extensions, which is executed out of a Java EE application context, using the standard JNDI API may result in a `UnsupportedOperationException` if the target namespace uses a `WritableServiceBasedNamingStore`. To work around that, the `bind()` invocation needs to be wrapped using WildFly proprietary APIs:

```
InitialContext initialContext = new InitialContext();
WritableServiceBasedNamingStore.pushOwner(serviceTarget);
try {
    initialContext.bind("java:global/a", 100);
} finally {
    WritableServiceBasedNamingStore.popOwner();
}
```

 The `ServiceTarget` removes the bind when uninstalled, thus using one out of the module/extension domain usage should be avoided, unless entries are removed using `unbind()`.



Naming Subsystem Configuration

It is also possible to bind to one of the three global namespaces using configuration in the naming subsystem. This can be done by either editing the `standalone.xml/domain.xml` file directly, or through the management API.

Four different types of bindings are supported:

- Simple - A primitive or `java.net.URL` entry (default is `java.lang.String`).
- Object Factory - This allows to specify the `javax.naming.spi.ObjectFactory` that is used to create the looked up value.
- External Context - An external context to federate, such as an LDAP Directory Service
- Lookup - The allows to create JNDI aliases, when this entry is looked up it will lookup the target and return the result.

An example `standalone.xml` might look like:

```
<subsystem xmlns="urn:jboss:domain:naming:2.0" >
  <bindings>
    <simple name="java:global/a" value="100" type="int" />
    <simple name="java:global/jbossDocs" value="https://docs.jboss.org" type="java.net.URL" />
    <object-factory name="java:global/b" module="com.acme" class="org.acme.MyObjectFactory" />
    <external-context name="java:global/federation/ldap/example"
class="javax.naming.directory.InitialDirContext" cache="true">
      <environment>
        <property name="java.naming.factory.initial" value="com.sun.jndi.ldap.LdapCtxFactory" />
        <property name="java.naming.provider.url" value="ldap://ldap.example.com:389" />
        <property name="java.naming.security.authentication" value="simple" />
        <property name="java.naming.security.principal" value="uid=admin,ou=system" />
        <property name="java.naming.security.credentials" value="secret" />
      </environment>
    </external-context>
    <lookup name="java:global/c" lookup="java:global/b" />
  </bindings>
</subsystem>
```

The CLI may also be used to bind an entry. As an example:

```
/subsystem=naming/binding=java\:global\mybinding:add(binding-type=simple, type=long,
value=1000)
```



WildFly's Administrator Guide includes a section describing in detail the Naming subsystem configuration.



11.2.2 Retrieving entries from JNDI

Resource Injection

For Java EE applications the recommended way to lookup a JNDI entry is to use `@Resource` injection:

```
@Resource(lookup = "java:global/mystring")
private String myString;

@Resource(name = "hello")
private String hello;

@Resource
ManagedExecutorService executor;
```

Note that `@Resource` is more than a JNDI lookup, it also binds an entry in the component's JNDI environment. The new bind JNDI name is defined by `@Resource`'s `name` attribute, which value, if unspecified, is the Java type concatenated with `/` and the field's name, for instance `java.lang.String/myString`. More, similar to when using deployment descriptors to bind JNDI entries, unless the name is an absolute JNDI name, it is considered relative to `java:comp/env`. For instance, with respect to the field named `myString` above, the `@Resource`'s `lookup` attribute instructs WildFly to lookup the value in `java:global/mystring`, bind it in `java:comp/env/java.lang.String/myString`, and then inject such value into the field.

With respect to the field named `hello`, there is no `lookup` attribute value defined, so the responsibility to provide the entry's value is delegated to the deployment descriptor. Considering that the deployment descriptor was the `web.xml` previously shown, which defines an environment entry with same `hello` name, then WildFly inject the valued defined in the deployment descriptor into the field.

The `executor` field has no attributes specified, so the bind's name would default to `java:comp/env/javax.enterprise.concurrent.ManagedExecutorService/executor`, but there is no such entry in the deployment descriptor, and when that happens it's up to WildFly to provide a default value or null, depending on the field's Java type. In this particular case WildFly would inject the default instance of a managed executor service, the value in `java:comp/DefaultManagedExecutorService`, as mandated by the EE Concurrency Utilities 1.0 Specification (JSR 236).



Standard Java SE JNDI API

Java EE applications may use, without any additional configuration needed, the standard JNDI API to lookup an entry from JNDI:

```
String myString = (String) new InitialContext().lookup("java:global/mystring");
```

or simply

```
String myString = InitialContext.doLookup("java:global/mystring");
```

11.3 Remote JNDI

WildFly supports two different types of remote JNDI. The old jnp based JNDI implementation used in JBoss AS versions prior to 7.x is no longer supported.

11.3.1 remote:

The `remote:` protocol uses the WildFly remoting protocol to lookup items from the servers local JNDI. To use it, you must have the appropriate jars on the class path, if you are maven user can be done simply by adding the following to your `pom.xml`:

```
<dependency>
  <groupId>org.wildfly</groupId>
  <artifactId>wildfly-ejb-client-bom</artifactId>
  <version>8.0.0.Final</version>
  <type>pom</type>
  <scope>compile</scope>
</dependency>
```

If you are not using maven a shaded jar that contains all required classes can be found in the `bin/client` directory of WildFly's distribution.

```
final Properties env = new Properties();
env.put(Context.INITIAL_CONTEXT_FACTORY,
org.jboss.naming.remote.client.InitialContextFactory.class.getName());
env.put(Context.PROVIDER_URL, "remote://localhost:4447");
remoteContext = new InitialContext(env);
```




11.3.2 ejb:

The `ejb:` namespace is provided by the `jboss-ejb-client` library. This protocol allows you to look up EJB's, using their application name, module name, `ejb` name and interface type.

This is a client side JNDI implementation. Instead of looking up an EJB on the server the lookup name contains enough information for the client side library to generate a proxy with the EJB information. When you invoke a method on this proxy it will use the current EJB client context to perform the invocation. If the current context does not have a connection to a server with the specified EJB deployed then an error will occur. Using this protocol it is possible to look up EJB's that do not actually exist, and no error will be thrown until the proxy is actually used. The exception to this is stateful session beans, which need to connect to a server when they are created in order to create the session bean instance on the server.

Some examples are:

```
ejb:myapp/myejbjar/MyEjbName!com.test.MyRemoteInterface
```

```
ejb:myapp/myejbjar/MyStatefulName!comp.test.MyStatefulRemoteInterface?stateful
```

The first example is a lookup of a singleton, stateless or EJB 2.x home interface. This lookup will not hit the server, instead a proxy will be generated for the remote interface specified in the name. The second example is for a stateful session bean, in this case the JNDI lookup will hit the server, in order to tell the server to create the SFSB session.

For more details on how the server connections are configured, please see [EJB invocations from a remote client using JNDI](#).



12 Spring applications development and migration guide

This document details the main points that need to be considered by Spring developers that wish to develop new applications or to migrate existing applications to be run into WildFly 8.

12.1 Dependencies and Modularity

WildFly 8 has a modular class loading strategy, different from previous versions of JBoss AS, which enforces a better class loading isolation between deployments and the application server itself. A detailed description can be found in the documentation dedicated to [class loading in WildFly 8](#).

This reduces significantly the risk of running into a class loading conflict and allows applications to package their own dependencies if they choose to do so. This makes it easier for Spring applications that package their own dependencies - such as logging frameworks or persistence providers to run on WildFly 8.

At the same time, this does not mean that duplications and conflicts cannot exist on the classpath. Some module dependencies are implicit, depending on the type of deployment as shown [here](#).

12.2 Persistence usage guide

Depending on the strategy being used, Spring applications can be:

- native Hibernate applications;
- JPA-based applications;
- native JDBC applications;

12.3 Native Spring/Hibernate applications

Applications that use the Hibernate API directly with Spring (i.e. through either one of `LocalSessionFactoryBean` or `AnnotationSessionFactoryBean`) may use a version of Hibernate 3 packaged inside the application. Hibernate 4 (which is provided through the 'org.hibernate' module of WildFly 8) is not supported by Spring 3.0 and Spring 3.1 (and may be supported by Spring 3.2 as described in [SPR-8096](#)), so adding this module as a dependency is not a solution.

12.4 based applications

Spring applications using JPA may choose between:

- using a server-deployed persistence unit;
- using a Spring-managed persistence unit.



12.4.1 Using server-deployed persistence units

Applications that use a server-deployed persistence unit must observe the typical Java EE rules in what concerns dependency management, i.e. the `javax.persistence` classes and persistence provider (Hibernate) are contained in modules which are added automatically by the application when the persistence unit is deployed.

In order to use the server-deployed persistence units from within Spring, either the persistence context or the persistence unit need to be registered in JNDI via `web.xml` as follows:

```
<persistence-context-ref>
  <persistence-context-ref-name>persistence/petclinic-em</persistence-unit-ref-name>
  <persistence-unit-name>petclinic</persistence-unit-name>
</persistence-context-ref>
```

or, respectively:

```
<persistence-unit-ref>
  <persistence-unit-ref-name>persistence/petclinic-emf</persistence-unit-ref-name>
  <persistence-unit-name>petclinic</persistence-unit-name>
</persistence-unit-ref>
```

When doing so, the persistence context or persistence unit are available to be looked up in JNDI, as follows:

```
<jee:jndi-lookup id="entityManager" jndi-name="java:comp/env/persistence/petclinic-em"
  expected-type="javax.persistence.EntityManager" />
```

or

```
<jee:jndi-lookup id="entityManagerFactory" jndi-name="java:comp/env/persistence/petclinic-emf"
  expected-type="javax.persistence.EntityManagerFactory" />
```

JNDI binding

JNDI binding via `persistence.xml` properties is not supported in WildFly 8.



12.4.2 Using Spring-managed persistence units

Spring applications running in WildFly 8 may also create persistence units on their own, using the `LocalContainerEntityManagerFactoryBean`. This is what these applications need to consider:

Placement of the persistence unit definitions

When the application server encounters a deployment that has a file named `META-INF/persistence.xml` (or, for that matter, `WEB-INF/classes/META-INF/persistence.xml`), it will attempt to create a persistence unit based on what is provided in the file. In most cases, such definition files are not compliant with the Java EE requirements, mostly because required elements such as the `datasource` of the persistence unit are supposed to be provided by the Spring context definitions, which will fail the deployment of the persistence unit, and consequently of the entire deployment.

Spring applications can easily avoid this type of conflict, by using a feature of the `LocalContainerEntityManagerFactoryBean` which is designed for this purpose. Persistence unit definition files can exist in other locations than `META-INF/persistence.xml` and the location can be indicated through the `persistenceXmlLocation` property of the factory bean class.

Assuming that the persistence unit is in the `META-INF/jpa-persistence.xml`, the corresponding definition can be:

```
<bean id="entityManagerFactory"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
    <property name="persistenceXmlLocation"
value="classpath*:META-INF/jpa-persistence.xml"/>
    <!-- other definitions -->
</bean>
```



12.4.3 Managing dependencies

Since the `LocalContainerEntityManagerFactoryBean` and the corresponding `HibernateJpaVendorAdapter` are based on Hibernate 3, it is required to use that version with the application. Therefore, the Hibernate 3 jars must be included in the deployment. At the same time, due the presence of `@PersistenceUnit` or `@PersistenceContext` annotations on the application classes, the application server will automatically add the 'org.hibernate' module as a dependency.

This can be avoided by instructing the server to exclude the module from the deployment's list of dependencies. In order to do so, include a `META-INF/jboss-deployment-structure.xml` or, for web applications, `WEB-INF/jboss-deployment-structure.xml` with the following content:

```
<jboss-deployment-structure xmlns="urn:jboss:deployment-structure:1.0">
  <deployment>
    <exclusions>
      <module name="org.hibernate" />
    </exclusions>
  </deployment>
</jboss-deployment-structure>
```




13 All WildFly documentation


Couldn't find a page to include called: All JBoss AS 7 documentation



14 Application Client Reference

As a Java EE6 compliant server, WildFly 8 contains an application client. An application client is essentially a cut down server instance, that allow you to use EE features such as injection in a client side program.

 This article is not a tutorial on application client development, rather it covers the specifics of the WildFly application client. There are tutorials available elsewhere that cover application client basics, such as [this one](#).

 Note that the application client is different to the EJB client libraries, it is perfectly possible to write client application that do not use the application client, but instead use the `jboss-ejb-client` library directly.

14.1 Getting Started

To launch the application client use the `appclient.sh` or `appclient.bat` script in the bin directory. For example:

```
./appclient.sh --host=10.0.0.1 myear.ear#appClient.jar arg1
```

The `--host` argument tells the `appclient` the server to connect to. The next argument is the application client deployment to use, application clients can only run a single deployment, and this deployment must also be deployed on the full server instance that the client is connecting too.

Any arguments after the deployment to use are passed directly through to the application clients `main` function.

14.2 Connecting to more than one host

If you want to connect to more than one host or make use of the clustering functionality then you need to specify a `jboss-ejb-client.properties` file rather than a host:

```
./appclient.sh --ejb-client-properties=my-jboss-ejb-client.properties myear.ear#appClient.jar arg1
```



14.3 Example

A simple example how to package an application client and use it with WildFly can be within the quickstart [appclient](#) which is located on Github .



15 CDI Reference

WildFly uses [Weld](#), the CDI reference implementation as its CDI provider. To activate CDI for a deployment simply add a `beans.xml` file in any archive in the deployment.

This document is not intended to be a CDI tutorial, it only covers CDI usage that is specific to WildFly. For some general information on CDI see the below links:

[CDI Specification](#)

[Weld Reference Guide](#)

[The AS7 Quickstarts](#)

15.1 Using CDI Beans from outside the deployment

For WildFly 8 onwards, it is now possible have classes outside the deployment be picked up as CDI beans. In order for this to work you must add a dependency on the external deployment that your beans are coming from, and make sure the META-INF directory of this deployment is imported, so that your deployment has visibility to the `beans.xml` file (To import beans from outside the deployment they must be in an archive with a `beans.xml` file).

There are two ways to do this, either using the `MANIFEST.MF` or using `jboss-deployment-structure.xml`.

Using `MANIFEST.MF` you need to add a `Dependencies` entry, with `meta-inf` specified after the entry, e.g.

```
Dependencies: com.my-cdi-module meta-inf, com.my-other-cdi-module meta-inf
```

Using `jboss-deployment-structure.xml` you need to add a dependency entry with `meta-inf="import"`, e.g.

```
<jboss-deployment-structure xmlns="urn:jboss:deployment-structure:1.2">
  <deployment>
    <dependencies>
      <module name="deployment.dl.jar" meta-inf="import"/>
    </dependencies>
  </deployment>
</jboss-deployment-structure>
```

Note that this can be used to create beans from both modules in the `modules` directory, and from other deployments.

For more information on class loading and adding dependencies to your deployment please see the [Class Loading Guide](#)



15.2 Suppressing implicit bean archives

CDI 1.1 brings new options to packaging of CDI-enabled applications. In addition to well-known explicit bean archives (basically any archive containing the **beans.xml** file) the specification introduces **implicit bean archives**.

An implicit bean archive is any archive that contains one or more classes annotated with a bean defining annotation (scope annotation) or one or more session beans. As a result, the beans.xml file is no longer required for CDI to work in your application.

In an implicit bean archive **only those classes** that are either annotated with bean defining annotations or are session beans are recognized by CDI as beans (other classes cannot be injected).

This has a side-effect, though. Libraries exist that make use of scope annotation (bean defining annotations) for their own convenience but are not designed to run with CDI support. Guava would be an example of such library. If your application bundles such library it will be recognized as a CDI archive and may [fail the deployment](#).

Fortunately, WildFly makes it possible to suppress implicit bean archives and only enable CDI in archives that bundle the beans.xml file. There are two ways to achieve this:

15.2.1 deployment configuration

You can either set this up for your deployment only by adding the following content to the **META-INF/jboss-all.xml** file of your application:

```
<jboss xmlns="urn:jboss:1.0">
  <weld xmlns="urn:jboss:weld:1.0" require-bean-descriptor="true"/>
</jboss>
```

15.2.2

15.2.3 Global configuration

Alternatively, you may configure this for all deployments in your WildFly instance by executing the following command:

```
/subsystem=weld:write-attribute(name=require-bean-descriptor,value=true)
```



15.3 Development mode

WildFly 10 introduces a special mode for application development which allows you to inspect and monitor your CDI deployments. This mode is turned off by default and note that some features of the **development mode may have negative impact on the performance and/or functionality of the application.**

15.3.1 deployment configuration

You can enable it locally in your application `web.xml` by setting the Servlet initialization parameter `org.jboss.weld.development` to `true`:

```
<context-param>
  <param-name>org.jboss.weld.development</param-name>
  <param-value>true</param-value>
</context-param>
```

15.3.2

15.3.3 Global configuration

Alternatively, you can enable it globally in Weld subsystem by setting `development-mode` attribute to `true`:

```
/subsystem=weld:write-attribute(name=development-mode,value=true)
```

For more details and example you can check [Weld development mode](#).

Once the development mode is enabled you can check your applications CDI information using Weld Probe - [Weld Probe](#).



15.4 portable mode

CDI 1.1 clarifies some aspects of how CDI portable extensions work. As a result, some extensions that do not use the API properly (but were tolerated in CDI 1.0 environment) may stop working with CDI 1.1. If this is the case of your application you will see an exception like this:

```
org.jboss.weld.exceptions.IllegalStateException: WELD-001332: BeanManager method getBeans() is not available during application initialization
```

Fortunately, there is a non-portable mode available in WildFly which skips some of the API usage checks and therefore allows the legacy extensions to work as before.

Again, there are two ways to enable the non-portable mode:

15.4.1 deployment configuration

You can either set this up for your deployment only by adding the following content to the **META-INF/jboss-all.xml** file of your application:

```
<jboss xmlns="urn:jboss:1.0">
  <weld xmlns="urn:jboss:weld:1.0" non-portable-mode="true" />
</jboss>
```

15.4.2 Global configuration

Alternatively, you may configure this for all deployments in your WildFly instance by executing the following command:

```
/subsystem=weld:write-attribute(name=non-portable-mode,value=true)
```

Note that new portable extensions should always use the [BeanManager API](#) properly and thus never required the non-portable mode. The non-portable mode only exists to preserve compatibility with legacy extensions!



16 Class Loading in WildFly

Since JBoss AS 7, Class loading is considerably different to previous versions of JBoss AS. Class loading is based on the [JBoss Modules](#) project. Instead of the more familiar hierarchical class loading environment, WildFly's class loading is based on modules that have to define explicit dependencies on other modules. Deployments in WildFly are also modules, and do not have access to classes that are defined in jars in the application server unless an explicit dependency on those classes is defined.

16.1 Deployment Module Names

Module names for top level deployments follow the format `deployment.myarchive.war` while sub deployments are named like `deployment.myear.ear.mywar.war`.

This means that it is possible for a deployment to import classes from another deployment using the other deployments module name, the details of how to add an explicit module dependency are explained below.

16.2 Automatic Dependencies

Even though in WildFly modules are isolated by default, as part of the deployment process some dependencies on modules defined by the application server are set up for you automatically. For instance, if you are deploying a Java EE application a dependency on the Java EE API's will be added to your module automatically. Similarly if your module contains a `beans.xml` file a dependency on [Weld](#) will be added automatically, along with any supporting modules that weld needs to operate.

For a complete list of the automatic dependencies that are added, please see [Implicit module dependencies for deployments](#).

Automatic dependencies can be excluded through the use of `jboss-deployment-structure.xml`.



16.3 Class Loading Precedence

A common source of errors in Java applications is including API classes in a deployment that are also provided by the container. This can result in multiple versions of the class being created and the deployment failing to deploy properly. To prevent this in WildFly, module dependencies are added in a specific order that should prevent this situation from occurring.

In order of highest priority to lowest priority

1. System Dependencies - These are dependencies that are added to the module automatically by the container, including the Java EE api's.
2. User Dependencies - These are dependencies that are added through `jboss-deployment-structure.xml` or through the `Dependencies: manifest` entry.
3. Local Resource - Class files packaged up inside the deployment itself, e.g. class files from `WEB-INF/classes` or `WEB-INF/lib` of a war.
4. Inter deployment dependencies - These are dependencies on other deployments in an ear deployment. This can include classes in an ear's lib directory, or classes defined in other ejb jars.

16.4 WAR Class Loading

The war is considered to be a single module, so classes defined in `WEB-INF/lib` are treated the same as classes in `WEB-INF/classes`. All classes packaged in the war will be loaded with the same class loader.

16.5 EAR Class Loading

Ear deployments are multi-module deployments. This means that not all classes inside an ear will necessarily have access to all other classes in the ear, unless explicit dependencies have been defined. By default the `EAR/lib` directory is a single module, and every WAR or EJB jar deployment is also a separate module. Sub deployments (wars and ejb-jars) always have a dependency on the parent module, which gives them access to classes in `EAR/lib`, however they do not always have an automatic dependency on each other. This behaviour is controlled via the `ear-subdeployments-isolated` setting in the ee subsystem configuration:

```
<subsystem xmlns="urn:jboss:domain:ee:1.0" >
  <ear-subdeployments-isolated>false</ear-subdeployments-isolated>
</subsystem>
```


By default this is set to false, which allows the sub-deployments to see classes belonging to other sub-deployments within the `.ear`.

For example, consider the following `.ear` deployment:



```
myapp.ear
|
|--- web.war
|
|--- ejb1.jar
|
|--- ejb2.jar
```


If the `ear-subdeployments-isolated` is set to `false`, then the classes in `web.war` can access classes belonging to `ejb1.jar` and `ejb2.jar`. Similarly, classes from `ejb1.jar` can access classes from `ejb2.jar` (and vice-versa).

 The `ear-subdeployments-isolated` element value has no effect on the isolated classloader of the `.war` file(s). i.e. irrespective of whether this flag is set to `true` or `false`, the `.war` within a `.ear` will have a isolated classloader and other sub-deployments within that `.ear` will not be able to access classes from that `.war`. This is as per spec.

If the `ear-subdeployments-isolated` is set to `true` then no automatic module dependencies between the sub-deployments are set up. User must manually setup the dependency with `Class-Path` entries, or by setting up explicit module dependencies.

Portability

The Java EE specification says that portable applications should not rely on sub deployments having access to other sub deployments unless an explicit `Class-Path` entry is set in the `MANIFEST.MF`. So portable applications should always use `Class-Path` entry to explicitly state their dependencies.

 It is also possible to override the `ear-subdeployments-isolated` element value at a per deployment level. See the section on `jboss-deployment-structure.xml` below.

Dependencies: **Manifest Entries**

Deployments (or more correctly modules within a deployment) may set up dependencies on other modules by adding a `Dependencies: manifest` entry. This entry consists of a comma separated list of module names that the deployment requires. The available modules can be seen under the `modules` directory in the application server distribution. For example to add a dependency on `javassist` and `apache velocity` you can add a manifest entry as follows:

```
Dependencies: org.javassist export,org.apache.velocity export services,organtlr
```


Each dependency entry may also specify some of the following parameters by adding them after the module name:



- `export` This means that the dependencies will be exported, so any module that depends on this module will also get access to the dependency.
- `services` By default items in `META-INF` of a dependency are not accessible, this makes items from `META-INF/services` accessible so `services` in the modules can be loaded.
- `optional` If this is specified the deployment will not fail if the module is not available.
- `meta-inf` This will make the contents of the `META-INF` directory available (unlike `services`, which just makes `META-INF/services` available). In general this will not cause any deployment descriptors in `META-INF` to be processed, with the exception of `beans.xml`. If a `beans.xml` file is present this module will be scanned by Weld and any resulting beans will be available to the application.
- `annotations` If a jandex index has been created for the module these annotations will be merged into the deployments annotation index. The `Jandex` index can be generated using the `Jandex ant task`, and must be named `META-INF/jandex.idx`. Note that it is not necessary to break open the jar being indexed to add this to the modules class path, a better approach is to create a jar containing just this index, and adding it as an additional resource root in the `module.xml` file.

Adding a dependency to all modules in an EAR

Using the `export` parameter it is possible to add a dependency to all sub deployments in an ear. If a module is exported from a `Dependencies:` entry in the top level of the ear (or by a jar in the `ear/lib` directory) it will be available to all sub deployments as well.

 To generate a MANIFEST.MF entry when using maven put the following in your pom.xml:

pom.xml

```
<build>
  ...
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-war-plugin</artifactId>
      <configuration>
        <archive>
          <manifestEntries>
            <Dependencies>org.slf4j</Dependencies>
          </manifestEntries>
        </archive>
      </configuration>
    </plugin>
  </plugins>
</build>
```

If your deployment is a jar you must use the `maven-jar-plugin` rather than the `maven-war-plugin`.



16.5.1 Class Path Entries

It is also possible to add module dependencies on other modules inside the deployment using the `Class-Path` manifest entry. This can be used within an ear to set up dependencies between sub deployments, and also to allow modules access to additional jars deployed in an ear that are not sub deployments and are not in the `EAR/lib` directory. If a jar in the `EAR/lib` directory references a jar via `Class-Path`: then this additional jar is merged into the parent ear's module, and is accessible to all sub deployments in the ear.

16.6 Global Modules

It is also possible to set up global modules, that are accessible to all deployments. This is done by modifying the configuration file (`standalone/domain.xml`).

For example, to add `javassist` to all deployments you can use the following XML:

`standalone.xml/domain.xml`

```
<subsystem xmlns="urn:jboss:domain:ee:1.0" >
  <global-modules>
    <module name="org.javassist" slot="main" />
  </global-modules>
</subsystem>
```

Note that the `slot` field is optional and defaults to `main`.

16.7 JBoss Deployment Structure File

`jboss-deployment-structure.xml` is a JBoss specific deployment descriptor that can be used to control class loading in a fine grained manner. It should be placed in the top level deployment, in `META-INF` (or `WEB-INF` for web deployments). It can do the following:

- Prevent automatic dependencies from being added
- Add additional dependencies
- Define additional modules
- Change an EAR deployments isolated class loading behaviour
- Add additional resource roots to a module

An example of a complete `jboss-deployment-structure.xml` file for an ear deployment is as follows:

`jboss-deployment-structure.xml`


```
<jboss-deployment-structure>
  <!-- Make sub deployments isolated by default, so they cannot see each others classes without
  a Class-Path entry -->
```



```
<ear-subdeployments-isolated>true</ear-subdeployments-isolated>
<!-- This corresponds to the top level deployment. For a war this is the war's module, for an
ear -->
<!-- This is the top level ear module, which contains all the classes in the EAR's lib folder
-->
<deployment>
  <!-- exclude-subsystem prevents a subsystems deployment unit processors running on a
deployment -->
  <!-- which gives basically the same effect as removing the subsystem, but it only affects
single deployment -->
  <exclude-subsystems>
    <subsystem name="resteasy" />
  </exclude-subsystems>
  <!-- Exclusions allow you to prevent the server from automatically adding some dependencies
-->
  <exclusions>
    <module name="org.javassist" />
  </exclusions>
  <!-- This allows you to define additional dependencies, it is the same as using the
Dependencies: manifest attribute -->
  <dependencies>
    <module name="deployment.javassist.proxy" />
    <module name="deployment.myjavassist" />
    <!-- Import META-INF/services for ServiceLoader impls as well -->
    <module name="myservicemodule" services="import"/>
  </dependencies>
  <!-- These add additional classes to the module. In this case it is the same as including
the jar in the EAR's lib directory -->
  <resources>
    <resource-root path="my-library.jar" />
  </resources>
</deployment>
<sub-deployment name="myapp.war">
  <!-- This corresponds to the module for a web deployment -->
  <!-- it can use all the same tags as the <deployment> entry above -->
  <dependencies>
    <!-- Adds a dependency on a ejb jar. This could also be done with a Class-Path entry -->
    <module name="deployment.myear.ear.myejbjar.jar" />
  </dependencies>
  <!-- Set's local resources to have the lowest priority -->
  <!-- If the same class is both in the sub deployment and in another sub deployment that -->
  <!-- is visible to the war, then the Class from the other deployment will be loaded, -->
  <!-- rather than the class actually packaged in the war. -->
  <!-- This can be used to resolve ClassCastExceptions if the same class is in multiple sub
deployments-->
  <local-last value="true" />
</sub-deployment>
<!-- Now we are going to define two additional modules -->
<!-- This one is a different version of javassist that we have packaged -->
<module name="deployment.myjavassist" >
  <resources>
    <resource-root path="javassist.jar" >
      <!-- We want to use the servers version of javassist.util.proxy.* so we filter it out-->
      <filter>
        <exclude path="javassist/util/proxy" />
      </filter>
    </resource-root>
  </resources>
```



```
</module>
<!-- This is a module that re-exports the containers version of javassist.util.proxy -->
<!-- This means that there is only one version of the Proxy classes defined -->
<module name="deployment.javassist.proxy" >
  <dependencies>
    <module name="org.javassist" >
      <imports>
        <include path="javassist/util/proxy" />
        <exclude path="/**" />
      </imports>
    </module>
  </dependencies>
</module>
</jboss-deployment-structure>
```

 The xsd for `jboss-deployment-structure.xml` is available at <https://github.com/wildfly/wildfly/blob/master/build/src/main/resources/docs/schema/jboss-deployment-structure.xsd>

16.8 Accessing JDK classes

Not all JDK classes are exposed to a deployment by default. If your deployment uses JDK classes that are not exposed you can get access to them using `jboss-deployment-structure.xml` with system dependencies:

Using `jboss-deployment-structure.xml` to access JDK classes

```
<jboss-deployment-structure xmlns="urn:jboss:deployment-structure:1.1">
  <deployment>
    <dependencies>
      <system export="true">
        <paths>
          <path name="com/sun/corba/se/spi/legacy/connection"/>
        </paths>
      </system>
    </dependencies>
  </deployment>
</jboss-deployment-structure>
```

16.9 The "jboss.api" property and application use of modules shipped with WildFly

The WildFly distribution includes a large number of modules, a great many of which are included for use by WildFly internals, with no testing of the appropriateness of their direct use by applications or any commitment to continue to ship those modules in future releases if they are no longer needed by the internals. So how can a user know whether it is advisable for their application to specify an explicit dependency on a module WildFly ships? The "jboss.api" property specified in the module's `module.xml` file can tell you:

**Example declaration of the `jboss.api` property**

```
<module xmlns="urn:jboss:module:1.3" name="com.google.guava">
  <properties>
    <property name="jboss.api" value="private"/>
  </properties>
</module>
```

If a module does not have a property element like the above, then it's equivalent to one with a value of "public".

Following are the meanings of the various values you may see for the `jboss.api` property:

Value	Meaning
public	May be explicitly depended upon by end user applications. Will continue to be available in future releases within the same major series and should not have incompatible API changes in future releases within the same minor series, and ideally not within the same major series.
private	Intended for internal use only. Only tested according to internal usage. May not be safe for end user applications to use directly. Could change significantly or be removed in a future release without notice.
unsupported	If you see this value in a <code>module.xml</code> in a WildFly release, please file a bug report, as it is not applicable in WildFly. In EAP it has a meaning equivalent to "private" but that does not mean the module is "private" in WildFly; it could very easily be "public".
preview	May be explicitly depended upon by end user applications, but there are no guarantees of continued availability in future releases or that there will not be incompatible API changes. This is not a common classification in WildFly. It is not used in WildFly 10.
deprecated	May be explicitly depended upon by end user applications. Stable and reliable but an alternative should be sought. Will be removed in a future major release.

Note that these definitions are only applicable to WildFly. In EAP and other Red Hat products based on WildFly the same classifiers are used, with generally similar meaning, but the precise meaning is per the definitions on the Red Hat customer support portal.

If an application declares a direct dependency on a module marked "private", "unsupported" or "deprecated", during deployment a WARN message will be logged. The logging will be in log categories "org.jboss.as.dependency.private", "org.jboss.as.dependency.unsupported" and "org.jboss.as.dependency.deprecated" respectively. These categories are not used for other purposes, so once you feel sufficiently warned the logging can be safely suppressed by turning the log level for the relevant category to ERROR or higher.

Other than the WARN messages noted above, declaring a direct dependency on a non-public module has no impact on how WildFly processes the deployment.



17 Deployment Descriptors used In WildFly

This page gives a list and a description of all the valid deployment descriptors that a WildFly deployment can use. This document is a work in progress.

Descriptor	Location	Specification	Description	Info
<code>jboss-deployment-structure.xml</code>	META-INF or WEB-INF of the top level deployment		This file can be used to control class loading for the deployment	Class Loader WildFly
<code>beans.xml</code>	WEB-INF or META-INF	CDI	The presence of this descriptor (even if empty) activates CDI	Weld Reference Guide
<code>web.xml</code>	WEB-INF	Servlet	Web deployment descriptor	
<code>jboss-web.xml</code>	WEB-INF		JBoss Web deployment descriptor. This can be used to override settings from <code>web.xml</code> , and to set WildFly specific options	
<code>ejb-jar.xml</code>	WEB-INF of a war, or META-INF of an EJB jar	EJB	The EJB specification deployment descriptor	ejb-jar.xml specification



<code>jboss-ejb3.xml</code>	WEB-INF of a war, or META-INF of an EJB jar		The JBoss EJB deployment descriptor, this can be used to override settings from <code>ejb-jar.xml</code> , and to set WildFly specific settings	
<code>application.xml</code>	META-INF of an EAR	Java EE Platform Specification		application.x schema
<code>jboss-app.xml</code>	META-INF of an EAR		JBoss application deployment descriptor, can be used to override settings <code>application.xml</code> , and to set WildFly specific settings	
<code>persistence.xml</code>	META-INF	JPA	JPA descriptor used for defining persistence units	Hibernate Re Guide
<code>jboss-ejb-client.xml</code>	WEB-INF of a war, or META-INF of an EJB jar		Remote EJB settings. This file is used to setup the EJB client context for a deployment that is used for remote EJB invocations	EJB invocati from a remot server instar



<code>jbosscmp-jdbc.xml</code>	META-INF of an EJB jar		CMP deployment descriptor. Used to map CMP entity beans to a database. The format is largely unchanged from previous versions.	
<code>ra.xml</code>	META-INF of a rar archive		Spec deployment descriptor for resource adaptor deployments	IronJacamar Reference G Schema
<code>ironjacamar.xml</code>	META-INF of a rar archive		JBoss deployment descriptor for resource adaptor deployments	IronJacamar Reference G
<code>*-jms.xml</code>	META-INF or WEB-INF		JMS message destination deployment descriptor, used to deploy message destinations with a deployment	
<code>*-ds.xml</code>	META-INF or WEB-INF		Datasource deployment descriptor, use to bundle datasources with a deployment	DataSource Configuration



<code>application-client.xml</code>	META-INF of an application client jar	Java EE6 Platform Specification	The spec deployment descriptor for application client deployments	application-c schema
<code>jboss-client.xml</code>	META-INF of an application client jar		The WildFly specific deployment descriptor for application client deployments	
<code>jboss-webservices.xml</code>	META-INF for EJB webservice deployments or WEB-INF for POJO webservice deployments/EJB webservice endpoints bundled in .war		The JBossWS 4.0.x specific deployment descriptor for webservice endpoints	



18 Development Guidelines and Recommended Practices

The purpose of this page is to document tips and techniques that will assist developers in creating fast, secure, and reliable applications. It is also a place to note what you should **avoid** doing when developing applications.



19 EE Concurrency Utilities

19.1 Overview

EE Concurrency Utilities (JSR 236) is a technology introduced with Java EE 7, which adapts well known Java SE concurrency utilities to the Java EE application environment specifics. The Java EE application server is responsible for the creation (and shutdown) of every instance of the EE Concurrency Utilities, and provide these to the applications, ready to use.

The EE Concurrency Utilities support the propagation of the invocation context, capturing the existent context in the application threads to use in their own threads, the same way a logged-in user principal is propagated when a servlet invokes an EJB asynchronously. The propagation of the invocation context includes, by default, the class loading, JNDI and security contexts.

WildFly creates a single default instance of each EE Concurrency Utility type in all configurations within the distribution, as mandated by the specification, but additional instances, perhaps customised to better serve a specific usage, may be created through WildFly's EE Subsystem Configuration. To learn how to configure EE Concurrency Utilities please refer to [EE Concurrency Utilities Configuration](#). Additionally, the EE subsystem configuration also includes the configuration of which instance should be considered the default instance mandated by the Java EE specification, and such configuration is covered by [Default EE Bindings Configuration](#).



19.2 Context Service

The Context Service (`javax.enterprise.concurrent.ContextService`) is a brand new concurrency utility, which applications may use to build contextual proxies from existing objects.


A contextual proxy is an object that sets a invocation context, captured when created, whenever is invoked, before delegating the invocation to the original object.

Usage example:

```
public void onGet(...) {
    Runnable task = ...;
    Runnable contextualTask = contextService.createContextualProxy(task, Runnable.class);
    // ...
}
```


WildFly default configurations creates a single default instance of a Context Service, which may be retrieved through `@Resource` injection:

```
@Resource
private ContextService contextService;
```

 To retrieve instead a non default Context Service instance, `@Resource`'s `lookup` attribute needs to specify the JNDI name used in the wanted instance configuration. WildFly will always inject the default instance, no matter what's the name attribute value, if the `lookup` attribute is not defined.

Applications may alternatively use instead the standard JNDI API:

```
ContextService contextService = InitialContext.doLookup("java:comp/DefaultContextService");
```

 As mandated by the Java EE specification, the default Context Service instance's JNDI name is `java:comp/DefaultContextService`.

19.3 Managed Thread Factory

The Managed Thread Factory (`javax.enterprise.concurrent.ManagedThreadFactory`) allows Java EE applications to create Java threads. It is an extension of Java SE's Thread Factory (`java.util.concurrent.ThreadFactory`) adapted to the Java EE platform specifics.



Managed Thread Factory instances are managed by the application server, thus Java EE applications are forbidden to invoke any lifecycle related method.

In case the Managed Thread Factory is configured to use a Context Service, the application's thread context is captured when a thread creation is requested, and such context is propagated to the thread's Runnable execution.


Managed Thread Factory threads implement `javax.enterprise.concurrent.ManageableThread`, which allows an application to learn about termination status.

Usage example:

```
public void onGet(...) {
    Runnable task = ...;
    Thread thread = managedThreadFactory.newThread(task);
    thread.start();
    // ...
}
```


WildFly default configurations creates a single default instance of a Managed Thread Factory, which may be retrieved through `@Resource` injection:

```
@Resource
private ManagedThreadFactory managedThreadFactory;
```

 To retrieve instead a non default Managed Thread Factory instance, `@Resource`'s `lookup` attribute needs to specify the JNDI name used in the wanted instance configuration. WildFly will always inject the default instance, no matter what's the `name` attribute value, in case the `lookup` attribute is not defined.

Applications may alternatively use instead the standard JNDI API:

```
ManagedThreadFactory managedThreadFactory =
    InitialContext.doLookup("java:comp/DefaultManagedThreadFactory");
```

 As mandated by the Java EE specification, the default Managed Thread Factory instance's JNDI name is `java:comp/DefaultManagedThreadFactory`.



19.4 Managed Executor Service

The Managed Executor Service (`javax.enterprise.concurrent.ManagedExecutorService`) allows Java EE applications to submit tasks for asynchronous execution. It is an extension of Java SE's Executor Service (`java.util.concurrent.ExecutorService`) adapted to the Java EE platform requirements.

Managed Executor Service instances are managed by the application server, thus Java EE applications are forbidden to invoke any lifecycle related method.


In case the Managed Executor Service is configured to use a Context Service, the application's thread context is captured when the task is submitted, and propagated to the executor thread responsible for the task execution.

Usage example:

```
public void onGet(...) {
    Runnable task = ...;
    Future future = managedExecutorService.submit(task);
    // ...
}
```


WildFly default configurations creates a single default instance of a Managed Executor Service, which may be retrieved through `@Resource` injection:

```
@Resource
private ManagedExecutorService managedExecutorService;
```

 To retrieve instead a non default Managed Executor Service instance, `@Resource`'s `lookup` attribute needs to specify the JNDI name used in the wanted instance configuration. WildFly will always inject the default instance, no matter what's the `name` attribute value, in case the `lookup` attribute is not defined.

Applications may alternatively use instead the standard JNDI API:

```
ManagedExecutorService managedExecutorService =
InitialContext.doLookup("java:comp/DefaultManagedExecutorService");
```

 As mandated by the Java EE specification, the default Managed Executor Service instance's JNDI name is `java:comp/DefaultManagedExecutorService`.



19.5 Managed Scheduled Executor Service

The Managed Scheduled Executor Service (`javax.enterprise.concurrent.ManagedScheduledExecutorService`) allows Java EE applications to schedule tasks for asynchronous execution. It is an extension of Java SE's Executor Service (`java.util.concurrent.ScheduledExecutorService`) adapted to the Java EE platform requirements.

Managed Scheduled Executor Service instances are managed by the application server, thus Java EE applications are forbidden to invoke any lifecycle related method.


In case the Managed Scheduled Executor Service is configured to use a Context Service, the application's thread context is captured when the task is scheduled, and propagated to the executor thread responsible for the task execution.

Usage example:

```
public void onGet(...) {
    Runnable task = ...;
    ScheduledFuture future = managedScheduledExecutorService.schedule(task, 60,
        TimeUnit.SECONDS);
    // ...
}
```

WildFly default configurations creates a single default instance of a Managed Scheduled Executor Service, which may be retrieved through `@Resource` injection:

```
@Resource
private ManagedScheduledExecutorService managedScheduledExecutorService;
```

 To retrieve instead a non default Managed Scheduled Executor Service instance, `@Resource`'s `lookup` attribute needs to specify the JNDI name used in the wanted instance configuration. WildFly will always inject the default instance, no matter what's the `name` attribute value, in case the `lookup` attribute is not defined.

Applications may alternatively use instead the standard JNDI API:

```
ManagedScheduledExecutorService managedScheduledExecutorService =
    InitialContext.doLookup("java:comp/DefaultManagedScheduledExecutorService");
```



As mandated by the Java EE specification, the default Managed Scheduled Executor Service instance's JNDI name is `java:comp/DefaultManagedScheduledExecutorService`.



20 EJB 3 Reference Guide

This chapter details the extensions that are available when developing Enterprise Java Beans™ on WildFly 8.

Currently there is no support for configuring the extensions using an implementation specific descriptor file.

20.1 Resource Adapter for Message Driven Beans

Each Message Driven Bean must be connected to a resource adapter.

20.1.1 Specification of Resource Adapter using Metadata Annotations

The `ResourceAdapter` annotation is used to specify the resource adapter with which the MDB should connect.

The `value` of the annotation is the name of the deployment unit containing the resource adapter. For example `jms-ra.rar`.

For example:

```
@MessageDriven(messageListenerInterface = PostmanPat.class)
@ResourceAdapter("ejb3-rar.rar")
```




20.2 as Principal

Whenever a run-as role is specified for a given method invocation the default anonymous principal is used as the caller principal. This principal can be overridden by specifying a run-as principal.

20.2.1 Specification of Run-as Principal using Metadata Annotations

The `RunAsPrincipal` annotation is used to specify the run-as principal to use for a given method invocation.

The value of the annotation specifies the name of the principal to use. The actual type of the principal is undefined and should not be relied upon.

Using this annotation without specifying a run-as role is considered an error.

For example:

```
@RunAs("admin")
@RunAsPrincipal("MyBean")
```

20.3 Security Domain

Each Enterprise Java Bean [™] can be associated with a security domain. Only when an EJB is associated with a security domain will authentication and authorization be enforced.

20.3.1 Specification of Security Domain using Metadata Annotations

The `SecurityDomain` annotation is used to specify the security domain to associate with the EJB.

The value of the annotation is the name of the security domain to be used.

For example:

```
@SecurityDomain("other")
```

20.4 Transaction Timeout

For any newly started transaction a transaction timeout can be specified in seconds.



When a transaction timeout of 0 is used, then the actual transaction timeout will default to the domain configured default.

TODO: add link to tx subsystem

Although this is only applicable when using transaction attribute `REQUIRED` or `REQUIRES_NEW` the application server will not detect invalid setups.



New Transactions

Take care that even when transaction attribute `REQUIRED` is specified, the timeout will only be applicable if a **new** transaction is started.

20.4.1 Specification of Transaction Timeout with Metadata Annotations

The `TransactionTimeout` annotation is used to specify the transaction timeout for a given method.

The `value` of the annotation is the timeout used in the given `unit` granularity. It must be a positive integer or 0. Whenever 0 is specified the default domain configured timeout is used.

The `unit` specifies the granularity of the `value`. The actual value used is converted to seconds. Specifying a granularity lower than `SECONDS` is considered an error, even when the computed value will result in an even amount of seconds.

For example: `@TransactionTimeout(value = 10, unit = TimeUnit.SECONDS)`



20.4.2 Specification of Transaction Timeout in the Deployment Descriptor

The `trans-timeout` element is used to define the transaction timeout for business, home, component, and message-listener interface methods; no-interface view methods; web service endpoint methods; and timeout callback methods.

The `trans-timeout` element resides in the `urn:trans-timeout` namespace and is part of the standard `container-transaction` element as defined in the `jboss` namespace.

For the rules when a `container-transaction` is applicable please refer to EJB 3.1 FR 13.3.7.2.1.

Example of `trans-timeout`

`jboss-ejb3.xml`

```
<jboss:ejb-jar xmlns:jboss="http://www.jboss.com/xml/ns/javaee"
              xmlns="http://java.sun.com/xml/ns/javaee"
              xmlns:tx="urn:trans-timeout"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xsi:schemaLocation="http://www.jboss.com/xml/ns/javaee
http://www.jboss.org/j2ee/schema/jboss-ejb3-2_0.xsd
http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/ejb-jar_3_1.xsd
urn:trans-timeout http://www.jboss.org/j2ee/schema/trans-timeout-1_0.xsd"
              version="3.1"
              impl-version="2.0">
  <assembly-descriptor>
    <container-transaction>
      <method>
        <ejb-name>BeanWithTimeoutValue</ejb-name>
        <method-name>*</method-name>
        <method-intf>Local</method-intf>
      </method>
      <tx:trans-timeout>
        <tx:timeout>10</tx:timeout>
        <tx:unit>Seconds</tx:unit>
      </tx:trans-timeout>
    </container-transaction>
  </assembly-descriptor>
</jboss:ejb-jar>
```

20.5 Timer service

The service is responsible to call the registered timeout methods of the different session beans.



i A persistent timer will be identified by the name of the EAR, the name of the sub-deployment JAR and the Bean's name.
If one of those names are changed (e.g. EAR name contain a version) the timer entry became orphaned and the timer event will not longer be fired.

20.5.1 Single event timer

The timer is will be started once at the specified time.

In case of a server restart the timeout method of a persistent timer will only be called directly if the specified time is elapsed.

If the timer is not persistent (since EJB3.1 see 18.2.3) it will be not longer available if JBoss is restarted or the application is redeployed.

20.5.2 Recurring timer

The timer will be started at the specified first occurrence and after that point at each time if the interval is elapsed.

If the timer will be started during the last execution is not finished the execution will be suppressed with a warning to avoid concurrent execution.

In case of server downtime for a persistent timer, the timeout method will be called only once if one, or more than one, interval is elapsed.

If the timer is not persistent (since EJB3.1 see 18.2.3) it will not longer be active after the server is restarted or the application is redeployed.



20.5.3 Calendar timer

The timer will be started if the schedule expression match. It will be automatically deactivated and removed if there will be no next expiration possible, i.e. If you set a specific year.

For example:

```
@Schedule( ... dayOfMonth="1", month="1", year="2012")  
// start once at 01-01-2012 00:00:00
```

Programmatic calendar timer

If the timer is persistent it will be fetched at server start and the missed timeouts are called concurrent.

If a persistent timer contains an end date it will be executed once nevertheless how many times the execution was missed. Also a retry will be suppressed if the timeout method throw an Exception.

In case of such expired timer access to the given Timer object might throw a NoMoreTimeoutException or NoSuchObjectException.

If the timer is non persistent it will not longer be active after the server is restarted or the application is redeployed.

TODO: clarify whether this should happen concurrently/blocked or even fired only once like a recurring timer!

Annotated calendar timer

If the timer is non persistent it will not activated for missed events during the server is down. In case of server start the timer is scheduled based on the @Schedule annotation.

If the timer is persistent (default if not deactivated by annotation) all missed events are fetched at server start and the annotated timeout method is called concurrent.

TODO: clarify whether this should happen concurrently/blocked or even fired only once like a recurring timer!

20.6 Container interceptors

20.6.1 Overview

JBoss AS versions prior to WildFly8 allowed a JBoss specific way to plug-in user application specific interceptors on the server side so that those interceptors get invoked during an EJB invocation. Such interceptors differed from the typical (portable) spec provided Java EE interceptors. The Java EE interceptors are expected to run after the container has done necessary invocation processing which involves security context propagation, transaction management and other such duties. As a result, these Java EE interceptors come too late into the picture, if the user applications have to intercept the call before certain container specific interceptor(s) are run.



20.6.2 Typical EJB invocation call path on the server

A typical EJB invocation looks like this:

Client application

```
MyBeanInterface bean = lookupBean();  
  
bean.doSomething();
```

The invocation on the `bean.doSomething()` triggers the following (only relevant portion of the flow shown below):

1. WildFly specific interceptor (a.k.a container interceptor) 1
2. WildFly specific interceptor (a.k.a container interceptor) 2
3.
4. WildFly specific interceptor (a.k.a container interceptor) N
5. User application specific Java EE interceptor(s) (if any)
6. Invocation on the EJB instance's method

The WildFly specific interceptors include the security context propagation, transaction management and other container provided services. In some cases, the "container interceptors" (let's call them that) might even decide break the invocation flow and not let the invocation proceed (for example: due to the invoking caller not being among the allowed user roles who can invoke the method on the bean).

Previous versions of JBoss AS allowed a way to plug-in the user application specific interceptors (which relied on JBoss AS specific libraries) into this invocation flow so that they do run some application specific logic before the control reaches step#5 above. For example, AS5 allowed the use of JBoss AOP interceptors to do this.

WildFly 8 doesn't have such a feature.

20.6.3 Feature request for WildFly


There were many community users who requested for this feature to be made available in WildFly. As a result, <https://issues.jboss.org/browse/AS7-5897> JIRA was raised. This feature is now implemented.



20.6.4 Configuring container interceptors

As you can see from the JIRA <https://issues.jboss.org/browse/AS7-5897>, one of the goals of this feature implementation was to make sure that we don't introduce any new WildFly specific library dependencies for the container interceptors. So we decided to allow the Java EE interceptors (which are just POJO classes with lifecycle callback annotations) to be used as container interceptors. As such you won't need any dependency on any WildFly specific libraries. That will allow us to support this feature for a longer time in future versions of WildFly.

Furthermore, configuring these container interceptors is similar to configuring the Java EE interceptors for EJBs. In fact, it uses the same xsd elements that are allowed in `ejb-jar.xml` for 3.1 version of `ejb-jar` deployment descriptor.

 Container interceptors can only be configured via deployment descriptors. There's no annotation based way to configure container interceptors. This was an intentional decision, taken to avoid introducing any WildFly specific library dependency for the annotation.

Configuring the container interceptors can be done in `jboss-ejb3.xml` file, which then gets placed under the `META-INF` folder of the EJB deployment, just like the `ejb-jar.xml`. Here's an example of how the container interceptor(s) can be configured in `jboss-ejb3.xml`:

**jboss-ejb3.xml**

```
<jboss xmlns="http://www.jboss.com/xml/ns/javaee"
  xmlns:jee="http://java.sun.com/xml/ns/javaee"
  xmlns:ci="urn:container-interceptors:1.0">

  <jee:assembly-descriptor>
    <ci:container-interceptors>
      <!-- Default interceptor -->
      <jee:interceptor-binding>
        <ejb-name>*</ejb-name>

<interceptor-class>org.jboss.as.test.integration.ejb.container.interceptor.ContainerInterceptorOne
</jee:interceptor-binding>
      <!-- Class level container-interceptor -->
      <jee:interceptor-binding>
        <ejb-name>AnotherFlowTrackingBean</ejb-name>

<interceptor-class>org.jboss.as.test.integration.ejb.container.interceptor.ClassLevelContainerInte
</jee:interceptor-binding>
      <!-- Method specific container-interceptor -->
      <jee:interceptor-binding>
        <ejb-name>AnotherFlowTrackingBean</ejb-name>

<interceptor-class>org.jboss.as.test.integration.ejb.container.interceptor.MethodSpecificContainer
<method>
      <method-name>echoWithMethodSpecificContainerInterceptor</method-name>
    </method>
    </jee:interceptor-binding>
    <!-- container interceptors in a specific order -->
    <jee:interceptor-binding>
      <ejb-name>AnotherFlowTrackingBean</ejb-name>
      <interceptor-order>

<interceptor-class>org.jboss.as.test.integration.ejb.container.interceptor.ClassLevelContainerInte
<interceptor-class>org.jboss.as.test.integration.ejb.container.interceptor.MethodSpecificContainer
<interceptor-class>org.jboss.as.test.integration.ejb.container.interceptor.ContainerInterceptorOne
</interceptor-order>
      <method>
        <method-name>echoInSpecificOrderOfContainerInterceptors</method-name>
      </method>
    </jee:interceptor-binding>
  </ci:container-interceptors>
</jee:assembly-descriptor>
</jboss>
```

- The usage of `urn:container-interceptors:1.0` namespace which allows the container-interceptors elements to be configured
- The container-interceptors element which contain the interceptor bindings
- The interceptor bindings themselves are the same elements as what the EJB3.1 xsd allows for standard Java EE interceptors
- The interceptors can be bound either to all EJBs in the deployment (using the the `*` wildcard) or individual bean level (using the specific EJB name) or at specific method level for the EJBs.



The xsd for the urn:container-interceptors:1.0 namespace is available here

<https://github.com/jbossas/jboss-as/blob/master/ejb3/src/main/resources/jboss-ejb-container-interceptor>

The interceptor classes themselves are simple POJOs and use the `@javax.annotation.AroundInvoke` to mark the around invoke method which will get invoked during the invocation on the bean. Here's an example of the interceptor:

Example of container interceptor

```
public class ClassLevelContainerInterceptor {
    @AroundInvoke
    private Object iAmAround(final InvocationContext invocationContext) throws Exception {
        return this.getClass().getName() + " " + invocationContext.proceed();
    }
}
```

20.6.5 Container interceptor positioning in the interceptor chain

The container interceptors configured for a EJB are guaranteed to be run before the WildFly provided security interceptors, transaction management interceptors and other such interceptors thus allowing the user application specific container interceptors to setup any relevant context data before the invocation proceeds.

20.6.6 Semantic difference between container interceptor(s) and Java EE interceptor(s) API

Although the container interceptors are modeled to be similar to the Java EE interceptors, there are some differences in the API semantics. One such difference is that invoking on `javax.interceptor.InvocationContext.getTarget()` method is illegal for container interceptors since these interceptors are invoked way before the EJB components are setup or instantiated.

20.6.7 Testcase

This testcase in the WildFly codebase can be used for reference for implementing container interceptors in user applications

<https://github.com/jbossas/jboss-as/blob/master/testsuite/integration/basic/src/test/java/org/jboss/as/test/integ>




20.7 EJB3 Clustered Database Timers

20.7.1 Overview

Wildfly now supports clustered database backed timers. The clustering support is provided through the database, and as a result it is not intended to be a super high performance solution that supports thousands of timers going off a second, however properly tuned it should provide sufficient performance for most use cases.

Note that database timers can also be used in non-clustered mode.

 Note that for this to work correctly the underlying database must support the `READ_COMMITTED` or `SERIALIZABLE` isolation mode and the datasource must be configured accordingly



20.7.2 Setup

In order to use clustered timers it is necessary to add a database backed timer store. This can be done from the CLI with the following command:

```
/subsystem=ejb3/service=timer-service/database-data-store=my-clustered-store:add(allow-execution=t:datasource-jndi-name='java:/MyDatasource', refresh-interval=60000, database='postgresql', partition='mypartition')
```

An explanation of the parameters is below:

- **allow-execution** - If this node is allowed to execute timers. If this is false then timers added on this node will be added to the database for another node to execute. This allows you to limit timer execution to a few nodes in a cluster, which can greatly reduce database load for large clusters.
- **datasource-jndi-name** - The datasource to use
- **refresh-interval** - The refresh interval in milliseconds. This is the period of time that must elapse before this node will check the database for new timers added by other nodes. A smaller value means that timers will be picked up more quickly, however it will result in more load on the database. This is most important to tune if you are adding timers that will expire quickly. If the node that added the timer cannot execute it (e.g. because it has failed or because allow-execution is false), this timer may not be executed until a node has refreshed.
- **database** - Define the type of database that is in use. Some SQL statements are customised by database, and this tells the data store which version of the SQL to use.
Without this attribute the server try to detected the type automatically, current supported types are *postgresql*, *mysql*, *oracle*, *db2*, *hsqldb* and *h2*.
Note that this SQL resides in the file
modules/system/layers/base/org/jboss/as/ejb3/main/timers/timer-sql.properties
And as such is it possible to modify the SQL that is executed or add support for new databases by adding new DB specific SQL to this file (if you do add support for a new database it would be greatly appreciated if you could contribute the SQL back to the project).
- **partition** - A node will only see timers from other nodes that have the same partition name. This allows you to break a large cluster up into several smaller clusters, which should improve performance. e.g. instead of having a cluster of 100 nodes, where all hundred are trying to execute and refresh the same timers, you can create 20 clusters of 5 nodes by giving ever group of 5 a different partition name.

Non clustered timers

Note that you can still use the database data store for non-clustered timers, in which case set the refresh interval to zero and make sure that every node has a unique partition name (or uses a different database).



20.7.3 Using clustered timers in a deployment

It is possible to use the data store as default for all applications by changing the default-data-store within the ejb3 subsystem:

```
<timer-service thread-pool-name="timer" default-data-store="clustered-store">
  <data-stores>
    <database-data-store name="clustered-store"
datasource-jndi-name=" java:jboss/datasources/ExampleDS" partition="timer"/>
  </data-stores>
</timer-service>
```

Another option is to use a separate data store for specific applications, all that is required is to set the timer data store name in jboss-ejb3.xml:

```
<?xml version="1.1" encoding="UTF-8"?>
<jboss:ejb-jar xmlns:jboss="http://www.jboss.com/xml/ns/javaee"
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:timer="urn:timer-service:1.0"
  xsi:schemaLocation="http://www.jboss.com/xml/ns/javaee
http://www.jboss.org/j2ee/schema/jboss-ejb3-2_0.xsd
  http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/ejb-jar_3_1.xsd"
  version="3.1"
  impl-version="2.0">
  <assembly-descriptor>
    <timer:timer>
      <ejb-name>*</ejb-name>
      <timer:persistence-store-name>my-clustered-store</timer:persistence-store-name>
    </timer:timer>
  </assembly-descriptor>
</jboss:ejb-jar>
```

20.7.4 Technical details

Internally every node that is allowed to execute timers schedules a timeout for every timer it knows about. When this timeout expires then this node attempts to 'lock' the timer, by updating its state to running. The query this executes looks like:

```
UPDATE JBOSS_EJB_TIMER SET TIMER_STATE=? WHERE ID=? AND TIMER_STATE<>? AND NEXT_DATE=?;
```

Due to the use of a transaction and READ_COMMITTED or SERIALIZABLE isolation mode only one node will succeed in updating the row, and this is the node that the timer will run on.



20.8 EJB3 subsystem configuration guide

This page lists the options that are available for configuring the EJB subsystem.

A complete example of the config is shown below, with a full explanation of each



```
<subsystem xmlns="urn:jboss:domain:ejb3:1.2">
  <session-bean>
    <stateless>
      <bean-instance-pool-ref pool-name="slsb-strict-max-pool"/>
    </stateless>
    <stateful default-access-timeout="5000" cache-ref="simple" clustered-cache-ref="clustered"/>
    <singleton default-access-timeout="5000"/>
  </session-bean>
  <mdb>
    <resource-adapter-ref resource-adapter-name="hornetq-ra"/>
    <bean-instance-pool-ref pool-name="mdb-strict-max-pool"/>
  </mdb>
  <entity-bean>
    <bean-instance-pool-ref pool-name="entity-strict-max-pool"/>
  </entity-bean>
  <pools>
    <bean-instance-pools>
      <strict-max-pool name="slsb-strict-max-pool" max-pool-size="20"
instance-acquisition-timeout="5" instance-acquisition-timeout-unit="MINUTES"/>
      <strict-max-pool name="mdb-strict-max-pool" max-pool-size="20"
instance-acquisition-timeout="5" instance-acquisition-timeout-unit="MINUTES"/>
      <strict-max-pool name="entity-strict-max-pool" max-pool-size="100"
instance-acquisition-timeout="5" instance-acquisition-timeout-unit="MINUTES"/>
    </bean-instance-pools>
  </pools>
  <caches>
    <cache name="simple" aliases="NoPassivationCache"/>
    <cache name="passivating" passivation-store-ref="file" aliases="SimpleStatefulCache"/>
    <cache name="clustered" passivation-store-ref="infinispan" aliases="StatefulTreeCache"/>
  </caches>
  <passivation-stores>
    <file-passivation-store name="file"/>
    <cluster-passivation-store name="infinispan" cache-container="ejb"/>
  </passivation-stores>
  <async thread-pool-name="default"/>
  <timer-service thread-pool-name="default">
    <data-store path="timer-service-data" relative-to="jboss.server.data.dir"/>
  </timer-service>
  <remote connector-ref="remoting-connector" thread-pool-name="default"/>
  <thread-pools>
    <thread-pool name="default">
      <max-threads count="10"/>
      <keepalive-time time="100" unit="milliseconds"/>
    </thread-pool>
  </thread-pools>
  <iiop enable-by-default="false" use-qualified-name="false"/>
  <in-vm-remote-interface-invocation pass-by-value="false"/> <!-- Warning see notes below about
possible issues -->
</subsystem>
```



20.8.1 <session-bean>

<stateless>

This element is used to configure the instance pool that is used by default for stateless session beans. If it is not present stateless session beans are not pooled, but are instead created on demand for every invocation. The instance pool can be overridden on a per deployment or per bean level using `jboss-ejb3.xml` or the `org.jboss.ejb3.annotation.Pool` annotation. The instance pools themselves are configured in the `<pools>` element.

<stateful>

This element is used to configure Stateful Session Beans.

- `default-access-timeout` This attribute specifies the default time concurrent invocations on the same bean instance will wait to acquire the instance lock. It can be overridden via the deployment descriptor or via the `javax.ejb.AccessTimeout` annotation.
- `cache-ref` This attribute is used to set the default cache for non-clustered beans. It can be overridden by `jboss-ejb3.xml`, or via the `org.jboss.ejb3.annotation.Cache` annotation.
- `clustered-cache-ref` This attribute is used to set the default cache for clustered beans.

<singleton>

This element is used to configure Singleton Session Beans.

- `default-access-timeout` This attribute specifies the default time concurrent invocations will wait to acquire the instance lock. It can be overridden via the deployment descriptor or via the `javax.ejb.AccessTimeout` annotation.

20.8.2 <mdb>

<resource-adaptor-ref>

This element sets the default resource adaptor for Message Driven Beans.

<bean-instance-pool-ref>

This element is used to configure the instance pool that is used by default for Message Driven Beans. If it is not present they are not pooled, but are instead created on demand for every invocation. The instance pool can be overridden on a per deployment or per bean level using `jboss-ejb3.xml` or the `org.jboss.ejb3.annotation.Pool` annotation. The instance pools themselves are configured in the `<pools>` element.



20.8.3 <entity-bean>

This element is used to configure the behavior for EJB2 EntityBeans.

<bean-instance-pool-ref>

This element is used to configure the instance pool that is used by default for Entity Beans. If it is not present they are not pooled, but are instead created on demand for every invocation. The instance pool can be overridden on a per deployment or per bean level using `jboss-ejb3.xml` or the `org.jboss.ejb3.annotation.Pool` annotation. The instance pools themselves are configured in the `<pools>` element.

20.8.4

20.8.5 <pools>

20.8.6 <caches>

20.8.7 <passivation-stores>

20.8.8 <async>

This element enables async EJB invocations. It is also used to specify the thread pool that these invocations will use.

20.8.9 <timer-service>

This element enables the EJB timer service. It is also used to specify the thread pool that these invocations will use.

<data-store>

This is used to configure the directory that persistent timer information is saved to.



20.8.10 <remote>

This is used to enable remote EJB invocations. It specifies the remoting connector to use (as defined in the remoting subsystem configuration), and the thread pool to use for remote invocations.

20.8.11 <thread-pools>

This is used to configure the thread pools used by async, timer and remote invocations.

20.8.12 <iiop>

This is used to enable IIOP (i.e. CORBA) invocation of EJB's. If this element is present then the JacORB subsystem must also be installed. It supports the following two attributes:

- `enable-by-default` If this is true then all EJB's with EJB2.x home interfaces are exposed via IIOP, otherwise they must be explicitly enabled via `jboss-ejb3.xml`.
- `use-qualified-name` If this is true then EJB's are bound to the corba naming context with a binding name that contains the application and modules name of the deployment (e.g. `myear/myejbjar/MyBean`), if this is false the default binding name is simply the bean name.



20.8.13 <in-vm-remote-interface-invocation>

By default remote interface invocations use pass by value, as required by the EJB spec. This element can be used to enable pass by reference, which can give you a performance boost. Note WildFly will do a shallow check to see if the caller and the EJB have access to the same class definitions, which means if you are passing something such as a `List<MyObject>`, WildFly only checks the `List` to see if it is the same class definition on the call & EJB side. If the top level class definition is the same, JBoss will make the call using pass by reference, which means that if `MyObject` or any objects beneath it are loaded from different classloaders, you would get a `ClassCastException`. If the top level class definitions are loaded from different classloaders, JBoss will use pass by value. JBoss cannot do a deep check of all of the classes to ensure no `ClassCastExceptions` will occur because doing a deep check would eliminate any performance boost you would have received by using call by reference. It is recommended that you configure pass by reference only on callers that you are sure will use the same class definitions and not globally. This can be done via a configuration in the `jboss-ejb-client.xml` as shown below.

To configure a caller/client use pass by reference, you configure your top level deployment with a `META-INF/jboss-ejb-client.xml` containing:

```
<jboss-ejb-client xmlns="urn:jboss:ejb-client:1.0">
  <client-context>
    <ejb-receivers local-receiver-pass-by-value="false"/>
  </client-context>
</jboss-ejb-client>
```



20.9 EJB IIOP Guide

20.9.1 Enabling IIOP

To enable IIOP you must have the JacORB subsystem installed, and the `<iiop/>` element present in the `ejb3` subsystem configuration. The `standalone-full.xml` configuration that comes with the distribution has both of these enabled.

The `<iiop/>` element takes two attributes that control the default behaviour of the server, for full details see [EJB3 subsystem configuration guide](#).

20.9.2 Enabling JTS

To enable JTS simply add a `<jts/>` element to the transactions subsystem configuration.

It is also necessary to enable the JacORB transactions interceptor as shown below.

```
<subsystem xmlns="urn:jboss:domain:jacorb:1.1">
  <orb>
    <initializers transactions="on"/>
  </orb>
</subsystem>
```

20.9.3 Dynamic Stub's

Downloading stubs directly from the server is no longer supported. If you do not wish to pre-generate your stub classes JDK Dynamic stubs can be used instead. To enable JDK dynamic stubs simply set the `com.sun.CORBA.ORBUseDynamicStub` system property to `true`.

20.9.4 Configuring EJB IIOP settings via `jboss-ejb3.xml`

TODO

20.10 `jboss-ejb3.xml` Reference

`jboss-ejb3.xml` is a custom deployment descriptor that can be placed in either `ejb-jar` or `war` archives. If it is placed in an `ejb-jar` then it must be placed in the `META-INF` folder, in a web archive it must be placed in the `WEB-INF` folder.

The contents of `jboss-ejb3.xml` are merged with the contents of `ejb-jar.xml`, with the `jboss-ejb3.xml` items taking precedence.



20.10.1 Example File

A simple example is shown below:

```
<?xml version="1.1" encoding="UTF-8"?>
<jboss:ejb-jar xmlns:jboss="http://www.jboss.com/xml/ns/javaee"
              xmlns="http://java.sun.com/xml/ns/javaee"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xmlns:s="urn:security:1.1"
              xsi:schemaLocation="http://www.jboss.com/xml/ns/javaee
http://www.jboss.org/j2ee/schema/jboss-ejb3-2_0.xsd http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/ ejb-jar_3_1.xsd"
              version="3.1"
              impl-version="2.0">
  <enterprise-beans>
    <message-driven>
      <ejb-name>ReplyingMDB</ejb-name>

      <ejb-class>org.jboss.as.test.integration.ejb.mdb.messagedestination.ReplyingMDB</ejb-class>
      <activation-config>
        <activation-config-property>

          <activation-config-property-name>destination</activation-config-property-name>
          <activation-config-property-value>java:jboss/mdbtest/messageDestinationQueue
          </activation-config-property-value>
        </activation-config-property>
      </activation-config>
    </message-driven>
  </enterprise-beans>
  <assembly-descriptor>
    <s:security>
      <ejb-name>DDMyDomainSFSB</ejb-name>
      <s:security-domain>myDomain</s:security-domain>
      <s:run-as-principal>myPrincipal</s:run-as-principal>
    </s:security>
  </assembly-descriptor>
</jboss:ejb-jar>
```

As you can see the format is largely similar to `ejb-jar.xml`, in fact they even use the same namespaces, however `jboss-ejb3.xml` adds some additional namespaces of its own to allow for configuring non-spec info. The format of the standard `http://java.sun.com/xml/ns/javaee` is well documented elsewhere, this document will cover the non-standard namespaces.

The root namespace `http://www.jboss.com/xml/ns/javaee`

Assembly descriptor namespaces

The following namespaces can all be used in the `<assembly-descriptor>` element. They can be used to apply their configuration to a single bean, or to all beans in the deployment by using `*` as the `ejb-name`.



The security namespace `urn:security`

This allows you to set the security domain and the run-as principal for an EJB.

```
<s:security>
  <ejb-name>*/</ejb-name>
  <s:security-domain>myDomain</s:security-domain>
  <s:run-as-principal>myPrincipal</s:run-as-principal>
</s:security>
```

The resource adaptor namespace `urn:resource-adapter-binding`

This allows you to set the resource adaptor for an MDB.

```
<r:resource-adapter-binding>
  <ejb-name>*/</ejb-name>
  <r:resource-adapter-name>myResourceAdaptor</r:resource-adapter-name>
</r:resource-adapter-binding>
```

The IIOp namespace `urn:iioP`

The IIOp namespace is where IIOp settings are configured. As there are quite a large number of options these are covered in the [IIOp guide](#).

The pool namespace `urn:ejb-pool:1.0`

This allows you to select the pool that is used by the SLSB or MDB. Pools are defined in the server configuration (i.e. `standalone.xml` or `domain.xml`)

```
<p:pool>
  <ejb-name>*/</ejb-name>
  <p:bean-instance-pool-ref>my-pool</p:bean-instance-pool-ref>
</p:pool>
```

The cache namespace `urn:ejb-cache:1.0`

This allows you to select the cache that is used by the SFSB. Caches are defined in the server configuration (i.e. `standalone.xml` or `domain.xml`)

```
<c:cache>
  <ejb-name>*/</ejb-name>
  <c:cache-ref>my-cache</c:cache-ref>
</c:cache>
```

The clustering namespace `urn:clustering:1.0`

This namespace is deprecated and as of WildFly 8 its use has no effect. The clustering behavior of EJBs is determined by the profile in use on the server.



20.11 Message Driven Beans Controlled Delivery

There are three mechanisms in WildFly that allow controlling if a specific MDB is actively receiving or not messages:

- delivery active
- delivery groups
- clustered singleton

We will see each one of them in the following sections.

20.11.1 Delivery Active

Delivery active is simply an attribute associated with the MDB that indicates if the MDB is receiving messages or not. If an MDB is not currently receiving messages, the messages will be saved in the queue or topic for later, according to the rules of the topic/queue.

You can configure delivery active using xml or annotations, and you can change its value after deployment using the cli.

- jboss-ejb3.xml:

In the jboss-ejb3 xml file, configure the value of active as false to mark that the MDB will not be receiving messages as soon as it is deployed:

```
<?xml version="1.1" encoding="UTF-8"?>
<jboss:ejb-jar xmlns:jboss="http://www.jboss.com/xml/ns/javaee"
               xmlns="http://java.sun.com/xml/ns/javaee"
               xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xmlns:d="urn:delivery-active:1.1"
               xsi:schemaLocation="http://www.jboss.com/xml/ns/javaee
http://www.jboss.org/j2ee/schema/jboss-ejb3-2_0.xsd http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/ejb-jar_3_1.xsd" version="3.1"
               impl-version="2.0">
  <assembly-descriptor>
    <d:delivery>
      <ejb-name>HelloWorldQueueMDB</ejb-name>
      <d:active>>false</d:active>
    </d:delivery>
  </assembly-descriptor>
</jboss:ejb-jar>
```

You can use a wildcard "*" in the place of ejb-name if you want to apply that active value to all MDBs in your application.

- annotation

Alternatively, you can use the `org.jboss.ejb3.annotation.DeliveryActive` annotation, as in the example below:



```
@MessageDriven(name = "HelloWorldMDB", activationConfig = {

    @ActivationConfigProperty(propertyName = "destinationType", propertyValue = "javax.jms.Queue"),

    @ActivationConfigProperty(propertyName = "destination", propertyValue =
"queue/HELLOWORLDMDBQueue"),

    @ActivationConfigProperty(propertyName = "acknowledgeMode", propertyValue = "Auto-acknowledge")
})

@DeliveryActive(false)

public class HelloWorldMDB implements MessageListener {
    public void onMessage(Message rcvMessage) {
        // ...
    }
}
```

Start-delivery and Stop-Delivery

These management operations dynamically change the value of the active attribute, enabling or disabling delivery for the MDB. at runtime To use them, connect to the Wildfly instance you want to manage, then enter the path of the MDB you want to manage delivery for:

```
[standalone@localhost:9990 /] cd
deployment=jboss-helloworld-mdb.war/subsystem=ejb3/message-driven-bean=HelloWorldMDB

[standalone@localhost:9990 message-driven-bean=HelloWorldMDB] :stop-delivery
{"outcome" => "success"}

[standalone@localhost:9990 message-driven-bean=HelloWorldMDB] :start-delivery
{"outcome" => "success"}
```

20.11.2 Delivery Groups

Delivery groups provide a straightforward way to manage delivery for a group of MDBs. Every MDB belonging to a delivery group has delivery active if and only if that group is active, and has delivery inactive whenever the group is not active.

You can add a delivery group to the ejb3 subsystem using either the subsystem xml or cli. Next, we will see examples of each case. In those examples we will add only a single delivery group, but keep in mind that you can add as many delivery groups as you need to a Wildfly instance.

- the ejb3 subsystem xml (located in your configuration xml, such as standalone.xml)



```
<subsystem xmlns="urn:jboss:domain:ejb3:4.0">
  ...
  <mdb>
    ...
    <delivery-groups>
      <delivery-group name="mdb-group-name" active="true"/>
    </delivery-groups>
  </mdb>
  ...
</subsystem>
```

The example above adds a delivery group named “mdb-group-name” (you can use whatever name suits you best as the group name). The “true” active attribute indicates that all MDBs belonging to that group will have delivery active right after deployment. If you mark that attribute as false, you are indicating that every MDB belonging to the group will not start receiving messages after deployment, a condition that will remain until the group becomes active.

- jboss-cli

You can add a mdb-delivery-group using the add command as below:

```
[standalone@localhost:9990 /] ./subsystem=ejb3/mdb-delivery-group=mdb-group-name:add
{"outcome" => "success"}
```




Reading and Writing the Delivery State of a Delivery Group

You can check whether delivery is active for a group by reading the active attribute, which defaults to true:

```
[standalone@localhost:9990 /]
./subsystem=ejb3/mdb-delivery-group=mdb-group-name:read-attribute(name=active)
{ "outcome" => "success", "result" => true }
```

To make the the delivery-group inactive, just write the active attribute with a false value:

```
[standalone@localhost:9990 /]
./subsystem=ejb3/mdb-delivery-group=mdb-group-name:write-attribute(name=active,value=false)
{ "outcome" => "success" }

[standalone@localhost:9990 /]
./subsystem=ejb3/mdb-delivery-group=mdb-group-name:read-attribute(name=active)
{ "outcome" => "success", "result" => false }
```

To make it active again, write the attribute with a true value:

```
[standalone@localhost:9990 /]
./subsystem=ejb3/mdb-delivery-group=mdb-group-name:write-attribute(name=active,value=true)
{ "outcome" => "success" }

[standalone@localhost:9990 /]
./subsystem=ejb3/mdb-delivery-group=mdb-group-name:read-attribute(name=active)
{ "outcome" => "success", "result" => true }
```

Using Delivery Groups

To mark that an MDB belongs to a delivery-group, declare so in the jboss-ejb3.xml file:



```
<?xml version="1.1" encoding="UTF-8"?>

<jboss:ejb-jar xmlns:jboss="http://www.jboss.com/xml/ns/javaee"
              xmlns="http://java.sun.com/xml/ns/javaee"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xmlns:d="urn:delivery-active:1.1"
              xsi:schemaLocation="http://www.jboss.com/xml/ns/javaee
http://www.jboss.org/j2ee/schema/jboss-ejb3-2_0.xsd http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/ejb-jar_3_1.xsd"
              version="3.1"
              impl-version="2.0">
  <assembly-descriptor>
    <d:delivery>
      <ejb-name>HelloWorldMDB</ejb-name>
      <d:group>mdb-delivery-group</d:group>
    </d:delivery>
  </assembly-descriptor>
</jboss:ejb-jar>
```

You can also use a wildcard to mark that all MDBs in your application belong to a delivery-group. In the following example, we add all MDBs in the application to group1, except for HelloWorldMDB, that is added to group2:

```
<?xml version="1.1" encoding="UTF-8"?>
<jboss:ejb-jar xmlns:jboss="http://www.jboss.com/xml/ns/javaee"
              xmlns="http://java.sun.com/xml/ns/javaee"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xmlns:d="urn:delivery-active:1.1"
              xsi:schemaLocation="http://www.jboss.com/xml/ns/javaee
http://www.jboss.org/j2ee/schema/jboss-ejb3-2_0.xsd http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/ejb-jar_3_1.xsd"
              version="3.1"
              impl-version="2.0">
  <assembly-descriptor>
    <d:delivery>
      <ejb-name>*</ejb-name>
      <d:group>group1</d:group>
    </d:delivery>
    <d:delivery>
      <ejb-name>HelloWorldMDB</ejb-name>
      <d:group>group2</d:group>
    </d:delivery>
  </assembly-descriptor>
</jboss:ejb-jar>
```

Another option is to use `org.jboss.ejb3.annotation.DeliveryGroup` annotation on each MDB class belonging to a group:



```
@MessageDriven(name = "HelloWorldQueueMDB", activationConfig = {
    @ActivationConfigProperty(propertyName = "destinationType", propertyValue = "javax.jms.Queue"),
    @ActivationConfigProperty(propertyName = "destination", propertyValue =
"queue/HELLOWORLDMDBQueue"),
    @ActivationConfigProperty(propertyName = "acknowledgeMode", propertyValue = "Auto-acknowledge")
})

@DeliveryGroup("group2")

public class HelloWorldMDB implements MessageListener {
    ...
}
```

A MDB cannot belong to more than one delivery group. Also, all the delivery-groups used by an application must be installed in the Wildfly server upon deployment, or the deployment will fail with a message stating that the delivery-group is missing.

20.11.3 Clustered Singleton Delivery

Delivery can be marked as singleton in a clustered environment. In this case, only one node in the cluster will have delivery active for that MDB, whereas in all other nodes, delivery will be inactive. This option can be used for applications that are deployed in all nodes of the cluster. Such applications will be active in all nodes of the cluster, except for the MDBs that are marked as clustered singleton. For those MDBs, only one cluster node will be processing their messages. In case that node stops, another node will have delivery activated, guaranteeing that there is always one node processing the messages. This node is what we call the MDB clustered singleton master node.

Notice that applications using clustered singleton delivery can only be deployed in clustered Wildfly servers (i.e., servers that are using the ha configuration).

To mark delivery as clustered singleton, you can use the `jboss-ejb3.xml` or the `@ClusteredSingleton` annotation:

- `jboss-ejb3.xml`:



```
<?xml version="1.1" encoding="UTF-8"?>
<jboss:ejb-jar xmlns:jboss="http://www.jboss.com/xml/ns/javaee"
               xmlns="http://java.sun.com/xml/ns/javaee"
               xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xmlns:c="urn:clustering:1.1"
               xsi:schemaLocation="http://www.jboss.com/xml/ns/javaee
http://www.jboss.org/j2ee/schema/jboss-ejb3-2_0.xsd http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/ejb-jar_3_1.xsd"
               version="3.1"
               impl-version="2.0">
  <assembly-descriptor>
    <c:clustering>
      <ejb-name>HelloWorldMDB</ejb-name>
      <c:clustered-singleton>true</c:clustered-singleton>
    </c:clustering>
  </assembly-descriptor>
</jboss:ejb-jar>
```

As in the previous `jboss-ejb3.xml` examples, a wildcard can be used in the place of the `ejb-name` to indicate that all MDBs in the application are singleton clustered.

- annotation

You can use the `org.jboss.ejb3.annotation.ClusteredSingleton` annotation to mark an MDB as clustered singleton:

```
@MessageDriven(name = "HelloWorldQueueMDB", activationConfig = {
    @ActivationConfigProperty(propertyName = "destinationType", propertyValue = "javax.jms.Queue"),
    @ActivationConfigProperty(propertyName = "destination", propertyValue =
"queue/HELLOWORLDMDBQueue"),
    @ActivationConfigProperty(propertyName = "acknowledgeMode", propertyValue = "Auto-acknowledge")
})

@ClusteredSingleton

public class HelloWorldMDB implements MessageListener { ... }
```



20.11.4 Using Multiple MDB Delivery Control Mechanisms

The previous delivery control mechanisms can be used together in a single MDB. In this case, they work as a set of restrictions for delivery to be active in a MDB.

For example, if an MDB belongs to a delivery group and is also a clustered singleton MDB, the delivery will be active for that MDB only if the delivery group is active in the cluster node that was elected as the singleton master.

Also, if you use `jboss-cli` to `stopDelivery` on a MDB that belongs to a delivery group, the MDB will stop receiving messages in case that group was active. If that group was not active, the MDB will continue in the same, inactive state. But, once that group is active, the MDB will not receive messages, unless a `startDelivery` operation is executed to revert the previously executed `stopDelivery` operation.

Invoking `stopDelivery` on an MDB that is marked as clustered singleton will work in a similar way: no visible effect if the current node is not the clustered singleton master; but it will stop delivery of messages for that MDB if the current node is the clustered singleton master. If the current node is not the master, but eventually becomes so, the delivery of messages will not be active for that MDB, unless a `startDelivery` operation is invoked.

In other words, when more than one delivery control mechanism is used in conjunction, they act as a set of restrictions that need all to be true in order for the MDB to receive messages:

- **delivery-group + stop-delivery**: the delivery group needs to be active and the delivery needs to be started in order for that MDB to start receiving messages;
- **delivery-group + clustered singleton**: the delivery group needs to be active and the current node needs to be the clustered singleton master node in order for that MDB to start receiving messages;
- **delivery-group + clustered singleton + stop-delivery**: as above, delivery-group active, current node equals the clustered singleton master node, plus, start-delivery needs to be invoked on that MDB, only with these three factors being true the MDB will start receiving messages.

20.12 Securing EJBs

20.12.1 Overview

The Java EE spec specifies certain annotations (like `@RolesAllowed`, `@PermitAll`, `@DenyAll`) which can be used on EJB implementation classes and/or the business method implementations of the beans. Like with all other configurations, these security related configurations can also be done via the deployment descriptor (`ejb-jar.xml`). We *won't* be going into the details of Java EE specific annotations/deployment descriptor configurations in this chapter but instead will be looking at the vendor specific extensions to the security configurations.



20.12.2 Security Domain

The Java EE spec doesn't mandate a specific way to configure security domain for a bean. It leaves it to the vendor implementations to allow such configurations, the way they wish. In WildFly 8, the use of `@org.jboss.ejb3.annotation.SecurityDomain` annotation allows the developer to configure the security domain for a bean. Here's an example:

```
import org.jboss.ejb3.annotation.SecurityDomain;

import javax.ejb.Stateless;

@Stateless

@SecurityDomain("other")

public class MyBean ...

{

....
```

The use of `@SecurityDomain` annotation lets the developer to point the container to the name of the security domain which is configured in the EJB3 subsystem in the standalone/domain configuration. The configuration of the security domain in the EJB3 subsystem is out of the scope of this chapter.

An alternate way of configuring a security domain, instead of using annotation, is to use `jboss-ejb3.xml` deployment descriptor. Here's an example of how the configuration will look like:

```
<?xml version="1.0" encoding="UTF-8"?>
<jboss:jboss
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:jboss="http://www.jboss.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:s="urn:security:1.1"
  version="3.1" impl-version="2.0">

  <assembly-descriptor>
    <s:security>
<!-- Even wildcard * is supported -->
      <ejb-name>MyBean</ejb-name>
      <!-- Name of the security domain which is configured in the EJB3 subsystem -->
      <s:security-domain>other</s:security-domain>
    </s:security>
  </assembly-descriptor>
</jboss:jboss>
```

As you can see we use the `security-domain` element to configure the security domain.



i The `jboss-ejb3.xml` is expected to be placed in the `.jar/META-INF` folder of a `.jar` deployment or `.war/WEB-INF` folder of a `.war` deployment.

20.12.3 Absence of security domain configuration but presence of other security metadata

Let's consider the following example bean:

```
@Stateless
public class FooBean {

    @RolesAllowed("bar")
    public void doSomething() {
        ..
    }
    ...
}
```

As you can see the `doSomething` method is configured to be accessible for users with role "bar". However, the bean isn't configured for any specific security domain. Prior to WildFly 8, the absence of an explicitly configured security domain on the bean would leave the bean unsecured, which meant that even if the `doSomething` method was configured with `@RolesAllowed("bar")` anyone even without the "bar" role could invoke on the bean.

In WildFly 8, the presence of any security metadata (like `@RolesAllowed`, `@PermitAll`, `@DenyAll`, `@RunAs`, `@RunAsPrincipal`) on the bean or any business method of the bean, makes the bean secure, even in the absence of an explicitly configured security domain. In such cases, the security domain name is default to "other". Users can explicitly configure an security domain for the bean if they want to using either the annotation or deployment descriptor approach explained earlier.

20.12.4 Access to methods without explicit security metadata, on a secured bean

Consider this example bean:



```
@Stateless
public class FooBean {

    @RolesAllowed("bar")
    public void doSomething() {
        ..
    }

    public void helloWorld() {
        ...
    }
}
```

As you can see the `doSomething` method is marked for access for only users with role "bar". That enables security on the bean (with security domain defaulted to "other"). However, notice that the method `helloWorld` doesn't have any specific security configurations.

In WildFly 8, such methods which have no explicit security configurations, in a secured bean, will be treated similar to a method with `@DenyAll` configuration. What that means is, no one is allowed access to the `helloWorld` method. This behaviour can be controlled via the `jboss-ejb3.xml` deployment descriptor at a per bean level or a per deployment level as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<jboss:jboss
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:jboss="http://www.jboss.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:s="urn:security:1.1"
    version="3.1" impl-version="2.0">

    <assembly-descriptor>
        <s:security>
            <!-- Even wildcard * is supported where * is equivalent to all EJBs in the deployment -->
            <ejb-name>FooBean</ejb-name>

            <s:missing-method-permissions-deny-access>false</s:missing-method-permissions-deny-access>
        </s:security>
    </assembly-descriptor>
</jboss:jboss>
```

Notice the use of `<missing-method-permissions-deny-access>` element. The value for this element can either be true or false. If this element isn't configured then it is equivalent to a value of true i.e. no one is allowed access to methods, which have no explicit security configurations, on secured beans. Setting this to false allows access to such methods for all users i.e. the behaviour will be switched to be similar to `@PermitAll`.

This behaviour can also be configured at the EJB3 subsystem level so that it applies to all EJB3 deployments on the server, as follows:



```
<subsystem xmlns="urn:jboss:domain:ejb3:1.4">
...
    <default-missing-method-permissions-deny-access value="true"/>
...
</subsystem>
```

Again, the `default-missing-method-permissions-deny-access` element accepts either a true or false value. A value of true makes the behaviour similar to `@DenyAll` and a value of false makes it behave like `@PermitAll`



21 EJB invocations from a remote client using JNDI

This chapter explains how to invoke EJBs from a remote client by using the JNDI API to first lookup the bean proxy and then invoke on that proxy.

i After you have read this article, do remember to take a look at [Remote EJB invocations via JNDI - EJB client API or remote-naming project](#)

Before getting into the details, we would like the users to know that we have introduced a new EJB client API, which is a WildFly-specific API and allows invocation on remote EJBs. This client API isn't based on JNDI. So remote clients need not rely on JNDI API to invoke on EJBs. A separate document covering the EJB remote client API will be made available. For now, you can refer to the javadocs of the EJB client project at <http://docs.jboss.org/ejbclient/>. In this document, we'll just concentrate on the traditional JNDI based invocation on EJBs. So let's get started:

21.1 Deploying your EJBs on the server side:

⚠ Users who already have EJBs deployed on the server side can just skip to the next section.

As a first step, you'll have to deploy your application containing the EJBs on the Wildfly server. If you want those EJBs to be remotely invocable, then you'll have to expose at least one remote view for that bean. In this example, let's consider a simple Calculator stateless bean which exposes a RemoteCalculator remote business interface. We'll also have a simple stateful CounterBean which exposes a RemoteCounter remote business interface. Here's the code:

```
package org.jboss.as.quickstarts.ejb.remote.stateless;

/**
 * @author Jaikiran Pai
 */
public interface RemoteCalculator {

    int add(int a, int b);

    int subtract(int a, int b);
}
```



```
package org.jboss.as.quickstarts.ejb.remote.stateless;

import javax.ejb.Remote;
import javax.ejb.Stateless;

/**
 * @author Jaikiran Pai
 */
@Stateless
@Remote(RemoteCalculator.class)
public class CalculatorBean implements RemoteCalculator {

    @Override
    public int add(int a, int b) {
        return a + b;
    }

    @Override
    public int subtract(int a, int b) {
        return a - b;
    }
}
```

```
package org.jboss.as.quickstarts.ejb.remote.stateful;

/**
 * @author Jaikiran Pai
 */
public interface RemoteCounter {

    void increment();

    void decrement();

    int getCount();
}
```



```
package org.jboss.as.quickstarts.ejb.remote.stateful;

import javax.ejb.Remote;
import javax.ejb.Stateful;

/**
 * @author Jaikiran Pai
 */
@Stateful
@Remote(RemoteCounter.class)
public class CounterBean implements RemoteCounter {

    private int count = 0;

    @Override
    public void increment() {
        this.count++;
    }

    @Override
    public void decrement() {
        this.count--;
    }

    @Override
    public int getCount() {
        return this.count;
    }
}
```

Let's package this in a jar (how you package it in a jar is out of scope of this chapter) named "jboss-as-ejb-remote-app.jar" and deploy it to the server. Make sure that your deployment has been processed successfully and there aren't any errors.

21.2 Writing a remote client application for accessing and invoking the EJBs deployed on the server

The next step is to write an application which will invoke the EJBs that you deployed on the server. In WildFly, you can either choose to use the WildFly specific EJB client API to do the invocation or use JNDI to lookup a proxy for your bean and invoke on that returned proxy. In this chapter we will concentrate on the JNDI lookup and invocation and will leave the EJB client API for a separate chapter.

So let's take a look at what the client code looks like for looking up the JNDI proxy and invoking on it. Here's the entire client code which invokes on a stateless bean:

```
package org.jboss.as.quickstarts.ejb.remote.client;

import javax.naming.Context;
import javax.naming.InitialContext;
```



```
import javax.naming.NamingException;
import java.security.Security;
import java.util.Hashtable;

import org.jboss.as.quickstarts.ejb.remote.stateful.CounterBean;
import org.jboss.as.quickstarts.ejb.remote.stateful.RemoteCounter;
import org.jboss.as.quickstarts.ejb.remote.stateless.CalculatorBean;
import org.jboss.as.quickstarts.ejb.remote.stateless.RemoteCalculator;
import org.jboss.sasl.JBossSaslProvider;

/**
 * A sample program which acts a remote client for a EJB deployed on Wildfly 10 server.
 * This program shows how to lookup stateful and stateless beans via JNDI and then invoke on
them
 *
 * @author Jaikiran Pai
 */
public class RemoteEJBClient {

    public static void main(String[] args) throws Exception {
        // Invoke a stateless bean
        invokeStatelessBean();

        // Invoke a stateful bean
        invokeStatefulBean();
    }

    /**
     * Looks up a stateless bean and invokes on it
     *
     * @throws NamingException
     */
    private static void invokeStatelessBean() throws NamingException {
        // Let's lookup the remote stateless calculator
        final RemoteCalculator statelessRemoteCalculator = lookupRemoteStatelessCalculator();
        System.out.println("Obtained a remote stateless calculator for invocation");
        // invoke on the remote calculator
        int a = 204;
        int b = 340;
        System.out.println("Adding " + a + " and " + b + " via the remote stateless calculator
deployed on the server");
        int sum = statelessRemoteCalculator.add(a, b);
        System.out.println("Remote calculator returned sum = " + sum);
        if (sum != a + b) {
            throw new RuntimeException("Remote stateless calculator returned an incorrect sum "
+ sum + " ,expected sum was " + (a + b));
        }
        // try one more invocation, this time for subtraction
        int num1 = 3434;
        int num2 = 2332;
        System.out.println("Subtracting " + num2 + " from " + num1 + " via the remote stateless
calculator deployed on the server");
        int difference = statelessRemoteCalculator.subtract(num1, num2);
        System.out.println("Remote calculator returned difference = " + difference);
        if (difference != num1 - num2) {
            throw new RuntimeException("Remote stateless calculator returned an incorrect
difference " + difference + " ,expected difference was " + (num1 - num2));
        }
    }
}
```



```
}

/**
 * Looks up a stateful bean and invokes on it
 *
 * @throws NamingException
 */
private static void invokeStatefulBean() throws NamingException {
    // Let's lookup the remote stateful counter
    final RemoteCounter statefulRemoteCounter = lookupRemoteStatefulCounter();
    System.out.println("Obtained a remote stateful counter for invocation");
    // invoke on the remote counter bean
    final int NUM_TIMES = 20;
    System.out.println("Counter will now be incremented " + NUM_TIMES + " times");
    for (int i = 0; i < NUM_TIMES; i++) {
        System.out.println("Incrementing counter");
        statefulRemoteCounter.increment();
        System.out.println("Count after increment is " + statefulRemoteCounter.getCount());
    }
    // now decrementing
    System.out.println("Counter will now be decremented " + NUM_TIMES + " times");
    for (int i = NUM_TIMES; i > 0; i--) {
        System.out.println("Decrementing counter");
        statefulRemoteCounter.decrement();
        System.out.println("Count after decrement is " + statefulRemoteCounter.getCount());
    }
}

/**
 * Looks up and returns the proxy to remote stateless calculator bean
 *
 * @return
 * @throws NamingException
 */
private static RemoteCalculator lookupRemoteStatelessCalculator() throws NamingException {
    final Hashtable jndiProperties = new Hashtable();
    jndiProperties.put(Context.URL_PKG_PREFIXES, "org.jboss.ejb.client.naming");
    final Context context = new InitialContext(jndiProperties);
    // The app name is the application name of the deployed EJBs. This is typically the ear
name
    // without the .ear suffix. However, the application name could be overridden in the
application.xml of the
    // EJB deployment on the server.
    // Since we haven't deployed the application as a .ear, the app name for us will be an
empty string
    final String appName = "";
    // This is the module name of the deployed EJBs on the server. This is typically the jar
name of the
    // EJB deployment, without the .jar suffix, but can be overridden via the ejb-jar.xml
    // In this example, we have deployed the EJBs in a jboss-as-remote-app.jar, so the
module name is
    // jboss-as-remote-app
    final String moduleName = "jboss-as-remote-app";
    // AS7 allows each deployment to have an (optional) distinct name. We haven't specified
a distinct name for
    // our EJB deployment, so this is an empty string
    final String distinctName = "";
    // The EJB name which by default is the simple class name of the bean implementation
```



```
class
    final String beanName = CalculatorBean.class.getSimpleName();
    // the remote view fully qualified class name
    final String viewClassName = RemoteCalculator.class.getName();
    // let's do the lookup
    return (RemoteCalculator) context.lookup("ejb:" + appName + "/" + moduleName + "/" +
distinctName + "/" + beanName + "!" + viewClassName);
}

/**
 * Looks up and returns the proxy to remote stateful counter bean
 *
 * @return
 * @throws NamingException
 */
private static RemoteCounter lookupRemoteStatefulCounter() throws NamingException {
    final Hashtable jndiProperties = new Hashtable();
    jndiProperties.put(Context.URL_PKG_PREFIXES, "org.jboss.ejb.client.naming");
    final Context context = new InitialContext(jndiProperties);
    // The app name is the application name of the deployed EJBs. This is typically the ear
name
    // without the .ear suffix. However, the application name could be overridden in the
application.xml of the
    // EJB deployment on the server.
    // Since we haven't deployed the application as a .ear, the app name for us will be an
empty string
    final String appName = "";
    // This is the module name of the deployed EJBs on the server. This is typically the jar
name of the
    // EJB deployment, without the .jar suffix, but can be overridden via the ejb-jar.xml
    // In this example, we have deployed the EJBs in a jboss-as-ejb-remote-app.jar, so the
module name is
    // jboss-as-ejb-remote-app
    final String moduleName = "jboss-as-ejb-remote-app";
    // AS7 allows each deployment to have an (optional) distinct name. We haven't specified
a distinct name for
    // our EJB deployment, so this is an empty string
    final String distinctName = "";
    // The EJB name which by default is the simple class name of the bean implementation
class
    final String beanName = CounterBean.class.getSimpleName();
    // the remote view fully qualified class name
    final String viewClassName = RemoteCounter.class.getName();
    // let's do the lookup (notice the ?stateful string as the last part of the jndi name
for stateful bean lookup)
    return (RemoteCounter) context.lookup("ejb:" + appName + "/" + moduleName + "/" +
distinctName + "/" + beanName + "!" + viewClassName + "?stateful");
}
}
```



The entire server side and client side code is hosted at the github repo here [ejb-remote](#)



The code has some comments which will help you understand each of those lines. But we'll explain here in more detail what the code does. As a first step in the client code, we'll do a lookup of the EJB using a JNDI name. In AS7, for remote access to EJBs, you use the `ejb:` namespace with the following syntax:

For stateless beans:

```
ejb:<app-name>/<module-name>/<distinct-name>/<bean-name>!<fully-qualified-classname-of-the-remote-
```

For stateful beans:

```
ejb:<app-name>/<module-name>/<distinct-name>/<bean-name>!<fully-qualified-classname-of-the-remote-
```

The `ejb:` namespace identifies it as a EJB lookup and is a constant (i.e. doesn't change) for doing EJB lookups. The rest of the parts in the jndi name are as follows:

app-name : This is the name of the `.ear` (without the `.ear` suffix) that you have deployed on the server and contains your EJBs.

- Java EE 6 allows you to override the application name, to a name of your choice by setting it in the `application.xml`. If the deployment uses such an override then the `app-name` used in the JNDI name should match that name.
- EJBs can also be deployed in a `.war` or a plain `.jar` (like we did in step 1). In such cases where the deployment isn't an `.ear` file, then the `app-name` must be an empty string, while doing the lookup.

module-name : This is the name of the `.jar` (without the `.jar` suffix) that you have deployed on the server and the contains your EJBs. If the EJBs are deployed in a `.war` then the module name is the `.war` name (without the `.war` suffix).

- Java EE 6 allows you to override the module name, by setting it in the `ejb-jar.xml/web.xml` of your deployment. If the deployment uses such an override then the `module-name` used in the JNDI name should match that name.
- Module name part cannot be an empty string in the JNDI name

distinct-name : This is a WildFly-specific name which can be optionally assigned to the deployments that are deployed on the server. More about the purpose and usage of this will be explained in a separate chapter. If a deployment doesn't use `distinct-name` then, use an empty string in the JNDI name, for `distinct-name`

bean-name : This is the name of the bean for which you are doing the lookup. The bean name is typically the unqualified classname of the bean implementation class, but can be overridden through either `ejb-jar.xml` or via annotations. The bean name part cannot be an empty string in the JNDI name.

fully-qualified-classname-of-the-remote-interface : This is the fully qualified class name of the interface for which you are doing the lookup. The interface should be one of the remote interfaces exposed by the bean on the server. The fully qualified class name part cannot be an empty string in the JNDI name.



For stateful beans, the JNDI name expects an additional "?stateful" to be appended after the fully qualified interface name part. This is because for stateful beans, a new session gets created on JNDI lookup and the EJB client API implementation doesn't contact the server during the JNDI lookup to know what kind of a bean the JNDI name represents (we'll come to this in a while). So the JNDI name itself is expected to indicate that the client is looking up a stateful bean, so that an appropriate session can be created.

Now that we know the syntax, let's see our code and check what JNDI name it uses. Since our stateless EJB named CalculatorBean is deployed in a jboss-as-ejb-remote-app.jar (without any ear) and since we are looking up the org.jboss.as.quickstarts.ejb.remote.stateless.RemoteCalculator remote interface, our JNDI name will be:

```
ejb:/jboss-as-ejb-remote-app//CalculatorBean!org.jboss.as.quickstarts.ejb.remote.stateless.RemoteC
```

That's what the lookupRemoteStatelessCalculator() method in the above client code uses.

For the stateful EJB named CounterBean which is deployed in the same jboss-as-ejb-remote-app.jar and which exposes the org.jboss.as.quickstarts.ejb.remote.stateful.RemoteCounter, the JNDI name will be:

```
ejb:/jboss-as-ejb-remote-app//CounterBean!org.jboss.as.quickstarts.ejb.remote.stateful.RemoteCount
```

That's what the lookupRemoteStatefulCounter() method in the above client code uses.

Now that we know of the JNDI name, let's take a look at the following piece of code in the lookupRemoteStatelessCalculator():

```
final Hashtable jndiProperties = new Hashtable();
jndiProperties.put(Context.URL_PKG_PREFIXES, "org.jboss.ejb.client.naming");
final Context context = new InitialContext(jndiProperties);
```

Here we are creating a JNDI InitialContext object by passing it some JNDI properties. The Context.URL_PKG_PREFIXES is set to org.jboss.ejb.client.naming. This is necessary because we should let the JNDI API know what handles the ejb: namespace that we use in our JNDI names for lookup. The "org.jboss.ejb.client.naming" has a URLContextFactory implementation which will be used by the JNDI APIs to parse and return an object for ejb: namespace lookups. You can either pass these properties to the constructor of the InitialContext class or have a jndi.properties file in the classpath of the client application, which (atleast) contains the following property:

```
java.naming.factory.url.pkgs=org.jboss.ejb.client.naming
```

So at this point, we have setup the InitialContext and also have the JNDI name ready to do the lookup. You can now do the lookup and the appropriate proxy which will be castable to the remote interface that you used as the fully qualified class name in the JNDI name, will be returned. Some of you might be wondering, how the JNDI implementation knew which server address to look, for your deployed EJBs. The answer is in AS7, the proxies returned via JNDI name lookup for ejb: namespace do not connect to the server unless an invocation on those proxies is done.



Now let's get to the point where we invoke on this returned proxy:

```
// Let's lookup the remote stateless calculator
    final RemoteCalculator statelessRemoteCalculator = lookupRemoteStatelessCalculator();
    System.out.println("Obtained a remote stateless calculator for invocation");
    // invoke on the remote calculator
    int a = 204;
    int b = 340;
    System.out.println("Adding " + a + " and " + b + " via the remote stateless calculator
    deployed on the server");
    int sum = statelessRemoteCalculator.add(a, b);
```

We can see here that the proxy returned after the lookup is used to invoke the `add(...)` method of the bean. It's at this point that the JNDI implementation (which is backed by the EJB client API) needs to know the server details. So let's now get to the important part of setting up the EJB client context properties.

21.3 Setting up EJB client context properties

A EJB client context is a context which contains contextual information for carrying out remote invocations on EJBs. This is a WildFly-specific API. The EJB client context can be associated with multiple EJB receivers. Each EJB receiver is capable of handling invocations on different EJBs. For example, an EJB receiver "Foo" might be able to handle invocation on a bean identified by `app-A/module-A/distinctinctName-A/Bar!RemoteBar`, whereas a EJB receiver named "Blah" might be able to handle invocation on a bean identified by `app-B/module-B/distinctName-B/BeanB!RemoteBean`. Each such EJB receiver knows about what set of EJBs it can handle and each of the EJB receiver knows which server target to use for handling the invocations on the bean. For example, if you have a AS7 server at 10.20.30.40 IP address which has its remoting port opened at 4447 and if that's the server on which you deployed that `CalculatorBean`, then you can setup a EJB receiver which knows its target address is 10.20.30.40:4447. Such an EJB receiver will be capable enough to communicate to the server via the JBoss specific EJB remote client protocol (details of which will be explained in-depth in a separate chapter).

Now that we know what a EJB client context is and what a EJB receiver is, let's see how we can setup a client context with 1 EJB receiver which can connect to 10.20.30.40 IP address at port 4447. That EJB client context will then be used (internally) by the JNDI implementation to handle invocations on the bean proxy.

The client will have to place a `jboss-ejb-client.properties` file in the classpath of the application. The `jboss-ejb-client.properties` can contain the following properties:




```
endpoint.name=client-endpoint
remote.connectionprovider.create.options.org.xnio.Options.SSL_ENABLED=false


remote.connections=default

remote.connection.default.host=10.20.30.40
remote.connection.default.port = 8080
remote.connection.default.connect.options.org.xnio.Options.SASL_POLICY_NOANONYMOUS=false

remote.connection.default.username=appuser
remote.connection.default.password=apppassword
```

 This file includes a reference to a default password. Be sure to change this as soon as possible.

The above properties file is just an example. The actual file that was used for this sample program is available here for reference [jboss-ejb-client.properties](#)

 We'll see what each of it means.

First the `endpoint.name` property. We mentioned earlier that the EJB receivers will communicate with the server for EJB invocations. Internally, they use JBoss Remoting project to carry out the communication. The `endpoint.name` property represents the name that will be used to create the client side of the endpoint. The `endpoint.name` property is optional and if not specified in the `jboss-ejb-client.properties` file, it will default to "config-based-ejb-client-endpoint" name.

Next is the `remote.connectionprovider.create.options.<...>` properties:

```
remote.connectionprovider.create.options.org.xnio.Options.SSL_ENABLED=false
```

The "remote.connectionprovider.create.options." property prefix can be used to pass the options that will be used while create the connection provider which will handle the "remote:" protocol. In this example we use the "remote.connectionprovider.create.options." property prefix to pass the "org.xnio.Options.SSL_ENABLED" property value as false. That property will then be used during the connection provider creation. Similarly other properties can be passed too, just append it to the "remote.connectionprovider.create.options." prefix

Next we'll see:

```
remote.connections=default
```



This is where you define the connections that you want to setup for communication with the remote server. The "remote.connections" property uses a comma separated value of connection "names". The connection names are just logical and are used grouping together the connection configuration properties later on in the properties file. The example above sets up a single remote connection named "default". There can be more than one connections that are configured. For example:

```
remote.connections=one, two
```

Here we are listing 2 connections named "one" and "two". Ultimately, each of the connections will map to a EJB receiver. So if you have 2 connections, that will setup 2 EJB receivers that will be added to the EJB client context. Each of these connections will be configured with the connection specific properties as follows:

```
remote.connection.default.host=10.20.30.40
remote.connection.default.port = 8080
remote.connection.default.connect.options.org.xnio.Options.SASL_POLICY_NOANONYMOUS=false
```

As you can see we are using the "remote.connection.<connection-name>." prefix for specifying the connection specific property. The connection name here is "default" and we are setting the "host" property of that connection to point to 10.20.30.40. Similarly we set the "port" for that connection to 4447.

By default WildFly uses 8080 as the remoting port. The EJB client API uses the http port, with the http-upgrade functionality, for communicating with the server for remote invocations, so that's the port we use in our client programs (unless the server is configured for some other http port)




```
remote.connection.default.username=appuser
remote.connection.default.password=apppassword
```

The given user/password must be set by using the command bin/add-user.sh (or.bat). The user and password must be set because the security-realm is enabled for the subsystem remoting (see standalone*.xml or domain.xml) by default. If you do not need the security for remoting you might remove the attribute security-realm in the configuration.

security-realm is enabled by default.



 We then use the "remote.connection.<connection-name>.connect.options." property prefix to setup options that will be used during the connection creation.

Here's an example of setting up multiple connections with different properties for each of those:

```
remote.connectionprovider.create.options.org.xnio.Options.SSL_ENABLED=false

remote.connections=one, two

remote.connection.one.host=localhost
remote.connection.one.port=6999
remote.connection.one.connect.options.org.xnio.Options.SASL_POLICY_NOANONYMOUS=false

remote.connection.two.host=localhost
remote.connection.two.port=7999
remote.connection.two.connect.options.org.xnio.Options.SASL_POLICY_NOANONYMOUS=false
```

As you can see we setup 2 connections "one" and "two" which both point to "localhost" as the "host" but different ports. Each of these connections will internally be used to create the EJB receivers in the EJB client context.

So that's how the `jboss-ejb-client.properties` file can be setup and placed in the classpath.

Using a different file for setting up EJB client context

The EJB client code will by default look for `jboss-ejb-client.properties` in the classpath. However, you can specify a different file of your choice by setting the "jboss.ejb.client.properties.file.path" system property which points to a properties file on your filesystem, containing the client context configurations. An example for that would be

```
"-Djboss.ejb.client.properties.file.path=/home/me/my-client/custom-jboss-ejb-client.properties"
```

Setting up the client classpath with the jars that are required to run the client application

A `jboss-client` jar is shipped in the distribution. It's available at `WILDFLY_HOME/bin/client/jboss-client.jar`. Place this jar in the classpath of your client application.

If you are using Maven to build the client application, then please follow the instructions in the `WILDFLY_HOME/bin/client/README.txt` to add this jar as a Maven dependency.





21.4 Summary

In the above examples, we saw what it takes to invoke a EJB from a remote client. To summarize:


- On the server side you need to deploy EJBs which expose the remote views.
- On the client side you need a client program which:
 - Has a `jboss-ejb-client.properties` in its classpath to setup the server connection information
 - Either has a `jndi.properties` to specify the `java.naming.factory.url.pkgs` property or passes that as a property to the `InitialContext` constructor
 - Setup the client classpath to include the `jboss-client` jar that's required for remote invocation of the EJBs. The location of the jar is mentioned above. You'll also need to have your application's bean interface jars and other jars that are required by your application, in the client classpath



22 EJB invocations from a remote server instance

The purpose of this chapter is to demonstrate how to lookup and invoke on EJBs deployed on an WildFly server instance **from another** WildFly server instance. This is different from invoking the EJBs [from a remote standalone client](#)


Let's call the server, from which the invocation happens to the EJB, as "Client Server" and the server on which the bean is deployed as the "Destination Server".

 Note that this chapter deals with the case where the bean is deployed on the "Destination Server" but **not** on the "Client Server".

22.1 Application packaging

In this example, we'll consider a EJB which is packaged in a `myejb.jar` which is within a `myapp.ear`. Here's how it would look like:

```
myapp.ear
|
|---- myejb.jar
|      |
|      |---- <org.myapp.ejb.*> // EJB classes
```

 Note that packaging itself isn't really important in the context of this article. You can deploy the EJBs in any standard way (`.ear`, `.war` or `.jar`).



22.2 Beans

In our example, we'll consider a simple stateless session bean which is as follows:

```
package org.myapp.ejb;

public interface Greeter {

    String greet(String user);

}
```

```
package org.myapp.ejb;

import javax.ejb.Remote;
import javax.ejb.Stateless;

@Stateless
@Remote (Greeter.class)
public class GreeterBean implements Greeter {

    @Override
    public String greet(String user) {
        return "Hello " + user + ", have a pleasant day!";
    }

}
```

22.3 Security

WildFly 8 is secure by default. What this means is that no communication can happen with an WildFly instance from a remote client (irrespective of whether it is a standalone client or another server instance) without passing the appropriate credentials. Remember that in this example, our "client server" will be communicating with the "destination server". So in order to allow this communication to happen successfully, we'll have to configure user credentials which we will be using during this communication. So let's start with the necessary configurations for this.



22.4 Configuring a user on the "Destination Server"

As a first step we'll configure a user on the destination server who will be allowed to access the destination server. We create the user using the `add-user` script that's available in the `JBOSS_HOME/bin` folder. In this example, we'll be configuring a `Application` User named `ejb` and with a password `test` in the `ApplicationRealm`. Running the `add-user` script is an interactive process and you will see questions/output as follows:


add-user


```
jpai@jpai-laptop:bin$ ./add-user.sh

What type of user do you wish to add?
  a) Management User (mgmt-users.properties)
  b) Application User (application-users.properties)
(a): b

Enter the details of the new user to add.
Realm (ApplicationRealm) :
Username : ejb
Password :
Re-enter Password :
What roles do you want this user to belong to? (Please enter a comma separated list, or leave
blank for none)\[\&nbsp; \]:
About to add user 'ejb' for realm 'ApplicationRealm'
Is this correct yes/no? yes
Added user 'ejb' to file
'/jboss-as-7.1.1.Final/standalone/configuration/application-users.properties'
Added user 'ejb' to file
'/jboss-as-7.1.1.Final/domain/configuration/application-users.properties'
Added user 'ejb' with roles to file
'/jboss-as-7.1.1.Final/standalone/configuration/application-roles.properties'
Added user 'ejb' with roles to file
'/jboss-as-7.1.1.Final/domain/configuration/application-roles.properties'
```

As you can see in the output above we have now configured a user on the destination server who'll be allowed to access this server. We'll use this user credentials later on in the client server for communicating with this server. The important bits to remember are the user we have created in this example is `ejb` and the password is `test`.

 Note that you can use any username and password combination you want to.

 You do **not** require the server to be started to add a user using the `add-user` script.



22.5 Start the "Destination Server"

As a next step towards running this example, we'll start the "Destination Server". In this example, we'll use the standalone server and use the *standalone-full.xml* configuration. The startup command will look like:

```
./standalone.sh -server-config=standalone-full.xml
```

Ensure that the server has started without any errors.



It's very important to note that if you are starting both the server instances on the same machine, then each of those server instances **must** have a unique `jboss.node.name` system property. You can do that by passing an appropriate value for `-Djboss.node.name` system property to the startup script:

```
./standalone.sh -server-config=standalone-full.xml -Djboss.node.name=<add appropriate value here>
```

22.6 Deploying the application

The application (*myapp.ear* in our case) will be deployed to "Destination Server". The process of deploying the application is out of scope of this chapter. You can either use the Command Line Interface or the Admin console or any IDE or manually copy it to `JBOSS_HOME/standalone/deployments` folder (for standalone server). Just ensure that the application has been deployed successfully.

So far, we have built a EJB application and deployed it on the "Destination Server". Now let's move to the "Client Server" which acts as the client for the deployed EJBs on the "Destination Server".

22.7 Configuring the "Client Server" to point to the EJB remoting connector on the "Destination Server"

As a first step on the "Client Server", we need to let the server know about the "Destination Server"'s EJB remoting connector, over which it can communicate during the EJB invocations. To do that, we'll have to add a "*remote-outbound-connection*" to the remoting subsystem on the "Client Server". The "*remote-outbound-connection*" configuration indicates that a outbound connection will be created to a remote server instance from that server. The "*remote-outbound-connection*" will be backed by a "*outbound-socket-binding*" which will point to a remote host and a remote port (of the "Destination Server"). So let's see how we create these configurations.



22.8 Start the "Client Server"

In this example, we'll start the "Client Server" on the same machine as the "Destination Server". We have copied the entire server installation to a different folder and while starting the "Client Server" we'll use a port-offset (of 100 in this example) to avoid port conflicts:

```
./standalone.sh -server-config=standalone-full.xml -Djboss.socket.binding.port-offset=100
```

22.9 Create a security realm on the client server

Remember that we need to communicate with a secure destination server. In order to do that the client server has to pass the user credentials to the destination server. Earlier we created a user on the destination server who'll be allowed to communicate with that server. Now on the "client server" we'll create a security-realm which will be used to pass the user information.

In this example we'll use a security realm which stores a Base64 encoded password and then passes on that credentials when asked for. Earlier we created a user named `ejb` and password `test`. So our first task here would be to create the base64 encoded version of the password `test`. You can use any utility which generates you a base64 version for a string. I used [this online site](#) which generates the base64 encoded string. So for the `test` password, the base64 encoded version is `dGVzdA==`

- ✔ While generating the base64 encoded string make sure that you don't have any trailing or leading spaces for the original password. That can lead to incorrect encoded versions being generated.

- ✔ With new versions the add-user script will show the base64 password if you type 'y' if you've been ask


```
Is this new user going to be used for one AS process to connect to another AS process  
e.g. slave domain controller?
```

Now that we have generated that base64 encoded password, let's use in the in the security realm that we are going to configure on the "client server". I'll first shutdown the client server and edit the `standalone-full.xml` file to add the following in the `<management>` section

Now let's create a "*security-realm*" for the base64 encoded password.

```
/core-service=management/security-realm=ejb-security-realm:add()  
/core-service=management/security-realm=ejb-security-realm/server-identity=secret:add(value=dGVzdA==)
```



 Notice that the CLI show the message "*process-state*" => "*reload-required*", so you have to restart the server before you can use this change.

upon successful invocation of this command, the following configuration will be created in the *management* section:

standalone-full.xml

```
<management>
  <security-realms>
    ...
    <security-realm name="ejb-security-realm">
      <server-identities>
        <secret value="dGVzdA==" />
      </server-identities>
    </security-realm>
  </security-realms>
  ...

```

As you can see I have created a security realm named "ejb-security-realm" (you can name it anything) with the base64 encoded password. So that completes the security realm configuration for the client server. Now let's move on to the next step.



22.10 Create a outbound-socket-binding on the "Client Server"

Let's first create a *outbound-socket-binding* which points the "Destination Server"'s host and port. We'll use the CLI to create this configuration:

```
/socket-binding-group=standard-sockets/remote-destination-outbound-socket-binding=remote-ejb:add(host=localhost, port=8080)
```

The above command will create a outbound-socket-binding named "*remote-ejb*" (we can name it anything) which points to "localhost" as the host and port 8080 as the destination port. Note that the host information should match the host/IP of the "Destination Server" (in this example we are running on the same machine so we use "localhost") and the port information should match the http-remoting connector port used by the EJB subsystem (by default it's 8080). When this command is run successfully, we'll see that the standalone-full.xml (the file which we used to start the server) was updated with the following outbound-socket-binding in the socket-binding-group:

```
<socket-binding-group name="standard-sockets" default-interface="public"
port-offset="{jboss.socket.binding.port-offset:0}">
  ...
  <outbound-socket-binding name="remote-ejb">
    <remote-destination host="localhost" port="8080"/>
  </outbound-socket-binding>
</socket-binding-group>
```

22.11 Create a "remote-outbound-connection" which uses this newly created "outbound-socket-binding"

Now let's create a "*remote-outbound-connection*" which will use the newly created outbound-socket-binding (pointing to the EJB remoting connector of the "Destination Server"). We'll continue to use the CLI to create this configuration:

```
/subsystem=remoting/remote-outbound-connection=remote-ejb-connection:add(outbound-socket-binding-ref=remote-ejb, protocol=http-remoting, security-realm=ejb-security-realm, username=ejb)
```

The above command creates a remote-outbound-connection, named "*remote-ejb-connection*" (we can name it anything), in the remoting subsystem and uses the previously created "*remote-ejb*" outbound-socket-binding (notice the outbound-socket-binding-ref in that command) with the http-remoting protocol. Furthermore, we also set the security-realm attribute to point to the security-realm that we created in the previous step. Also notice that we have set the username attribute to use the user name who is allowed to communicate with the destination server.



What this step does is, it creates a outbound connection, on the client server, to the remote destination server and sets up the username to the user who allowed to communicate with that destination server and also sets up the security-realm to a pre-configured security-realm capable of passing along the user credentials (in this case the password). This way when a connection has to be established from the client server to the destination server, the connection creation logic will have the necessary security credentials to pass along and setup a successful secured connection.

Now let's run the following two operations to set some default connection creation options for the outbound connection:

```
/subsystem=remoting/remote-outbound-connection=remote-ejb-connection/property=SASL_POLICY_NOANONYMOUS
```

```
/subsystem=remoting/remote-outbound-connection=remote-ejb-connection/property=SSL_ENABLED:add(value=false)
```

Ultimately, upon successful invocation of this command, the following configuration will be created in the remoting subsystem:

```
<subsystem xmlns="urn:jboss:domain:remoting:1.1">
  ....
  <outbound-connections>
    <remote-outbound-connection name="remote-ejb-connection"
outbound-socket-binding-ref="remote-ejb" protocol="http-remoting"
security-realm="ejb-security-realm" username="ejb">
      <properties>
        <property name="SASL_POLICY_NOANONYMOUS" value="false"/>
        <property name="SSL_ENABLED" value="false"/>
      </properties>
    </remote-outbound-connection>
  </outbound-connections>
</subsystem>
```

From a server configuration point of view, that's all we need on the "Client Server". Our next step is to deploy an application on the "Client Server" which will invoke on the bean deployed on the "Destination Server".




22.12 Packaging the client application on the "Client Server"

Like on the "Destination Server", we'll use .ear packaging for the client application too. But like previously mentioned, that's not mandatory. You can even use a .war or .jar deployments. Here's how our client application packaging will look like:

```
client-app.ear
|
|--- META-INF
|       |
|       |--- jboss-ejb-client.xml
|
|--- web.war
|       |
|       |--- WEB-INF/classes
|               |
|               |---- <org.myapp.FooServlet> // classes in the web app
```

In the client application we'll use a servlet which invokes on the bean deployed on the "Destination Server". We can even invoke the bean on the "Destination Server" from a EJB on the "Client Server". The code remains the same (JNDI lookup, followed by invocation on the proxy). The important part to notice in this client application is the file *jboss-ejb-client.xml* which is packaged in the META-INF folder of a top level deployment (in this case our client-app.ear). This *jboss-ejb-client.xml* contains the EJB client configurations which will be used during the EJB invocations for finding the appropriate destinations (also known as, EJB receivers). The contents of the *jboss-ejb-client.xml* are explained next.

 If your application is deployed as a top level .war deployment, then the *jboss-ejb-client.xml* is expected to be placed in .war/WEB-INF/ folder (i.e. the same location where you place any web.xml file).



22.13 Contents on jboss-ejb-client.xml

The jboss-ejb-client.xml will look like:

```
<jboss-ejb-client xmlns="urn:jboss:ejb-client:1.0">
  <client-context>
    <ejb-receivers>
      <remoting-ejb-receiver outbound-connection-ref="remote-ejb-connection"/>
    </ejb-receivers>
  </client-context>
</jboss-ejb-client>
```

You'll notice that we have configured the EJB client context (for this application) to use a `remoting-ejb-receiver` which points to our earlier created "`remote-outbound-connection`" named "`remote-ejb-connection`". This links the EJB client context to use the "`remote-ejb-connection`" which ultimately points to the EJB remoting connector on the "Destination Server".

22.14 Deploy the client application

Let's deploy the client application on the "Client Server". The process of deploying the application is out of scope, of this chapter. You can use either the CLI or the admin console or a IDE or deploy manually to `JBOSS_HOME/standalone/deployments` folder. Just ensure that the application is deployed successfully.



22.15 Client code invoking the bean

We mentioned that we'll be using a servlet to invoke on the bean, but the code to invoke the bean isn't servlet specific and can be used in other components (like EJB) too. So let's see how it looks like:

```
import javax.naming.Context;
import java.util.Hashtable;
import javax.naming.InitialContext;

...
public void invokeOnBean() {
    try {
        final Hashtable props = new Hashtable();
        // setup the ejb: namespace URL factory
        props.put(Context.URL_PKG_PREFIXES, "org.jboss.ejb.client.naming");
        // create the InitialContext
        final Context context = new javax.naming.InitialContext(props);

        // Lookup the Greeter bean using the ejb: namespace syntax which is explained here
https://docs.jboss.org/author/display/AS71/EJB+invocations+from+a+remote+client+using+JNDI
        final Greeter bean = (Greeter) context.lookup("ejb:" + "myapp" + "/" + "myejb" + "/"
+ "" + "/" + "GreeterBean" + "!" + org.myapp.ejb.Greeter.class.getName());

        // invoke on the bean
        final String greeting = bean.greet("Tom");

        System.out.println("Received greeting: " + greeting);

    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

That's it! The above code will invoke on the bean deployed on the "Destination Server" and return the result.



23 Example Applications - Migrated to WildFly

23.1 Example Applications Migrated from Previous Releases

The applications in this section were written for a previous version of the server but have been modified to run on WildFly 8. Changes were made to resolve issues that arose during deployment and runtime or to fix problems with application behaviour. Each example below documents the changes that were made to get the application to run successfully on WildFly.

23.1.1 Seam 2 JPA example

To see details on changes required to run this application on WildFly, see [Seam 2 JPA example deployment on WildFly 8](#).

23.1.2 Seam 2 DVD Store example

For details on how to migrate this demo application, see [Seam 2 DVD Store example on WildFly 8](#) on Marek Novotny's Blog.

23.1.3 Seam 2 Booking example

For details on how to migrate this demo application, see [Seam 2 Booking example on WildFly 8](#) on Marek Novotny's Blog.

23.1.4 Seam 2 Booking - step-by-step migration of binaries

This document takes a somewhat different "brute force" approach. The idea is to deploy the binaries to WildFly, then see what issues you hit and learn how debug and resolve them. See [Seam 2 Booking EAR Migration of Binaries - Step by Step](#).

23.1.5 jBPM-Console application

Kris Verlaenen migrated this application from AS 5 to WildFly 8. For details about this migration, see [jBPM5 on WildFly](#) on his Kris's Blog.



23.1.6 Order application used for performance testing

Andy Miller migrated this application from AS 5 to WildFly. For details about this migration, see [Order Application Migration from EAP5.1 to WildFly](#).

23.1.7 Migrate example application

A step by step work through of issues, and their solutions, that might crop up when migrating applications to WildFly 8. See the following [github project](#) for details.

23.2 Example Applications Based on EE6

Applications in this section were designed and written specifically to use the features and functions of EE6.

- Quickstarts: A number of quickstart applications were written to demonstrate Java EE 6 and a few additional technologies. They provide small, specific, working examples that can be used as a reference for your own project. For more information about the quickstarts, see [Get Started Developing Applications](#)

23.3 Porting the Order Application from EAP 5.1 to WildFly 8

Andy Miller ported an example Order application that was used for performance testing from EAP 5.1 to WildFly 8. These are the notes he made during the migration process.

23.3.1 Overview of the application

The application is relatively simple. it contains three servlets, some stateless session beans, a stateful session bean, and some entities.

In addition to application code changes, modifications were made to the way the EAR was packaged. This is because WildFly removed support of some proprietary features that were available in EAP 5.1.



23.3.2 Summary of changes

Code Changes

Modify JNDI lookup code

Since this application was first written for EAP 4.2/4.3, which did not support EJB reference injection, the servlets were using pre-EE 5 methods for looking up stateless and stateful session bean interfaces. While migrating to WildFly, it seemed a good time to change the code to use the `@EJB` annotation, although this was not a required change.

The real difference is in the lookup name. WildFly only supports the new EE 6 portable JNDI names rather than the old EAR structure based names. The JNDI lookup code changed as follows:

Example of code in the EAP 5.1 version:

```
try {
    context = new InitialContext();
    distributionCenterManager = (DistributionCenterManager)
context.lookup("OrderManagerApp/DistributionCenterManagerBean/local");
} catch(Exception lookupError) {
    throw new ServletException("Couldn't find DistributionCenterManager bean", lookupError);
}
try {
    customerManager = (CustomerManager)
context.lookup("OrderManagerApp/CustomerManagerBean/local");
} catch(Exception lookupError) {
    throw new ServletException("Couldn't find CustomerManager bean", lookupError);
}

try {
    productManager = (ProductManager)
context.lookup("OrderManagerApp/ProductManagerBean/local");
} catch(Exception lookupError) {
    throw new ServletException("Couldn't find the ProductManager bean", lookupError);
}
```

Example of how this is now coded in WildFly:

```
@EJB(lookup="java:app/OrderManagerEJB/DistributionCenterManagerBean!services.ejb.DistributionCenterManager")
DistributionCenterManager distributionCenterManager;

@EJB(lookup="java:app/OrderManagerEJB/CustomerManagerBean!services.ejb.CustomerManager")
private CustomerManager customerManager;

@EJB(lookup="java:app/OrderManagerEJB/ProductManagerBean!services.ejb.ProductManager")
private ProductManager productManager;
```

In addition to the change to injection, which was supported in EAP 5.1.0, the lookup name changed from:



```
OrderManagerApp/DistributionCenterManagerBean/local
```

to:

```
java:app/OrderManagerEJB/DistributionCenterManagerBean!services.ejb.DistributionCenterManager
```

All the other beans were changed in a similar manner. They are now based on the portable JNDI names described in EE 6.



Modify logging code

The next major change was to logging within the application. The old version was using the commons logging infrastructure and Log4J that is bundled in the application server. Rather than bundling third-party logging, the application was modified to use the new WildFly Logging infrastructure.

The code changes themselves are rather trivial, as this example illustrates:

Old JBoss Commons Logging/Log4J:

```
private static Log log = LogFactory.getLog(CustomerManagerBean.class);
```

New WildFly Logging

```
private static Logger logger = Logger.getLogger(CustomerManagerBean.class.toString());
```

Old JBoss Commons Logging/Log4J:

```
if(log.isTraceEnabled()) {  
    log.trace("Just flushed " + batchSize + " rows to the database.");  
    log.trace("Total rows flushed is " + (i+1));  
}
```

New WildFly Logging:

```
if(logger.isLoggable(Level.TRACE)) {  
    logger.log(Level.TRACE, "Just flushed " + batchSize + " rows to the database.");  
    logger.log(Level.TRACE, "Total rows flushed is " + (i+1));  
}
```

In addition to the code changes made to use the new AS7 JBoss log manager module, you must add this dependency to the `MANIFEST.MF` file as follows:

```
Manifest-Version: 1.0  
Dependencies: org.jboss.logmanager
```



Modify the code to use Infinispan for 2nd level cache

Jboss Cache has been replaced by Infinispan for 2nd level cache. This requires modification of the `persistence.xml` file.

This is what the file looked like in EAP 5.1:

```
<properties>
<property name="hibernate.cache.region.factory_class"
value="org.hibernate.cache.jbc2.JndiMultiplexedJBossCacheRegionFactory"/>
<property name="hibernate.cache.region.jbc2.cachefactory" value="java:CacheManager"/>
<property name="hibernate.cache.use_second_level_cache" value="true"/>
<property name="hibernate.cache.use_query_cache" value="false"/>
<property name="hibernate.cache.use_minimal_puts" value="true"/>
<property name="hibernate.cache.region.jbc2.cfg.entity" value="mvcc-entity"/>
<property name="hibernate.cache.region_prefix" value="services"/>
</properties>
```

This is how it was modified to use Infinispan for the same configuration:

```
<properties>
<property name="hibernate.cache.use_second_level_cache" value="true"/>
<property name="hibernate.cache.use_minimal_puts" value="true"/>
</properties>
<shared-cache-mode>ENABLE_SELECTIVE</shared-cache-mode>
```

Most of the properties are removed since they will default to the correct values for the second level cache. See ["Using the Infinispan second level cache"](#) for more details.

That was the extent of the code changes required to migrate the application to AS7.



EAR Packaging Changes

Due to modular class loading changes, the structure of the existing EAR failed to deploy successfully in WildFly.

The old structure of the EAR was as follows:

```
$ jar tf OrderManagerApp.ear
META-INF/MANIFEST.MF
META-INF/application.xml
OrderManagerWeb.war
OrderManagerEntities.jar
OrderManagerEJB.jar
META-INF/
```

In this structure, the entities and the `persistence.xml` were in one jar file, `OrderManagerEntities.jar`, and the stateless and stateful session beans were in another jar file, `OrderManagerEJB.jar`. This did not work due to modular class loading changes in WildFly. There are a couple of ways to resolve this issue:

1. Modify the class path in the `MANIFEST.MF`
2. Flatten the code and put all the beans in one JAR file.

The second approach was selected because it simplified the EAR structure:

```
$ jar tf OrderManagerApp.ear
META-INF/application.xml
OrderManagerWeb.war
OrderManagerEJB.jar
META-INF/
```

Since there is no longer an `OrderManagerEntities.jar` file, the `application.xml` file was modified to remove the entry.

An entry was added to the `MANIFEST.MF` file in the `OrderManagerWeb.war` to resolve another class loading issue resulting from the modification to use EJB reference injection in the servlets.

```
Manifest-Version: 1.0
Dependencies: org.jboss.logmanager
Class-Path: OrderManagerEJB.jar
```

The `Class-Path` entry tells the application to look in the `OrderManagerEJB.jar` file for the injected beans.



Summary

Although the existing EAR structure could have worked with additional modifications to the `MANIFEST.MF` file, this approach seemed more appealing because it simplified the structure while maintaining the web tier in its own WAR.

The source files for both versions is attached so you can view the changes that were made to the application.

23.4 Seam 2 Booking Application - Migration of Binaries from EAP5.1 to WildFly

This is a step-by-step how-to guide on porting the Seam Booking application binaries from EAP5.1 to WildFly 8. Although there are better approaches for migrating applications, the purpose of this document is to show the types of issues you might encounter when migrating an application and how to debug and resolve those issues.

For this example, the application EAR is deployed to the `JBOSS_HOME/standalone/deployments` directory with no changes other than extracting the archives so we can modify the XML files contained within them.



23.4.1 Step 1: Build and deploy the EAP5.1 version of the Seam Booking application

1. Build the EAR

```
cd /EAP5_HOME/jboss-eap5.1/seam/examples/booking
~/tools/apache-ant-1.8.2/bin/ant explode
```

2. Copy the EAR to the JBOSS_HOME deployments directory:

```
cp -r
EAP5_HOME/jboss-eap-5.1/seam/examples/booking/exploded-archives/jboss-seam-booking.ear
AS7_HOME/standalone/deployments/
cp -r
EAP5_HOME/jboss-eap-5.1/seam/examples/booking/exploded-archives/jboss-seam-booking.war
AS7_HOME/standalone/deployments/jboss-seam.ear
cp -r
EAP5_HOME/jboss-eap-5.1/seam/examples/booking/exploded-archives/jboss-seam-booking.jar
AS7_HOME/standalone/deployments/jboss-seam.ear
```

3. Start the WildFly server and check the log. You will see:

```
INFO [org.jboss.as.deployment] (DeploymentScanner-threads - 1) Found
jboss-seam-booking.ear in deployment directory. To trigger deployment create a file called
jboss-seam-booking.ear.dodeploy
```

4. Create an empty file with the name `jboss-seam-booking.ear.dodeploy` and copy it into the deployments directory. In the log, you will now see the following, indicating that it is deploying:

```
INFO [org.jboss.as.server.deployment] (MSC service thread 1-1) Starting deployment of
"jboss-seam-booking.ear"
INFO [org.jboss.as.server.deployment] (MSC service thread 1-3) Starting deployment of
"jboss-seam-booking.jar"
INFO [org.jboss.as.server.deployment] (MSC service thread 1-6) Starting deployment of
"jboss-seam.jar"
INFO [org.jboss.as.server.deployment] (MSC service thread 1-2) Starting deployment of
"jboss-seam-booking.war"
```

At this point, you will first encounter your first deployment error. In the next section, we will step through each issue and how to debug and resolve it.



23.4.2 Step 2: Debug and resolve deployment errors and exceptions

First Issue: `java.lang.ClassNotFoundException: javax.faces.FacesException`

When you deploy the application, the log contains the following error:

```
ERROR \[org.jboss.msc.service.fail\] (MSC service thread 1-1) MSC00001: Failed to start service
jboss.deployment.subunit."jboss-seam-booking.ear"."jboss-seam-booking.war".POST_MODULE:
org.jboss.msc.service.StartException in service
jboss.deployment.subunit."jboss-seam-booking.ear"."jboss-seam-booking.war".POST_MODULE:
Failed to process phase POST_MODULE of subdeployment "jboss-seam-booking.war" of deployment
"jboss-seam-booking.ear"
    (... additional logs removed ...)
Caused by: java.lang.ClassNotFoundException: javax.faces.FacesException from \[Module
"deployment.jboss-seam-booking.ear:main" from Service Module Loader\]
    at org.jboss.modules.ModuleClassLoader.findClass(ModuleClassLoader.java:191)
```

What it means:

The `ClassNotFoundException` indicates a missing dependency. In this case, it can not find the class `javax.faces.FacesException` and you need to explicitly add the dependency.



How to resolve it:

Find the module name for that class in the `AS7_HOME/modules` directory by looking for a path that matches the missing class. In this case, you will find 2 modules that match:

```
javax/faces/api/main
javax/faces/api/1.2
```

Both modules have the same module name: "javax.faces.api" but one in the main directory is for JSF 2.0 and the one located in the 1.2 directory is for JSF 1.2. If there was only one module available, we could simply create a `MANIFEST.MF` file and added the module dependency. But in this case, we want to use the JSF 1.2 version and not the 2.0 version in main, so we need to be able to specify one and exclude the other. To do this, we create a `jboss-deployment-structure.xml` file in the EAR `META-INF/` directory that contains the following data:

```
<jboss-deployment-structure xmlns="urn:jboss:deployment-structure:1.0">
  <deployment>
    <dependencies>
      <module name="javax.faces.api" slot="1.2" export="true"/>
    </dependencies>
  </deployment>
  <sub-deployment name="jboss-seam-booking.war">
    <exclusions>
      <module name="javax.faces.api" slot="main"/>
    </exclusions>
    <dependencies>
      <module name="javax.faces.api" slot="1.2"/>
    </dependencies>
  </sub-deployment>
</jboss-deployment-structure>
```

In the "deployment" section, we add the dependency for the `javax.faces.api` for the JSF 1.2 module. We also add the dependency for the JSF 1.2 module in the sub-deployment section for the WAR and exclude the module for JSF 2.0.

Redeploy the application by deleting the `standalone/deployments/jboss-seam-booking.ear.failed` file and creating a blank `jboss-seam-booking.ear.dodeploy` file in the same directory.

Next Issue: `java.lang.ClassNotFoundException: org.apache.commons.logging.Log`

When you deploy the application, the log contains the following error:



```
ERROR [org.jboss.msc.service.fail] (MSC service thread 1-8) MSC00001: Failed to start service
jboss.deployment.unit."jboss-seam-booking.ear".INSTALL:
org.jboss.msc.service.StartException in service
jboss.deployment.unit."jboss-seam-booking.ear".INSTALL:
Failed to process phase INSTALL of deployment "jboss-seam-booking.ear"
(.. additional logs removed ...)
Caused by: java.lang.ClassNotFoundException: org.apache.commons.logging.Log from [Module
"deployment.jboss-seam-booking.ear.jboss-seam-booking.war:main" from Service Module Loader]
```

What it means:

The `ClassNotFoundException` indicates a missing dependency. In this case, it can not find the class `org.apache.commons.logging.Log` and you need to explicitly add the dependency.

How to resolve it:

Find the module name for that class in the `JBOSS_HOME/modules/` directory by looking for a path that matches the missing class. In this case, you will find one module that matches the path `org/apache/commons/logging/`. The module name is `"org.apache.commons.logging"`.

Modify the `jboss-deployment-structure.xml` to add the module dependency to the deployment section of the file.

```
<module name="org.apache.commons.logging" export="true"/>
```

The `jboss-deployment-structure.xml` should now look like this:

```
<jboss-deployment-structure xmlns="urn:jboss:deployment-structure:1.0">
  <deployment>
    <dependencies>
      <module name="javax.faces.api" slot="1.2" export="true"/>
      <module name="org.apache.commons.logging" export="true"/>
    </dependencies>
  </deployment>
  <sub-deployment name="jboss-seam-booking.war">
    <exclusions>
      <module name="javax.faces.api" slot="main"/>
    </exclusions>
    <dependencies>
      <module name="javax.faces.api" slot="1.2"/>
    </dependencies>
  </sub-deployment>
</jboss-deployment-structure>
```

Redeploy the application by deleting the

`standalone/deployments/jboss-seam-booking.ear.failed` file and creating a blank `jboss-seam-booking.ear.dodeploy` file in the same directory.



Next Issue: `java.lang.ClassNotFoundException`: `org.dom4j.DocumentException`

When you deploy the application, the log contains the following error:

```
ERROR [org.apache.catalina.core.ContainerBase.[jboss.web].[default-host].[/seam-booking]] (MSC
service thread 1-3) Exception sending context initialized event to listener instance of class
org.jboss.seam.servlet.SeamListener: java.lang.NoClassDefFoundError: org/dom4j/DocumentException
(... additional logs removed ...)
Caused by: java.lang.ClassNotFoundException: org.dom4j.DocumentException from [Module
"deployment.jboss-seam-booking.ear.jboss-seam.jar:main" from Service Module Loader]
```

What it means:

Again, the `ClassNotFoundException` indicates a missing dependency. In this case, it can not find the class `org.dom4j.DocumentException`.



How to resolve it:

Find the module name in the `JBOSS_HOME/modules/` directory by looking for the `org/dom4j/DocumentException`. The module name is “org.dom4j”.

Modify the `jboss-deployment-structure.xml` to add the module dependency to the deployment section of the file.

```
<module name="org.dom4j" export="true"/>
```

The `jboss-deployment-structure.xml` file should now look like this:

```
<jboss-deployment-structure xmlns="urn:jboss:deployment-structure:1.0">
  <deployment>
    <dependencies>
      <module name="javax.faces.api" slot="1.2" export="true"/>
      <module name="org.apache.commons.logging" export="true"/>
      <module name="org.dom4j" export="true"/>
    </dependencies>
  </deployment>
  <sub-deployment name="jboss-seam-booking.war">
    <exclusions>
      <module name="javax.faces.api" slot="main"/>
    </exclusions>
    <dependencies>
      <module name="javax.faces.api" slot="1.2"/>
    </dependencies>
  </sub-deployment>
</jboss-deployment-structure>
```

Redeploy the application by deleting the

`standalone/deployments/jboss-seam-booking.ear.failed` file and creating a blank `jboss-seam-booking.ear.dodeploy` file in the same directory.



Next Issue: java.lang.ClassNotFoundException: org.hibernate.validator.InvalidValue

When you deploy the application, the log contains the following error:

```
ERROR [org.apache.catalina.core.ContainerBase.[jboss.web].[default-host].[/seam-booking]] (MSC
service thread 1-6) Exception sending context initialized event to listener instance of class
org.jboss.seam.servlet.SeamListener: java.lang.RuntimeException: Could not create Component:
org.jboss.seam.international.statusMessages
    (... additional logs removed ...)
Caused by: java.lang.ClassNotFoundException: org.hibernate.validator.InvalidValue from [Module
"deployment.jboss-seam-booking.ear.jboss-seam.jar:main" from Service Module Loader]
```

What it means:

Again, the `ClassNotFoundException` indicates a missing dependency. In this case, it can not find the class `org.hibernate.validator.InvalidValue`.

How to resolve it:

There is a module “`org.hibernate.validator`”, but the JAR does not contain the `org.hibernate.validator.InvalidValue` class, so adding the module dependency will not resolve this issue.

In this case, the JAR containing the class was part of the EAP 5.1 deployment. We will look for the JAR that contains the missing class in the `EAP5_HOME/jboss-eap-5.1/seam/lib/` directory. To do this, open a console and type the following:

```
cd EAP5_HOME/jboss-eap-5.1/seam/lib
grep 'org.hibernate.validator.InvalidValue' `find . -name '*.jar'`
```

The result shows:

```
Binary file ./hibernate-validator.jar matches
Binary file ./test/hibernate-all.jar matches
```

In this case, we need to copy the `hibernate-validator.jar` to the `jboss-seam-booking.ear/lib/` directory:

```
cp EAP5_HOME/jboss-eap-5.1/seam/lib/hibernate-validator.jar jboss-seam-booking.ear/lib
```

Redeploy the application by deleting the

`standalone/deployments/jboss-seam-booking.ear.failed` file and creating a blank `jboss-seam-booking.ear.dodeploy` file in the same directory.



Next Issue: java.lang.InstantiationException: org.jboss.seam.jsf.SeamApplicationFactory

When you deploy the application, the log contains the following error:

```
INFO [javax.enterprise.resource.webcontainer.jsf.config] (MSC service thread 1-7) Unsanitized
stacktrace from failed start...: com.sun.faces.config.ConfigurationException: Factory
'javax.faces.application.ApplicationFactory' was not configured properly.
    at
com.sun.faces.config.processor.FactoryConfigProcessor.verifyFactoriesExist(FactoryConfigProcessor.
[jsf-impl-2.0.4-b09-jbossorg-4.jar:2.0.4-b09-jbossorg-4]
    (... additional logs removed ...)
Caused by: javax.faces.FacesException: org.jboss.seam.jsf.SeamApplicationFactory
    at javax.faces.FactoryFinder.getImplGivenPreviousImpl(FactoryFinder.java:606)
[jsf-api-1.2_13.jar:1.2_13-b01-FCS]
    (... additional logs removed ...)
    at
com.sun.faces.config.processor.FactoryConfigProcessor.verifyFactoriesExist(FactoryConfigProcessor.
[jsf-impl-2.0.4-b09-jbossorg-4.jar:2.0.4-b09-jbossorg-4]
    ... 11 more
Caused by: java.lang.InstantiationException: org.jboss.seam.jsf.SeamApplicationFactory
    at java.lang.Class.newInstance0(Class.java:340) [:1.6.0_25]
    at java.lang.Class.newInstance(Class.java:308) [:1.6.0_25]
    at javax.faces.FactoryFinder.getImplGivenPreviousImpl(FactoryFinder.java:604)
[jsf-api-1.2_13.jar:1.2_13-b01-FCS]
    ... 16 more
```

What it means:

The `com.sun.faces.config.ConfigurationException` and `java.lang.InstantiationException` indicate a dependency issue. In this case, it is not as obvious.



How to resolve it:

We need to find the module that contains the `com.sun.faces` classes. While there is no `com.sun.faces` module, there are two `com.sun.jsf-impl` modules. A quick check of the `jsf-impl-1.2_13.jar` in the `1.2` directory shows it contains the `com.sun.faces` classes.

As we did with the `javax.faces.FacesException` `ClassNotFoundException`, we want to use the JSF 1.2 version and not the JSF 2.0 version in main, so we need to be able to specify one and exclude the other. We need to modify the `jboss-deployment-structure.xml` to add the module dependency to the deployment section of the file. We also need to add it to the WAR subdeployment and exclude the JSF 2.0 module. The file should now look like this:

```
<jboss-deployment-structure xmlns="urn:jboss:deployment-structure:1.0">
  <deployment>
    <dependencies>
      <module name="javax.faces.api" slot="1.2" export="true"/>
      <module name="com.sun.jsf-impl" slot="1.2" export="true"/>
      <module name="org.apache.commons.logging" export="true"/>
      <module name="org.dom4j" export="true"/>
    </dependencies>
  </deployment>
  <sub-deployment name="jboss-seam-booking.war">
    <exclusions>
      <module name="javax.faces.api" slot="main"/>
      <module name="com.sun.jsf-impl" slot="main"/>
    </exclusions>
    <dependencies>
      <module name="javax.faces.api" slot="1.2"/>
      <module name="com.sun.jsf-impl" slot="1.2"/>
    </dependencies>
  </sub-deployment>
</jboss-deployment-structure>
```

Redeploy the application by deleting the

`standalone/deployments/jboss-seam-booking.ear.failed` file and creating a blank `jboss-seam-booking.ear.dodeploy` file in the same directory.

Next Issue: `java.lang.ClassNotFoundException: org.apache.commons.collections.ArrayStack`

When you deploy the application, the log contains the following error:



```
ERROR [org.apache.catalina.core.ContainerBase.[jboss.web].[default-host].[/seam-booking]] (MSC
service thread 1-1) Exception sending context initialized event to listener instance of class
com.sun.faces.config.ConfigureListener: java.lang.RuntimeException:
com.sun.faces.config.ConfigurationException: CONFIGURATION FAILED!
org.apache.commons.collections.ArrayStack from [Module "deployment.jboss-seam-booking.ear:main"
from Service Module Loader]
    (... additional logs removed ...)
Caused by: java.lang.ClassNotFoundException: org.apache.commons.collections.ArrayStack from
[Module "deployment.jboss-seam-booking.ear:main" from Service Module Loader]
```

What it means:

Again, the `ClassNotFoundException` indicates a missing dependency. In this case, it can not find the class `org.apache.commons.collections.ArrayStack`.



How to resolve it:

Find the module name in the `JBOSS_HOME/modules/` directory by looking for the `org/apache/commons/collections` path. The module name is “`org.apache.commons.collections`”.

Modify the `jboss-deployment-structure.xml` to add the module dependency to the deployment section of the file.

```
<module name="org.apache.commons.collections" export="true"/>
```

The `jboss-deployment-structure.xml` file should now look like this:

```
<jboss-deployment-structure xmlns="urn:jboss:deployment-structure:1.0">
  <deployment>
    <dependencies>
      <module name="javax.faces.api" slot="1.2" export="true"/>
      <module name="com.sun.jsf-impl" slot="1.2" export="true"/>
      <module name="org.apache.commons.logging" export="true"/>
      <module name="org.dom4j" export="true"/>
      <module name="org.apache.commons.collections" export="true"/>
    </dependencies>
  </deployment>
  <sub-deployment name="jboss-seam-booking.war">
    <exclusions>
      <module name="javax.faces.api" slot="main"/>
      <module name="com.sun.jsf-impl" slot="main"/>
    </exclusions>
    <dependencies>
      <module name="javax.faces.api" slot="1.2"/>
      <module name="com.sun.jsf-impl" slot="1.2"/>
    </dependencies>
  </sub-deployment>
</jboss-deployment-structure>
```

Redeploy the application by deleting the `standalone/deployments/jboss-seam-booking.ear.failed` file and creating a blank `jboss-seam-booking.ear.dodeploy` file in the same directory.

Next Issue: Services with missing/unavailable dependencies

When you deploy the application, the log contains the following error:



```
ERROR [org.jboss.as.deployment] (DeploymentScanner-threads - 2) {"Composite operation failed and
was rolled back. Steps that failed:" => {"Operation step-2" => {"Services with
missing/unavailable dependencies" =>
["jboss.deployment.subunit.\"jboss-seam-booking.ear\".\"jboss-seam-booking.jar\".component.Authent
missing [
jboss.naming.context.java.comp.jboss-seam-booking.\"jboss-seam-booking.jar\".AuthenticatorAction.\"
]\", \"jboss.deployment.subunit.\"jboss-seam-booking.ear\".\"jboss-seam-booking.jar\".component.Hotel
missing [
jboss.naming.context.java.comp.jboss-seam-booking.\"jboss-seam-booking.jar\".HotelSearchingAction.
]\", \"
<... additional logs removed ...>
\"jboss.deployment.subunit.\"jboss-seam-booking.ear\".\"jboss-seam-booking.jar\".component.BookingL
missing [
jboss.naming.context.java.comp.jboss-seam-booking.\"jboss-seam-booking.jar\".BookingListAction.\"e
]\", \"jboss.persistenceunit.\"jboss-seam-booking.ear/jboss-seam-booking.jar#bookingDatabase\"
missing [ jboss.naming.context.java.bookingDatasource ]}}}}
```

What it means:

When you get a “Services with missing/unavailable dependencies” error, look that the text within the brackets after “missing”.

In this case you see:

```
missing [
jboss.naming.context.java.comp.jboss-seam-booking.\"jboss-seam-booking.jar\".AuthenticatorAction.\"
]
```

The “/em” indicates an Entity Manager and datasource issue.

How to resolve it:

In WildFly 8, datasource configuration has changed and needs to be defined in the `standalone/configuration/standalone.xml` file. Since WildFly ships with an example database that is already defined in the `standalone.xml` file, we will modify the `persistence.xml` file to use that example database. Looking in the `standalone.xml` file, you can see that the `jndi-name` for the example database is “`java:jboss/datasources/ExampleDS`”.

Modify the `jboss-seam-booking.jar/META-INF/persistence.xml` file to comment the existing `jta-data-source` element and replace it as follows:

```
<!-- <jta-data-source>java:/bookingDatasource</jta-data-source> -->
<jta-data-source>java:jboss/datasources/ExampleDS</jta-data-source>
```

Redeploy the application by deleting the `standalone/deployments/jboss-seam-booking.ear.failed` file and creating a blank `jboss-seam-booking.ear.dodeploy` file in the same directory.



Next Issue: java.lang.ClassNotFoundException: org.hibernate.cache.HashtableCacheProvider

When you deploy the application, the log contains the following error:

```
ERROR [org.jboss.msc.service.fail] (MSC service thread 1-4) MSC00001: Failed to start service
jboss.persistenceunit."jboss-seam-booking.ear/jboss-seam-booking.jar#bookingDatabase":
org.jboss.msc.service.StartException in service
jboss.persistenceunit."jboss-seam-booking.ear/jboss-seam-booking.jar#bookingDatabase": Failed to
start service
  at org.jboss.msc.service.ServiceControllerImpl$StartTask.run(ServiceControllerImpl.java:1786)
  (... log messages removed ...)
Caused by: javax.persistence.PersistenceException: [PersistenceUnit: bookingDatabase] Unable to
build EntityManagerFactory
  at org.hibernate.ejb.Ejb3Configuration.buildEntityManagerFactory(Ejb3Configuration.java:903)
  {... log messages removed ...}
Caused by: org.hibernate.HibernateException: could not instantiate RegionFactory
[org.hibernate.cache.internal.bridge.RegionFactoryCacheProviderBridge]
  at org.hibernate.cfg.SettingsFactory.createRegionFactory(SettingsFactory.java:355)
  (... log messages removed ...)
Caused by: java.lang.reflect.InvocationTargetException
  at sun.reflect.NativeConstructorAccessorImpl.newInstance0(Native Method) [:1.6.0_25]
  (... log messages removed ...)
Caused by: org.hibernate.cache.CacheException: could not instantiate CacheProvider
[org.hibernate.cache.HashtableCacheProvider]
  at
org.hibernate.cache.internal.bridge.RegionFactoryCacheProviderBridge.<init>(RegionFactoryCacheProv
... 20 more
Caused by: java.lang.ClassNotFoundException: org.hibernate.cache.HashtableCacheProvider from
[Module "org.hibernate:main" from local module loader @12a3793 (roots:
/home/sgilda/tools/jboss7/modules)]
  at org.jboss.modules.ModuleClassLoader.findClass(ModuleClassLoader.java:191)
  (... log messages removed ...)
```

What it means:

The `ClassNotFoundException` indicates a missing dependency. In this case, it can not find the class `org.hibernate.cache.HashtableCacheProvider`.



How to resolve it:

There is no module for “org.hibernate.cache”. In this case, the JAR containing the class was part of the EAP 5.1 deployment. We will look for the JAR that contains the missing class in the `EAP5_HOME/jboss-eap-5.1/seam/lib/` directory.

To do this, open a console and type the following:

```
cd EAP5_HOME/jboss-eap-5.1/seam/lib
grep 'org.hibernate.validator.InvalidValue' `find . -name '*.jar'`
```

The result shows:

```
Binary file ./hibernate-core.jar matches
Binary file ./test/hibernate-all.jar matches
```

In this case, we need to copy the `hibernate-core.jar` to the `jboss-seam-booking.ear/lib/` directory:

```
cp EAP5_HOME/jboss-eap-5.1/seam/lib/hibernate-core.jar jboss-seam-booking.ear/lib
```

Redeploy the application by deleting the `standalone/deployments/jboss-seam-booking.ear.failed` file and creating a blank `jboss-seam-booking.ear.dodeploy` file in the same directory.

Next Issue: `java.lang.ClassCastException: org.hibernate.cache.HashtableCacheProvider`

When you deploy the application, the log contains the following error:



```
ERROR [org.jboss.msc.service.fail] (MSC service thread 1-2) MSC00001: Failed to start service
jboss.persistenceunit."jboss-seam-booking.ear/jboss-seam-booking.jar#bookingDatabase":
org.jboss.msc.service.StartException in service
jboss.persistenceunit."jboss-seam-booking.ear/jboss-seam-booking.jar#bookingDatabase": Failed to
start service
  at org.jboss.msc.service.ServiceControllerImpl$StartTask.run(ServiceControllerImpl.java:1786)
  (... log messages removed ...)
Caused by: javax.persistence.PersistenceException: [PersistenceUnit: bookingDatabase] Unable to
build EntityManagerFactory
  at org.hibernate.ejb.Ejb3Configuration.buildEntityManagerFactory(Ejb3Configuration.java:903)
  (... log messages removed ...)
Caused by: org.hibernate.HibernateException: could not instantiate RegionFactory
[org.hibernate.cache.internal.bridge.RegionFactoryCacheProviderBridge]
  at org.hibernate.cfg.SettingsFactory.createRegionFactory(SettingsFactory.java:355)
  (... log messages removed ...)
Caused by: java.lang.reflect.InvocationTargetException
  at sun.reflect.NativeConstructorAccessorImpl.newInstance0(Native Method) [:1.6.0_25]
  (... log messages removed ...)
Caused by: org.hibernate.cache.CacheException: could not instantiate CacheProvider
[org.hibernate.cache.HashtableCacheProvider]
  at
org.hibernate.cache.internal.bridge.RegionFactoryCacheProviderBridge.<init>(RegionFactoryCacheProv
... 20 more
Caused by: java.lang.ClassCastException: org.hibernate.cache.HashtableCacheProvider cannot be
cast to org.hibernate.cache.spi.CacheProvider
  at
org.hibernate.cache.internal.bridge.RegionFactoryCacheProviderBridge.<init>(RegionFactoryCacheProv
... 20 more
```

What it means:

A `ClassCastException` can be a result of many problems. If you look at this exception in the log, it appears the class `org.hibernate.cache.HashtableCacheProvider` extends `org.hibernate.cache.spi.CacheProvider` and is being loaded by a different class loader than the class it extends. The `org.hibernate.cache.HashtableCacheProvider` class is in in the `hibernate-core.jar` and is being loaded by the application class loader. The class it extends, `org.hibernate.cache.spi.CacheProvider`, is in the `org/hibernate/main/hibernate-core-4.0.0.Beta1.jar` and is implicitly loaded by that module.

This is not obvious, but due to changes in Hibernate 4, this problem is caused by a backward compatibility issue due moving the `HashtableCacheProvider` class into another package. This class was moved from the `org.hibernate.cache` package to the `org.hibernate.cache.internal` package. If you don't remove the `hibernate.cache.provider_class` property from the `persistence.xml` file, it will force the Seam application to bundle the old Hibernate libraries, resulting in `ClassCastExceptions`. In WildFly, you should move away from using `HashtableCacheProvider` and use `Infinispan` instead.



How to resolve it:

In WildFly, you need to comment out the `hibernate.cache.provider_class` property in the `jboss-seam-booking.jar/META-INF/persistence.xml` file as follows:

```
<!-- <property name="hibernate.cache.provider_class"
value="org.hibernate.cache.HashtableCacheProvider"/> -->
```

Redeploy the application by deleting the `standalone/deployments/jboss-seam-booking.ear.failed` file and creating a blank `jboss-seam-booking.ear.dodeploy` file in the same directory.

No more issues: Deployment errors should be resolved

At this point, the application should deploy without errors, but when you access the URL “<http://localhost:8080/seam-booking>” in a browser and attempt "Account Login", you will get a runtime error “The page isn't redirecting properly”. In the next section, we will step through each runtime issue and how to debug and resolve it.

23.4.3 Step 3: Debug and resolve runtime errors and exceptions

First Issue: `javax.naming.NameNotFoundException: Name 'jboss-seam-booking' not found in context "`

The application deploys successfully, but when you access the URL “<http://localhost:8080/seam-booking>” in a browser, you get “The page isn't redirecting properly” and the log contains the following error:

```
SEVERE [org.jboss.seam.jsf.SeamPhaseListener] (http--127.0.0.1-8080-1) swallowing exception:
java.lang.IllegalStateException: Could not start transaction
  at org.jboss.seam.jsf.SeamPhaseListener.begin(SeamPhaseListener.java:598) [jboss-seam.jar:]
  (... log messages removed ...)
Caused by: org.jboss.seam.InstantiationException: Could not instantiate Seam component:
org.jboss.seam.transaction.synchronizations
  at org.jboss.seam.Component.newInstance(Component.java:2170) [jboss-seam.jar:]
  (... log messages removed ...)
Caused by: javax.naming.NameNotFoundException: Name 'jboss-seam-booking' not found in context ''
  at org.jboss.as.naming.util.NamingUtils.nameNotFoundException(NamingUtils.java:109)
  (... log messages removed ...)
```

What it means:

A `NameNotFoundException` indicates a JNDI naming issue. JNDI naming rules have changed in WildFly and we need to modify the lookup names to follow the new rules.



How to resolve it:

To debug this, look earlier in the server log trace to what JNDI binding were used. Looking at the server log we see this:

```
15:01:16,138 INFO
[org.jboss.as.ejb3.deployment.processors.EjbJndiBindingsDeploymentUnitProcessor] (MSC service
thread 1-1) JNDI bindings for session bean named RegisterAction in deployment unit subdeployment
"jboss-seam-booking.jar" of deployment "jboss-seam-booking.ear" are as follows:

    java:global/jboss-seam-booking/jboss-seam-booking.jar/RegisterAction!org.jboss.seam.example.booki
java:app/jboss-seam-booking.jar/RegisterAction!org.jboss.seam.example.booking.Register
    java:module/RegisterAction!org.jboss.seam.example.booking.Register
    java:global/jboss-seam-booking/jboss-seam-booking.jar/RegisterAction
    java:app/jboss-seam-booking.jar/RegisterAction
    java:module/RegisterAction

15:01:16,138 INFO
[org.jboss.as.ejb3.deployment.processors.EjbJndiBindingsDeploymentUnitProcessor] (MSC service
thread 1-1) JNDI bindings for session bean named BookingListAction in deployment unit
subdeployment "jboss-seam-booking.jar" of deployment "jboss-seam-booking.ear" are as follows:

    java:global/jboss-seam-booking/jboss-seam-booking.jar/BookingListAction!org.jboss.seam.example.bo
java:app/jboss-seam-booking.jar/BookingListAction!org.jboss.seam.example.booking.BookingList
    java:module/BookingListAction!org.jboss.seam.example.booking.BookingList
    java:global/jboss-seam-booking/jboss-seam-booking.jar/BookingListAction
    java:app/jboss-seam-booking.jar/BookingListAction
    java:module/BookingListAction

15:01:16,138 INFO
[org.jboss.as.ejb3.deployment.processors.EjbJndiBindingsDeploymentUnitProcessor] (MSC service
thread 1-1) JNDI bindings for session bean named HotelBookingAction in deployment unit
subdeployment "jboss-seam-booking.jar" of deployment "jboss-seam-booking.ear" are as follows:

    java:global/jboss-seam-booking/jboss-seam-booking.jar/HotelBookingAction!org.jboss.seam.example.b
java:app/jboss-seam-booking.jar/HotelBookingAction!org.jboss.seam.example.booking.HotelBooking
    java:module/HotelBookingAction!org.jboss.seam.example.booking.HotelBooking
    java:global/jboss-seam-booking/jboss-seam-booking.jar/HotelBookingAction
    java:app/jboss-seam-booking.jar/HotelBookingAction
    java:module/HotelBookingAction

15:01:16,138 INFO
[org.jboss.as.ejb3.deployment.processors.EjbJndiBindingsDeploymentUnitProcessor] (MSC service
thread 1-1) JNDI bindings for session bean named AuthenticatorAction in deployment unit
subdeployment "jboss-seam-booking.jar" of deployment "jboss-seam-booking.ear" are as follows:

    java:global/jboss-seam-booking/jboss-seam-booking.jar/AuthenticatorAction!org.jboss.seam.example..
java:app/jboss-seam-booking.jar/AuthenticatorAction!org.jboss.seam.example.booking.Authenticator
    java:module/AuthenticatorAction!org.jboss.seam.example.booking.Authenticator
    java:global/jboss-seam-booking/jboss-seam-booking.jar/AuthenticatorAction
    java:app/jboss-seam-booking.jar/AuthenticatorAction
    java:module/AuthenticatorAction

15:01:16,139 INFO
[org.jboss.as.ejb3.deployment.processors.EjbJndiBindingsDeploymentUnitProcessor] (MSC service
thread 1-1) JNDI bindings for session bean named ChangePasswordAction in deployment unit
```



```
subdeployment "jboss-seam-booking.jar" of deployment "jboss-seam-booking.ear" are as follows:

  java:global/jboss-seam-booking/jboss-seam-booking.jar/ChangePasswordAction!org.jboss.seam.example
  java:app/jboss-seam-booking.jar/ChangePasswordAction!org.jboss.seam.example.booking.ChangePassword
  java:module/ChangePasswordAction!org.jboss.seam.example.booking.ChangePassword
  java:global/jboss-seam-booking/jboss-seam-booking.jar/ChangePasswordAction
  java:app/jboss-seam-booking.jar/ChangePasswordAction
  java:module/ChangePasswordAction

15:01:16,139 INFO
[org.jboss.as.ejb3.deployment.processors.EjbJndiBindingsDeploymentUnitProcessor] (MSC service
thread 1-1) JNDI bindings for session bean named HotelSearchingAction in deployment unit
subdeployment "jboss-seam-booking.jar" of deployment "jboss-seam-booking.ear" are as follows:

  java:global/jboss-seam-booking/jboss-seam-booking.jar/HotelSearchingAction!org.jboss.seam.example
  java:app/jboss-seam-booking.jar/HotelSearchingAction!org.jboss.seam.example.booking.HotelSearching
  java:module/HotelSearchingAction!org.jboss.seam.example.booking.HotelSearching
  java:global/jboss-seam-booking/jboss-seam-booking.jar/HotelSearchingAction
  java:app/jboss-seam-booking.jar/HotelSearchingAction
  java:module/HotelSearchingAction

15:01:16,140 INFO
[org.jboss.as.ejb3.deployment.processors.EjbJndiBindingsDeploymentUnitProcessor] (MSC service
thread 1-6) JNDI bindings for session bean named EjbSynchronizations in deployment unit
subdeployment "jboss-seam.jar" of deployment "jboss-seam-booking.ear" are as follows:

  java:global/jboss-seam-booking/jboss-seam/EjbSynchronizations!org.jboss.seam.transaction.LocalEjb
  java:app/jboss-seam/EjbSynchronizations!org.jboss.seam.transaction.LocalEjbSynchronizations
  java:module/EjbSynchronizations!org.jboss.seam.transaction.LocalEjbSynchronizations
  java:global/jboss-seam-booking/jboss-seam/EjbSynchronizations
  java:app/jboss-seam/EjbSynchronizations
  java:module/EjbSynchronizations

15:01:16,140 INFO
[org.jboss.as.ejb3.deployment.processors.EjbJndiBindingsDeploymentUnitProcessor] (MSC service
thread 1-6) JNDI bindings for session bean named TimerServiceDispatcher in deployment unit
subdeployment "jboss-seam.jar" of deployment "jboss-seam-booking.ear" are as follows:

  java:global/jboss-seam-booking/jboss-seam/TimerServiceDispatcher!org.jboss.seam.async.LocalTimerS
  java:app/jboss-seam/TimerServiceDispatcher!org.jboss.seam.async.LocalTimerServiceDispatcher
  java:module/TimerServiceDispatcher!org.jboss.seam.async.LocalTimerServiceDispatcher
  java:global/jboss-seam-booking/jboss-seam/TimerServiceDispatcher
  java:app/jboss-seam/TimerServiceDispatcher
  java:module/TimerServiceDispatcher
```

We need to modify the WAR's `lib/components.xml` file to use the new JNDI bindings. In the log, note the EJB JNDI bindings all start with `"java:app/jboss-seam-booking.jar"`

Replace the `<core:init>` element as follows:

```
<!--      <core:init jndi-pattern="jboss-seam-booking/#{ejbName}/local" debug="true"
distributable="false"/> -->
<core:init jndi-pattern="java:app/jboss-seam-booking.jar/#{ejbName}" debug="true"
distributable="false"/>
```



Next, we need to add the EjbSynchronizations and TimerServiceDispatcher JNDI bindings. Add the following component elements to the file:

```
<component class="org.jboss.seam.transaction.EjbSynchronizations"
jndi-name="java:app/jboss-seam/EjbSynchronizations" />
<component class="org.jboss.seam.async.TimerServiceDispatcher"
jndi-name="java:app/jboss-seam/TimerServiceDispatcher" />
```

The components.xml file should now look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<components xmlns="http://jboss.com/products/seam/components"
  xmlns:core="http://jboss.com/products/seam/core"
  xmlns:security="http://jboss.com/products/seam/security"
  xmlns:transaction="http://jboss.com/products/seam/transaction"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation=
    "http://jboss.com/products/seam/core http://jboss.com/products/seam/core-2.2.xsd
    http://jboss.com/products/seam/transaction
    http://jboss.com/products/seam/transaction-2.2.xsd
    http://jboss.com/products/seam/security
    http://jboss.com/products/seam/security-2.2.xsd
    http://jboss.com/products/seam/components
    http://jboss.com/products/seam/components-2.2.xsd">

  <!-- <core:init jndi-pattern="jboss-seam-booking/#{ejbName}/local" debug="true"
distributable="false"/> -->
  <core:init jndi-pattern="java:app/jboss-seam-booking.jar/#{ejbName}" debug="true"
distributable="false"/>

  <core:manager conversation-timeout="120000"
    concurrent-request-timeout="500"
    conversation-id-parameter="cid"/>

  <transaction:ejb-transaction/>

  <security:identity authenticate-method="#{authenticator.authenticate}"/>

  <component class="org.jboss.seam.transaction.EjbSynchronizations"
    jndi-name="java:app/jboss-seam/EjbSynchronizations" />
  <component class="org.jboss.seam.async.TimerServiceDispatcher"
    jndi-name="java:app/jboss-seam/TimerServiceDispatcher" />
</components>
```

Redeploy the application by deleting the standalone/deployments/jboss-seam-booking.ear.failed file and creating a blank jboss-seam-booking.ear.dodeploy file in the same directory.

At this point, the application should deploy and run without error. When you access the URL “<http://localhost:8080/seam-booking>” in a browser, you will be able to login successfully.



23.4.4 Step 4: Access the application

Access the URL "<http://localhost:8080/seam-booking/>" in a browser and login with demo/demo. You should see the Booking welcome page.

23.4.5 Summary of Changes

Although it would be much more efficient to determine dependencies in advance and add the implicit dependencies in one step, this exercise shows how problems appear in the log and provides some information on how to debug and resolve them.

The following is a summary of changes made to the application when migrating it to WildFly:

1. We created a `jboss-deployment-structure.xml` file in the EAR's `META-INF/` directory. We added "dependencies" and "exclusions" to resolve `ClassNotFoundException`s. This file contains the following data:

```
<jboss-deployment-structure xmlns="urn:jboss:deployment-structure:1.0">
  <deployment>
    <dependencies>
      <module name="javax.faces.api" slot="1.2" export="true"/>
      <module name="com.sun.jsf-impl" slot="1.2" export="true"/>
      <module name="org.apache.commons.logging" export="true"/>
        <module name="org.dom4j" export="true"/>
      <module name="org.apache.commons.collections" export="true"/>
    </dependencies>
  </deployment>
  <sub-deployment name="jboss-seam-booking.war">
    <exclusions>
      <module name="javax.faces.api" slot="main"/>
      <module name="com.sun.jsf-impl" slot="main"/>
    </exclusions>
    <dependencies>
      <module name="javax.faces.api" slot="1.2"/>
      <module name="com.sun.jsf-impl" slot="1.2"/>
    </dependencies>
  </sub-deployment>
</jboss-deployment-structure>
```

2. We copied the following JARs from the `EAP5_HOME/jboss-eap-5.1/seam/lib/` directory to the `jboss-seam-booking.ear/lib/` directory to resolve `ClassNotFoundException`s:

```
hibernate-core.jar
hibernate-validator.jar
```



3. We modified the `jboss-seam-booking.jar/META-INF/persistence.xml` file as follows.
 1. First, we changed the `jta-data-source` element to use the Example database that ships with AS7:

```
<!-- <jta-data-source>java:/bookingDatasource</jta-data-source> -->  
<jta-data-source>java:jboss/datasources/ExampleDS</jta-data-source>
```

2. Next, we commented out the `hibernate.cache.provider_class` property:

```
<!-- <property name="hibernate.cache.provider_class"  
value="org.hibernate.cache.HashtableCacheProvider"/> -->
```

4. We modified the WAR's `lib/components.xml` file to use the new JNDI bindings
 1. We replaced the `<core:init>` existing element as follows:

```
<!-- <core:init jndi-pattern="jboss-seam-booking/#{ejbName}/local" debug="true"  
distributable="false"/> -->  
<core:init jndi-pattern="java:app/jboss-seam-booking.jar/#{ejbName}" debug="true"  
distributable="false"/>
```

2. We added component elements for the "EjbSynchronizations" and "TimerServiceDispatcher" JNDI bindings

```
<component class="org.jboss.seam.transaction.EjbSynchronizations"  
jndi-name="java:app/jboss-seam/EjbSynchronizations"/>  
  <component class="org.jboss.seam.async.TimerServiceDispatcher"  
jndi-name="java:app/jboss-seam/TimerServiceDispatcher"/>
```

The unmodified EAR from EAP 5.1 (`jboss-seam-booking-eap51.ear.tar.gz`) and the EAR as modified to run on AS7 (`jboss-seam-booking-as7.ear.tar.gz`) are attached to this document.



24 How do I migrate my application from AS7 to WildFly

- [About this Document](#)
- [Overview of WildFly](#)
- [Server Migration](#)
 - [JacORB Subsystem](#)
 - [JacORB Subsystem Configuration](#)
 - [JBoss Web Subsystem](#)
 - [JBoss Web Subsystem Configuration](#)
 - [WebSockets](#)
 - [Messaging Subsystem](#)
 - [Messaging Subsystem Configuration](#)
 - [Management model](#)
 - [XML Configuration](#)
 - [Messaging Interceptors](#)
 - [JMS Destinations](#)
 - [Messaging Logging](#)
 - [Messaging Data](#)



- [Application Migration](#)
 - [EJBs](#)
 - [CMP Entity EJBs](#)
 - [EJB Client](#)
 - [Default Remote Connection Port](#)
 - [Default Connector](#)
 - [JMS](#)
 - [Proprietary JMS Resource Definitions](#)
 - [External JMS Clients](#)
 - [JPA \(and Hibernate\)](#)
 - [Applications That Plan to Use Hibernate ORM 5.0](#)
 - [Applications that currently use Hibernate ORM 4.0 - 4.3](#)
 - [Applications that currently use Hibernate 3](#)
 - [Web Applications](#)
 - [JBoss Web Valves](#)
 - [Web Services](#)
 - [CXF Spring Webservices](#)
 - [JAX-RPC](#)
 - [JAX-RS 2.0](#)
 - [REST Client API](#)
 - [Application Clustering](#)
 - [HA Singleton](#)
 - [Stateful Session EJB Clustering](#)
 - [Web Session Clustering](#)
 - [Other Specifications and Frameworks](#)
 - [Remote JNDI Clients](#)
 - [JSR-88](#)
 - [Module Dependencies](#)

24.1 About this Document

The purpose of this guide is to document changes that are needed to successfully run and deploy AS 7 applications on WildFly. It provides information on to resolve deployment and runtime problems and how to prevent changes in application behavior. This is the first step in moving to the new platform. Once the application is successfully deployed and running on the new platform, plans can be made to upgrade individual components to use the new functions and features of WildFly.



24.2 Overview of WildFly

The list of WildFly new functionality is extensive, being the most relevant, with respect to server and application migrations:

- Java EE7 - WildFly is a certified implementation of Java EE7, meeting both the Web and the Full profiles, and already includes support for the latest iterations of CDI (1.2) and Web Sockets (1.1).
- Undertow - A new cutting-edge web server in WildFly, designed for maximum throughput and scalability, including environments with over a million connections. And the latest web technologies, such as the new HTTP/2 standard, are already onboard.
- Apache ActiveMQ Artemis - WildFly's new JMS broker. Based on an code donation from HornetQ, this Apache subproject provides outstanding performance based on a proven non-blocking architecture.
- IronJacamar 1.2 - The latest IronJacamar provides a stable and feature rich JCA & Datasources support.
- JBossWS 5 - The fifth generation of JBossWS, a major leap forward, brings new features and performances improvements to WildFly Web Services
- RESTEasy 3 - WildFly includes the latest generation of RESTEasy, which goes beyond the standard Java EE REST APIs (JAX-RS 2.0), by also providing a number of useful extensions, such as JSON Web Encryption, Jackson, Yaml, JSON-P, and Jettison.
- OpenJDK ORB - WildFly switched the IIOP implementation from JacORB, to a downstream branch of the OpenJDK Orb, leading to better interoperability with the JVM ORB and the Java EE RI.
- Feature Rich Clustering - Clustering support was heavily refactored in WildFly, and includes several APIs for applications
- Port Reduction - By utilising HTTP upgrade, WildFly has moved nearly all of its protocols to be multiplexed over just two HTTP ports: a management port (9990), and an application port (8080).
- Enhanced Logging - The management API now supports the ability to list and view the available log files on a server, or even define custom formatters other than the default pattern formatter. Deployment's logging setup is also greatly enhanced.

The support for some technologies was removed, due to the high maintenance cost, low community interest, and much better alternative solutions:

- CMP EJB - JPA offers a much more performant and flexible API
- JAX-RPC - JAX-WS offers a much more accurate and complete solution
- JSR-88 - With very little adoption, the more complete deployment APIs provided by vendors are preferred

24.3 Server Migration

Migrating an AS7 server to WildFly consists of migrating custom configuration files, and some persisted data that may exist.



24.3.1 JacORB Subsystem

WildFly ORB support is provided by the JDK itself, instead of relying on JacORB. A subsystem configuration migration is required.

JacORB Subsystem Configuration

The extension's module `org.jboss.as.jacorb` is replaced by module `*org.wildfly.iiop-openjdk`, while the subsystem configuration namespace `urn:jboss:domain:jacorb:2.0` is replaced by `urn:jboss:domain:iiop-openjdk:1.0`.

The XML configuration of the new subsystem accepts only a subset of the legacy elements/attributes. Consider the following example of the JacORB subsystem configuration, containing all valid elements and attributes:

```
<subsystem xmlns="urn:jboss:domain:jacorb:1.3">
  <orb name="JBoss" print-version="off" use-imr="off" use-bom="off" cache-typecodes="off"
    cache-poa-names="off" giop-minor-version="2" socket-binding="jacorb"
    ssl-socket-binding="jacorb-ssl">
    <connection retries="5" retry-interval="500" client-timeout="0" server-timeout="0"
      max-server-connections="500" max-managed-buf-size="24" outbuf-size="2048"
      outbuf-cache-timeout="-1"/>
    <initializers security="off" transactions="spec"/>
  </orb>
  <poa monitoring="off" queue-wait="on" queue-min="10" queue-max="100">
    <request-processors pool-size="10" max-threads="32"/>
  </poa>
  <naming root-context="JBoss/Naming/root" export-corballoc="on"/>
  <interop sun="on" comet="off" iona="off" chunk-custom-rmi-valuetypes="on"
    lax-boolean-encoding="off" indirection-encoding-disable="off"
    strict-check-on-tc-creation="off"/>
  <security support-ssl="off" add-component-via-interceptor="on" client-supports="MutualAuth"
    client-requires="None" server-supports="MutualAuth" server-requires="None"/>
  <properties>
    <property name="some_property" value="some_value"/>
  </properties>
</subsystem>
```

Properties that are not supported and have to be removed:

- `<orb/>`: `client-timeout`, `max-managed-buf-size`, `max-server-connections`, `outbuf-cache-timeout`, `outbuf-size`, `connection retries`, `retry-interval`, `name`, `server-timeout`
- `<poa/>`: `queue-min`, `queue-max`, `pool-size`, `max-threads`

On-off properties: have to either be removed or in off mode:

- `<orb/>`: `cache-poa-names`, `cache-typecodes`, `print-version`, `use-bom`, `use-imr`
- `<interop/>`: all except `sun`
- `<poa/>`: `monitoring`, `queue-wait`



In case the legacy subsystem configuration is available, such configuration may be migrated to the new subsystem by invoking its `migrate` operation, using the CLI management client:

```
/subsystem=jacorb:migrate
```

There is also a `describe-migration` operation that returns a list of all the management operations that are performed to migrate from the legacy subsystem to the new one:

```
/subsystem=jacorb:describe-migration
```

Both `migrate` and `describe-migration` will also display a list of migration-warnings if there are some resource or attributes that can not be migrated automatically. The following is a list of these warnings:

- Properties X cannot be emulated using OpenJDK ORB and are not supported
This warning means that mentioned properties are not supported and won't be included in the new subsystem configuration. As a result of that admin must be aware that any behaviour implied by those properties would be inexistent. Admin has to check whether subsystem is able to operate correctly without that behaviour on the new server. Unsupported properties: `cache-poa-names`, `cache-typecodes`, `chunk-custom-rmi-valuetypes`, `client-timeout`, `comet`, `indirection-encoding-disable`, `iona`, `lax-boolean-encoding`, `max-managed-buf-size`, `max-server-connections`, `max-threads`, `outbuf-cache-timeout`, `outbuf-size`, `queue-max`, `queue-min`, `poa-monitoring`, `print-version`, `retries`, `retry-interval`, `queue-wait`, `server-timeout`, `strict-check-on-tc-creation`, `use-bom`, `use-imr`.
- The properties X use expressions. Configuration properties that are used to resolve those expressions should be transformed manually to the new `iiop-openjdk` subsystem format
Admin has to transform all the configuration files to work correctly with the `jacorb` subsystem. f.e. `jacorb` has a property `giop-minor-version` whereas `openjdk` uses property `giop-version`. Let's suppose we use '1' minor version in `jacorb` and have it configured in `standalone.conf` file as system variable: `-Diiop-giop-minor-version=1`. Admin is responsible for changing this variable to `1.1` after the migration to make sure that the new subsystem will work correctly.

24.3.2 JBoss Web Subsystem

JBoss Web is replaced by Undertow in WildFly, which means that the legacy subsystem configuration should be migrated to WildFly's Undertow subsystem configuration.

JBoss Web Subsystem Configuration

The extension's module `org.jboss.as.web` *is replaced by module `*org.wildfly.extension.undertow`, while the subsystem configuration namespace `urn:jboss:domain:web:*` is replaced by `urn:jboss:domain:undertow:3.0`.

The XML configuration of the new subsystem is relatively different. Consider the following example of the JBoss Web subsystem configuration, containing all valid elements and attributes:



```
<?xml version="1.0" encoding="UTF-8"?>
<subsystem xmlns="urn:jboss:domain:web:2.2" default-virtual-server="default-host" native="true"
default-session-timeout="30" instance-id="foo">
  <configuration>
    <static-resources listings="true"
      sendfile="1000"
      file-encoding="utf-8"
      read-only="true"
      webdav="false"
      secret="secret"
      max-depth="5"
      disabled="false"
    />
    <jsp-configuration development="true"
      disabled="false"
      keep-generated="true"
      trim-spaces="true"
      tag-pooling="true"
      mapped-file="true"
      check-interval="20"
      modification-test-interval="1000"
      recompile-on-fail="true"
      smap="true"
      dump-smap="true"
      generate-strings-as-char-arrays="true"
      error-on-use-bean-invalid-class-attribute="true"
      scratch-dir="/some/dir"
      source-vm="1.7"
      target-vm="1.7"
      java-encoding="utf-8"
      x-powered-by="true"
      display-source-fragment="true" />
    <mime-mapping name="ogx" value="application/ogg" />
    <welcome-file>titi</welcome-file>
  </configuration>
  <connector name="http" scheme="http"
    protocol="HTTP/1.1"
    socket-binding="http"
    enabled="true"
    enable-lookups="false"
    proxy-binding="reverse-proxy"
    max-post-size="2097153"
    max-save-post-size="512"
    redirect-binding="https"
    max-connections="300"
    secure="false"
    executor="some-executor"
  />
  <connector name="https" scheme="https" protocol="HTTP/1.1" secure="true"
socket-binding="https">
    <ssl certificate-key-file="{file-base}/server.keystore"
      ca-certificate-file="{file-base}/jsse.keystore"
      key-alias="test"
      password="changeit"
      cipher-suite="SSL_RSA_WITH_3DES_EDE_CBC_SHA"
      protocol="SSLv3"
      verify-client="true"
    />
  </connector>
</subsystem>
```



```
        verify-depth="3"
        certificate-file="certificate-file.ext"
        ca-revocation-url="https://example.org/some/url"
        ca-certificate-password="changeit"
        keystore-type="JKS"
        truststore-type="JKS"
        session-cache-size="512"
        session-timeout="3000"
        ssl-protocol="RFC4279"
    />
</connector>
<connector name="http-vs" scheme="http" protocol="HTTP/1.1" socket-binding="http" >
    <virtual-server name="vs1" />
    <virtual-server name="vs2" />
</connector>
<virtual-server name="default-host" enable-welcome-root="true" default-web-module="foo.war">
    <alias name="localhost" />
    <alias name="example.com" />
    <access-log resolve-hosts="true" extended="true" pattern="extended" prefix="prefix"
rotate="true" >
        <directory relative-to="jboss.server.base.dir" path="toto" />
    </access-log>
    <rewrite name="myrewrite" pattern="^/helloworld(.*)" substitution="/helloworld/test.jsp"
flags="L" />
    <rewrite name="with-conditions" pattern="^/helloworld(.*)"
substitution="/helloworld/test.jsp" flags="L" >
        <condition name="https" pattern="off" test="%{HTTPS}" flags="NC"/>
        <condition name="user" test="%{USER}" pattern="toto" flags="NC"/>
        <condition name="no-flags" test="%{USER}" pattern="toto"/>
    </rewrite>
    <sso reauthenticate="true" domain="myDomain" cache-name="myCache"
        cache-container="cache-container" http-only="true"/>
</virtual-server>
<virtual-server name="vs1" />
<virtual-server name="vs2" />
<valve name="myvalve" module="org.jboss.some.module" class-name="org.jboss.some.class"
enabled="true">
    <param param-name="param-name" param-value="some-value"/>
</valve>
<valve name="accessLog" module="org.jboss.as.web"
class-name="org.apache.catalina.valves.AccessLogValve">
    <param param-name="prefix" param-value="myapp_access_log." />
    <param param-name="suffix" param-value=".log" />
    <param param-name="rotatable" param-value="true" />
    <param param-name="fileDateFormat" param-value="yyyy-MM-dd" />
    <param param-name="pattern" param-value="common" />
    <param param-name="directory" param-value="${jboss.server.log.dir}" />
    <param param-name="resolveHosts" param-value="false"/>
    <param param-name="conditionIf" param-value="log-enabled"/>
</valve>
<valve name="request-dumper" module="org.jboss.as.web"
class-name="org.apache.catalina.valves.RequestDumperValve"/>
<valve name="remote-addr" module="org.jboss.as.web"
class-name="org.apache.catalina.valves.RemoteAddrValve">
    <param param-name="allow" param-value="127.0.0.1,127.0.0.2" />
    <param param-name="deny" param-value="192.168.1.20" />
</valve>
<valve name="crawler" class-name="org.apache.catalina.valves.CrawlerSessionManagerValve"
```



```
module="org.jboss.as.web" >
  <param param-name="sessionInactiveInterval" param-value="1" />
  <param param-name="crawlerUserAgents" param-value="Google" />
</valve>
<valve name="proxy" class-name="org.apache.catalina.valves.RemoteIpValve"
module="org.jboss.as.web" >
  <param param-name="internalProxies" param-value="192\.168\.0\.10|192\.168\.0\.11" />
  <param param-name="remoteIpHeader" param-value="x-forwarded-for" />
  <param param-name="proxiesHeader" param-value="x-forwarded-by" />
  <param param-name="trustedProxies" param-value="proxy1|proxy2" />
</valve>
</subsystem>
```

FIXME compare with Undertow, list unsupported features

It's possible to do a migration of the legacy subsystem configuration, and related persisted data. , by invoking the legacy's subsystem's `migrate` operation, using the CLI management client:

```
/subsystem=web:migrate
```

There is also a `describe-migration` operation that returns a list of all the management operations that are performed to migrate from the legacy subsystem to the new one:

```
/subsystem=web:describe-migration
```

Both `migrate` and `describe-migration` will also display a list of migration-warnings if there are some resource or attributes that can not be migrated automatically. The following is a list of these warnings:



- Could not migrate resource X

This warning means that mentioned resource configuration is not supported and won't be included in the new subsystem configuration. As a result of that admin must be aware that any behaviour implied by those resources would be inexistent. Admin has to check whether subsystem is able to operate correctly without that behaviour on the new server.

FIXME must document which are the resources that trigger this

- Could not migrate attribute X from resource Y.

This warning means that mentioned resource configuration property is not supported and won't be included in the new subsystem configuration. As a result of that admin must be aware that any behaviour implied by those properties would be inexistent. Admin has to check whether subsystem is able to operate correctly without that behaviour on the new server.

FIXME must document which are the properties that trigger this

- Could not migrate SSL connector as no SSL config is defined
- Could not migrate verify-client attribute %s to the Undertow equivalent
- Could not migrate verify-client expression %s
- Could not migrate valve X

This warning means that mentioned valve configuration is not supported and won't be included in the new subsystem configuration. As a result of that admin must be aware that any behaviour implied by those resources would be inexistent. Admin has to check whether subsystem is able to operate correctly without that behaviour on the new server. This warning may happen for :

- org.apache.catalina.valves.RemoteAddrValve : must have at least one allowed or denied value.
- org.apache.catalina.valves.RemoteHostValve : must have at least one allowed or denied value.
- org.apache.catalina.authenticator.BasicAuthenticator
- org.apache.catalina.authenticator.DigestAuthenticator
- org.apache.catalina.authenticator.FormAuthenticator
- org.apache.catalina.authenticator.SSLAuthenticator
- org.apache.catalina.authenticator.SpnegoAuthenticator
- custom valves

- Could not migrate attribute X from valve Y

This warning means that mentioned valve configuration property is not supported and won't be included in the new subsystem configuration. As a result of that admin must be aware that any behaviour implied by those properties would be inexistent. Admin has to check whether subsystem is able to operate correctly without that behaviour on the new server. This warning may happen for :

- org.apache.catalina.valves.AccessLogValve : if you use the following parameters *resolveHosts* , *fileDateFormat*, *renameOnRotate*, *encoding*, *locale*, *requestAttributesEnabled*, *buffered*.
- org.apache.catalina.valves.ExtendedAccessLogValve : if you use the following parameters *resolveHosts*, *fileDateFormat*, *renameOnRotate*, *encoding*, *locale*, *requestAttributesEnabled*, *buffered*.
- org.apache.catalina.valves.RemoteIpValve:
 - if *remoteIpHeader* is defined and isn't set to "x-forwarded-for".
 - if *protocolHeader* is defined and isn't set to "x-forwarded-proto".
 - if you use the following parameters *httpServerPort* and *httpsServerPort* .



Also, please note that Undertow doesn't support JBoss Web **valves**, but some of these may be migrated to Undertow handlers, and JBoss Web subsystem's `migrate` operation do that too.

Here is a list of those valves and their corresponding Undertow handler:

Valve	Handler
<code>org.apache.catalina.valves.AccessLogValve</code>	<code>io.undertow.server.handlers.accesslog.AccessLogHandler</code>
<code>org.apache.catalina.valves.ExtendedAccessLogValve</code>	<code>io.undertow.server.handlers.accesslog.AccessLogHandler</code>
<code>org.apache.catalina.valves.RequestDumperValve</code>	<code>io.undertow.server.handlers.RequestDumpingHandler</code>
<code>org.apache.catalina.valves.RewriteValve</code>	<code>io.undertow.server.handlers.SetAttributeHandler</code>
<code>org.apache.catalina.valves.RemoteHostValve</code>	<code>io.undertow.server.handlers.AccessControlListHandler</code>
<code>org.apache.catalina.valves.RemoteAddrValve</code>	<code>io.undertow.server.handlers.IPAddressAccessControlHandler</code>
<code>org.apache.catalina.valves.RemoteIpValve</code>	<code>io.undertow.server.handlers.ProxyPeerAddressHandler</code>
<code>org.apache.catalina.valves.StuckThreadDetectionValve</code>	<code>io.undertow.server.handlers.StuckThreadDetectionHandler</code>
<code>org.apache.catalina.valves.CrawlerSessionManagerValve</code>	<code>io.undertow.servlet.handlers.CrawlerSessionManagerHandler</code>

The `org.apache.catalina.valves.JDBCAccessLogValve` can't be automatically migrated to `io.undertow.server.handlers.JDBCLogHandler` as the expectations differ.

The migration can be done manually thought :

1. create the driver module and add the driver to the list of available drivers
2. create a datasource pointing to the database where the log entries are going to be stored
3. add an **expression-filter** definition with the following expression:
"jdbc-access-log(datasource='datasource-jndi-name')

```
<valve name="jdbc" module="org.jboss.as.web"
class-name="org.apache.catalina.valves.JDBCAccessLogValve">
  <param param-name="driverName" param-value="com.mysql.jdbc.Driver" />
  <param param-name="connectionName" param-value="root" />
  <param param-name="connectionPassword" param-value="password" />
  <param param-name="connectionURL"
param-value="jdbc:mysql://localhost:3306/wildfly?zeroDateTimeBehavior=convertToNull" />
  <param param-name="format" param-value="combined" />
</valve>
```

should become:



```
<subsystem xmlns="urn:jboss:domain:datasources:1.2">
  <datasources>
    <datasource jndi-name="java:jboss/datasources/accessLogDS" pool-name="ccessLogDS"
enabled="true" use-java-context="true">

<connection-url>jdbc:mysql://localhost:3306/wildfly?zeroDateTimeBehavior=convertToNull</conn
<driver>mysql</driver>
    <security>
      <user-name>root</user-name>
      <password>password</password>
    </security>
  </datasource>
  ...
  <drivers>
    <driver name="mysql" module="com.mysql">
      <driver-class>com.mysql.jdbc.Driver</driver-class>
    </driver>
  ...
</drivers>
</datasources>
</subsystem>
...
<subsystem xmlns="urn:jboss:domain:undertow:3.1"
default-virtual-host="default-virtual-host" default-servlet-container="myContainer"
    default-server="some-server" instance-id="some-id" statistics-enabled="true">
  ...
  <server name="some-server" default-host="other-host" servlet-container="myContainer">
  ...
    <host name="other-host" alias="www.mysite.com, ${prop.value:default-alias}"
default-web-module="something.war" disable-console-redirect="true">
      <location name="/" handler="welcome-content" />
      <filter-ref name="jdbc-access" />
    </host>
  </server>
  ...
  <filters>
    <expression-filter name="jdbc-access"
expression="jdbc-access-log(datasource='java:jboss/datasources/accessLogDS')" />
  ...
</filters>

</subsystem>
```

Please note that any custom valve won't be migrated at all and will just be removed from the configuration. Also the authentication related valves are to be replaced by Undertow authentication mechanisms, and this have to be done manually.

FIXME how this last "manual" replacement is done? Need whole process documented and concrete example



WebSockets

In AS7, to use WebSockets, you had to configure the 'http' **connector** in the **web** subsystem of the server configuration file to use the NIO2 protocol. The following is an example of the Management CLI command to configure WebSockets in the previous releases.

```
/subsystem=web/connector=http/:write-attribute(name=protocol,value=org.apache.coyote.http11.Http11
```

WebSockets are a requirement of the Java EE 7 specification and the default configuration is included in WildFly. More complex WebSocket configuration is done in the **servlet-container** of the **undertow** subsystem of the server configuration file.

You no longer need to configure the server for default WebSocket support.

FIXME isn't `<websockets />` required for that?

24.3.3 Messaging Subsystem

WildFly JMS support is provided by ActiveMQ Artemis, instead of HornetQ. It's possible to do a migration of the legacy subsystem configuration, and related persisted data.

Messaging Subsystem Configuration

The extension's module **org.jboss.as.messaging** is replaced by module **org.wildfly.extension.messaging-activemq**, while the subsystem configuration namespace **urn:jboss:domain:messaging:3.0** is replaced by **urn:jboss:domain:messaging-activemq:1.0**.

Management model

In most cases, an effort was made to keep resource and attribute names as similar as possible to those used in previous releases. The following table lists some of the changes.

HornetQ name	ActiveMQ name
hornetq-server	server
hornetq-serverType	serverType
connectors	connector
discovery-group-name	discovery-group

The management operations invoked on the new messaging-subsystem starts with `/subsystem=messaging-activemq/server=X` while the legacy messaging subsystem was at `/subsystem=messaging/hornetq-server=X`.

In case the legacy subsystem configuration is available, such configuration may be migrated to the new subsystem by invoking its `migrate` operation, using the CLI management client:



```
/subsystem=messaging:migrate
```

There is also a `describe-migration` operation that returns a list of all the management operations that are performed to migrate from the legacy subsystem to the new one:

```
/subsystem=messaging:describe-migration
```

Both `migrate` and `describe-migration` will also display a list of migration-warnings if there are some resource or attributes that can not be migrated automatically. The following is a list of these warnings:

- The migrate operation can not be performed: the server must be in admin-only mode
The `migrate` operation requires starting the server in admin-only mode, which is done by adding parameter `--admin-only` to the server start command, e.g.

```
./standalone.sh --admin-only
```

- Can not migrate attribute `local-bind-address` from resource X. Use instead the `socket-attribute` to configure this broadcast-group.
- Can not migrate attribute `local-bind-port` from resource X. Use instead the `socket-binding` attribute to configure this broadcast-group.
- Can not migrate attribute `group-address` from resource X. Use instead the `socket-binding` attribute to configure this broadcast-group.
- Can not migrate attribute `group-port` from resource X. Use instead the `socket-binding` attribute to configure this broadcast-group.
Broadcast-group resources no longer accept `local-bind-address`, `local-bind-port`, `group-address`, `group-port` attributes. It only accepts a `socket-binding`. The warning notifies that resource X has an unsupported attribute. The user will have to set the `socket-binding` attribute on the resource and ensures it corresponds to a defined `socket-binding` resource.
- Classes providing the `%s` are discarded during the migration. To use them in the new `messaging-activemq` subsystem, you will have to extend the Artemis-based `Interceptor`.
Messaging interceptors support is significantly different in WildFly 10, any interceptors configured in the legacy subsystem are discarded during migration. Please refer to the `Messaging Interceptors` section to learn how to migrate legacy Messaging interceptors.
- Can not migrate the HA configuration of X. Its `shared-store` and `backup` attributes holds expressions and it is not possible to determine unambiguously how to create the corresponding `ha-policy` for the `messaging-activemq`'s server.
If the `hornetq-server X`'s `shared-store` or `backup` attributes hold an expression, such as `${xxx}`, then it's not possible to determine the actual `ha-policy` of the migrated server. In that case, we discard it and the user will have to add the correct `ha-policy` afterwards (`ha-policy` is a single resource underneath the `messaging-activemq`'s server resource).



- Can not migrate attribute local-bind-address from resource X. Use instead the socket-binding attribute to configure this discovery-group. Can not migrate attribute local-bind-port from resource X. Use instead the socket-binding attribute to configure this discovery-group.
- Can not migrate attribute group-address from resource X. Use instead the socket-binding attribute to configure this discovery-group.
- Can not migrate attribute group-port from resource X. Use instead the socket-binding attribute to configure this discovery-group.
discovery-group resources no longer accept local-bind-address, local-bind-port, group-address, group-port attributes. It only accepts a socket-binding. The warning notifies that resource X has an unsupported attribute.
The user will have to set the socket-binding attribute on the resource and ensures it corresponds to a defined socket-binding resource.
- Can not create a legacy-connection-factory based on connection-factory X. It uses a HornetQ in-vm connector that is not compatible with Artemis in-vm connector
Legacy subsystem's remote connection-factory resources are migrated into legacy-connection-factory resources, to allow old EAP6 clients to connect to EAP7. However a connection-factory using in-vm will not be migrated, because a in-vm client will be based on EAP7, not EAP 6. In other words, legacy-connection-factory are created only when the CF is using remote connectors, and this warning notifies about in-vm connection-factory X not migrated.
- Can not migrate attribute X from resource Y. The attribute uses an expression that can be resolved differently depending on system properties. After migration, this attribute must be added back with an actual value instead of the expression.
This warning appears when the migration logic needs to know the concrete value of attribute X during migration, but instead such value includes an expression that's can't be resolved, so the actual value can not be determined, and the attribute is discarded. It happens in several cases, for instance:
 - cluster-connection forward-when-no-consumers. This boolean attribute has been replaced by the message-load-balancing-type attribute (which is an enum of OFF, STRICT, ON_DEMAND)
 - broadcast-group and discovery-group's jgroups-stack and jgroups-channel attributes. They reference other resources and we no longer accept expressions for them.
- Can not migrate attribute X from resource Y. This attribute is not supported by the new messaging-activemq subsystem.
Some attributes are no longer supported in the new messaging-activemq subsystem and are simply discarded:
 - hornetq-server's failback-delay
 - http-connector's use-nio attribute
 - http-acceptor's use-nio attribute
 - remote-connector's use-nio attribute
 - remote-acceptor's use-nio attribute



XML Configuration

The XML configuration has changed significantly with the new messaging-activemq subsystem to provide a XML scheme more consistent with other WildFly subsystems.

It is not advised to change the XML configuration of the legacy messaging subsystem to conform to the new messaging-activemq subsystem. Instead, invoke the legacy subsystem `migrate` operation. This operation will write the XML configuration of the new **messaging-activemq** subsystem as a part of its execution.

Messaging Interceptors

Messaging Interceptors are significantly different in EAP 7, requiring both code and configuration changes by the user. In concrete the interceptor base Java class is now

org.apache.artemis.activemq.api.core.interceptor.Interceptor, and the user interceptor implementation classes may now be loaded by any server module. Note that prior to EAP 7 the interceptor classes could only be installed by adding these to the HornetQ module, thus requiring the user to change such module XML descriptor, its **module.xml**.

With respect to the server XML configuration, the user must now specify the module to load its interceptors in the new **messaging-activemq** subsystem XML config, e.g:

```
<subsystem xmlns="urn:jboss:domain:messaging-activemq:1.0">
  <server name="default">
    ...
    <incoming-interceptors>
      <class name="org.foo.incoming.myInterceptor" module="org.foo" />
      <class name="org.bar.incoming.myOtherInterceptor" module="org.bar" />
    </incoming-interceptors>
    <outgoing-interceptors>
      <class name="org.foo.outgoing.myInterceptor" module="org.foo" />
      <class name="org.bar.outgoing.myOtherInterceptor" module="org.bar" />
    </outgoing-interceptors>
  </server>
</subsystem>
```

JMS Destinations

In previous releases, JMS destination queues were configured in the `<jms-destinations>` element under the `hornetq-server` section of the **messaging** subsystem.

```
<jms-destinations>
<jms-queue name="testQueue">
<entry name="queue/test" />
<entry name="java:jboss/exported/jms/queue/test" />
</jms-queue>
</jms-destinations>
```

In WildFly, the JMS destination queue is configured in the default server of the messaging-activemq subsystem.

```
<jms-queue name="testQueue" entries="queue/test java:jboss/exported/jms/queue/test" />
```



Messaging Logging

The prefix of messaging log messages in WildFly is **WFLYMSGAMQ**, instead of **WFLYMSG**.

Messaging Data

The location of the messaging data has been changed in the new messaging-activemq subsystem:

- messagingbindings/ -> activemq/bindings/
- messagingjournal/ -> activemq/journal/
- messaginglargemessages/ -> activemq/largemessages/
- messagingpaging/ -> activemq/paging/

To migrate legacy messaging data, you will have to export the directories used by the legacy messaging subsystem and import them into the new subsystem's server by using its `import-journal` operation:

```
/subsystem=messaging-activemq/server=default:import-journal(file=<path to XML dump>)
```

The XML dump is a XML file generated by HornetQ `XmlDataExporter` util class.

24.4 Application Migration

Before you migrate your application, you should be aware that some features that were available in previous releases are now deprecated or missing.



24.4.1 EJBs

CMP Entity EJBs

Container-Managed Persistence entity beans support is optional in Java EE 7, and WildFly does not provide support for these.

CMP entity beans are defined in the **ejb-jar.xml** descriptor, in concrete an entity bean is CMP only if the **<entity/>**'s child element named **persistence-type** is included and has a value of **Container**. An example:

```
<?xml version="1.1" encoding="UTF-8"?>
<ejb-jar 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
http://java.sun.com/xml/ns/javaee/ ejb-jar_3_1.xsd"
  version="3.1">
  <enterprise-beans>
    <entity>
      <ejb-name>SimpleBMP</ejb-name>
      <local-home>org.jboss.as.test.integration.ejb.entity.bmp.BMPLocalHome</local-home>
      <local>org.jboss.as.test.integration.ejb.entity.bmp.BMPLocalInterface</local>
      <ejb-class>org.jboss.as.test.integration.ejb.entity.bmp.SimpleBMPBean</ejb-class>
      <persistence-type>Container</persistence-type>
      <prim-key-class>java.lang.Integer</prim-key-class>
      <reentrant>true</reentrant>
    </entity>
  </enterprise-beans>
</ejb-jar>
```

CMP entity beans should be replaced by JPA entities.



EJB Client

Default Remote Connection Port

The default remote connection port has changed from '4447' to '8080'.

In JBoss AS7, the `jboss-ejb-client.properties` file looked similar to the following:

```
remote.connectionprovider.create.options.org.xnio.Options.SSL_ENABLED=false
remote.connections=default
remote.connection.default.host=localhost
remote.connection.default.port=4447
remote.connection.default.connect.options.org.xnio.Options.SASL_POLICY_NOANONYMOUS=false
```

In WildFly, the properties file looks like this:

```
remote.connectionprovider.create.options.org.xnio.Options.SSL_ENABLED=false
remote.connections=default
remote.connection.default.host=localhost
remote.connection.default.port=8080
remote.connection.default.connect.options.org.xnio.Options.SASL_POLICY_NOANONYMOUS=false
```

Default Connector

In WildFly, the default connector has changed from "remoting" to "http-remoting". This change impacts clients that use libraries from one release of JBoss and to connect to server in a different release.

- If a client application uses the EJB client library from JBoss AS 7 and wants to connect to WildFly 10 server, the server must be configured to expose a remoting connector on a port other than "8080". The client must then connect using that newly configured connector.
- A client application that uses the EJB client library from WildFly 10 and wants to connect to a JBoss AS 7 server must be aware that the server instance does not use the http-remoting connector and instead uses a remoting connector. This is achieved by defining a new client-side connection property.

```
remote.connection.default.protocol=remote
```

External applications using JNDI, to remotely lookup up EJBs in a WildFly 10 server, may also need to be migrated, please refer to [Remote JNDI Clients](#) section for further information.



24.4.2 JMS

Proprietary JMS Resource Definitions

The proprietary XML descriptors, previously used to setup JMS resources, are deprecated in WildFly. Java EE 7 (section EE.5.18) standardised such functionality.

The deprecated descriptors are files bundled in the application package, which name ends with **-jms.xml**. Their namespace has been changed to **urn:jboss:messaging-activemq-deployment:1.0**.

External JMS Clients

JMS Resources are remotely looked up using JNDI, and looking up resources in a WildFly 10 server may require changes in the application code, please refer to [Remote JNDI Clients](#) section for further information.



24.4.3 JPA (and Hibernate)

Applications That Plan to Use Hibernate ORM 5.0

WildFly ships with Hibernate ORM 5.0 and those libraries are implicitly added to the application classpath when a persistence.xml is detected during deployment. If your application uses JPA, it will default to using the Hibernate ORM 5.0 libraries.

Hibernate ORM 5.0 introduces:

- Redesigned metamodel - Complete replacement for the current org.hibernate.mapping code
- Query parser - Improved query parser based on Antlr 3/4
- Multi-tenancy improvements - Discriminator-based multi-tenancy
- Follow-on fetches - Two-phase loading via LoadPlans/EntityGraphs

Applications that currently use Hibernate ORM 4.0 - 4.3

If your application needs second-level cache enabled, you should migrate to Hibernate ORM 5.0, which is integrated with Infinispan 8.0. Applications written with Hibernate ORM 4.x can still use Hibernate 4.x if you define a custom JBoss module with Hibernate 4.x JARs and exclude the Hibernate 5 classes from your application. It is recommended that you migrate your application to use Hibernate 5.

For information about the changes implemented between Hibernate 4 and Hibernate 5, see <https://github.com/hibernate/hibernate-orm/blob/master/migration-guide.adoc>

Applications that currently use Hibernate 3

The integration classes that made it easier to use Hibernate 3 in AS 7 were removed from WildFly 10. If your application still uses Hibernate 3 libraries, it is strongly recommended that you migrate your application to use Hibernate 5 as Hibernate 3 will no longer work in WildFly without a lot of effort. If you can not migrate to Hibernate 5, you must define a custom JBoss Module for the Hibernate 3 classes and exclude the Hibernate 5 classes from your application.



24.4.4 Web Applications

JBoss Web Valves

Undertow does not support the JBoss Web Valve functionality. This can be replaced by Undertow Handlers (see <http://undertow.io/undertow-docs/undertow-docs-1.3.0/index.html#undertow-handler-authors-guide> for more).

List of valves that were provided with JBoss Web, together with a corresponding Undertow handler, is provided above, in the section on the JBoss Web subsystem.

JBoss Web Valves are specified in the proprietary **jboss-web.xml** descriptor, through **<valve />** element(s). These can be replaced using the **<http-handler />** element(s). For example:

```
<jboss-web>
  <valve>
    <class-name>org.apache.catalina.valves.RequestDumperValve</class-name>
    <module>org.jboss.as.web</module>
  </valve>
</jboss-web>
```

can be replaced by

```
<jboss-web>
  <http-handler>
    <class-name>io.undertow.server.handlers.RequestDumpingHandler</class-name>
    <module>io.undertow.core</module>
  </http-handler>
</jboss-web>
```

24.4.5 Web Services

CXF Spring Webservices

The setup of web service's endpoints and clients, through a Spring XML descriptor, driving a CXF bus creation, is no longer supported in WildFly.

Any application containing a `jbossws-cxf.xml` must migrate all functionality specified in such XML descriptor, mostly already supported by the JAX-WS specification, included in Java EE 7. It is still possible to rely on direct Apache CXF API usage, loosing the Java EE portability of the application, for instance when specific Apache CXF functionalities are needed. Please refer to the Apache CXF Integration document for further information.



RPC

JAX-RPC is an API for building Web services and clients that used remote procedure calls (RPC) and XML, which was deprecated in Java EE 6, and is no longer supported by WildFly.

JAX-RPC Web Services may be identified by the presence of the XML descriptor named `web-services.xml`, containing a `<webservice-description/>` element that includes a child element named `<jaxrpc-mapping-file/>`. An example:

```
<webservices xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://www.ibm.com/webservices/xsd/j2ee_web_services_1_1.xsd" version="1.1">
  <webservice-description>
    <webservice-description-name>HelloService</webservice-description-name>
    <wsdl-file>WEB-INF/wsdl/HelloService.wsdl</wsdl-file>
    <jaxrpc-mapping-file>WEB-INF/mapping.xml</jaxrpc-mapping-file>
    <port-component>
      <port-component-name>Hello</port-component-name>
      <wsdl-port>HelloPort</wsdl-port>

    <service-endpoint-interface>org.jboss.chap12.hello.Hello</service-endpoint-interface>
      <service-impl-bean>
        <servlet-link>HelloWorldServlet</servlet-link>
      </service-impl-bean>
    </port-component>
  </webservice-description>
</webservices>
```

Applications using JAX-RPC should be migrated to use JAX-WS, the current Java EE standard web service framework.

RS 2.0

JSR 339: JAX-RS 2.0: The Java API for RESTful Web Services specification is located here:

<https://jcp.org/en/jsr/detail?id=339>

Some changes to the `MessageBodyWriter` interface may represent a backward incompatible change with respect to JAX-RS 1.X.

Be sure to define an `@Produces` or `@Consumes` for your endpoints. Failure to do so may result in an error similar to the following.

```
org.jboss.resteasy.core.NoMessageBodyWriterFoundFailure: Could not find MessageBodyWriter for
response object of type: <OBJECT> of media type: <CONTENT_TYPE>
```



REST Client API

Some REST Client API classes and methods are deprecated, for example:

`org.jboss.resteasy.client.ClientRequest` and `org.jboss.resteasy.client.ClientResponse`. Instead, use [org.jboss.resteasy.client.jaxrs.ResteasyClient](#) and `javax.ws.rs.core.Response`. See the `resteasy-jaxrs-client quickstart` for an example of an external JAX-RS RestEasy client that interacts with a JAX-RS Web service.

24.4.6 Application Clustering

HA Singleton

JBoss AS7 introduced singleton services - a mechanism for installing an service such that it would only start on one node in the cluster at a time, a HA Singleton. Such mechanism required usage of a private JBoss EAP Clustering API, designed around the class `org.jboss.as.clustering.singleton.SingletonService`, and was documented in detail at

https://access.redhat.com/documentation/en-US/JBoss_Enterprise_Application_Platform/6.4/html/Developme, and while not difficult to implement, the installation process suffered from a couple shortcomings:

- Installing multiple singleton services within a single deployment caused the deployer to hang.
- Installing a singleton service required the user to specify several private module dependencies in `/META-INF/MANIFEST.MF`

WildFly 10 introduces a new public API for building such services, which significantly simplifies the process, and solves the issues found in the legacy solution. The JBoss EAP 7 Quickstart application named `cluster-ha-singleton` examples a HA Singleton implementation using the new API, and may be found at <https://github.com/jboss-developer/jboss-eap-quickstarts/tree/7.0.x-develop/cluster-ha-singleton>.

FIXME: community URLs instead

Stateful Session EJB Clustering

WildFly 10 no longer requires Stateful Session EJBs to use the `org.jboss.ejb3.annotation.Clustered` annotation to enable clustering behaviour. By default, if the server is started using an HA profile, the state of your SFSBs will be replicated automatically. Disabling this behaviour is achievable on a per-EJB basis, by annotating your bean using `@Stateful(passivationCapable=false)`, which is new to the EJB 3.2 specification; or globally through the configuration of the EJB3 subsystem, in the server configuration.

Note that the `@Clustered` annotation, if used by an application, is simply ignored, the application deployment will not fail.

Web Session Clustering

WildFly 10 introduces a new web session clustering implementation, replacing the one found in AS7, which has been around for ages (since JBoss AS 3.2!), and was tightly coupled to the legacy JBoss Web subsystem source code. The most relevant changes in the new implementation are:



- Introduction of a proper session manager SPI, and an Infinispan implementation of it, decoupled from the web subsystem implementation
- Sessions are implemented as a facade over one or more cache entries, which means that the container's session manager itself does not retain a separate reference to each HttpSession
- Pessimistic locking of cache entries effectively ensures that only a single client on a single node ever accesses a given session at any given time
- Usage of cache entry grouping, instead of atomic maps, to ensure that multiple cache entries belonging to the same session are co-located.
- Session operations within a request only ever use a single batch/transaction. This results in fewer RPCs per request.
- Support for write-through cache stores, as well as passivation-only cache stores.

With respect to applications, the new web session clustering implementation deprecates/reinterprets much of the related configuration, which is included in JBoss's proprietary web application XML descriptor, **jboss-web.xml**:

- **<max-active-sessions/>**
Previously, session creation would fail if an additional session would cause the number of active sessions to exceed the value specified by **<max-active-sessions/>**.
In the new implementation, **<max-active-sessions/>** is used to enable session passivation. If session creation would cause the number of active sessions to exceed **<max-active-sessions/>**, then the oldest session known to the session manager will passivate to make room for the new session.
- **<passivation-config/>**
This configuration element and its sub-elements are no longer used in WildFly.
- **<use-session-passivation/>**
Previously, passivation was enabled via this attribute, yet in the new implementation, passivation is enabled by specifying a non-negative value for **<max-active-sessions/>**.
- **<passivation-min-idle-time/>**
Previously, sessions needed to be active for at least a specific amount of time before becoming a candidate for passivation. This could cause session creation to fail, even when passivation was enabled.
The new implementation does not support this logic and thus avoids this DoS vulnerability.
- **<passivation-max-idle-time/>**
Previously, a session would be passivated after it was idle for a specific amount of time.
The new implementation does not support eager passivation - only lazy passivation. Sessions are only passivated when necessary to comply with **<max-active-sessions/>**.
- **<replication-config/>**
The new implementation deprecates a number of sub-elements:



- **<replication-trigger/>**

Previously, session attributes could be treated as either mutable or immutable depending on the values specified by **<replication-trigger/>**:

- SET treated all attributes as immutable, requiring a separate `HttpSession.setAttribute(...)` to indicate that the value changed.
- SET_AND_GET treated all session attributes as mutable.
- SET_AND_NON_PRIMITIVE_GET recognised a small set of types (i.e. strings and boxed primitives) as immutable, and assumed that any other attribute was mutable.

The new implementation replaces this configuration option with a single, robust strategy.

Session attributes are assumed to be mutable unless one of the following is true:

- The value is a known immutable value:
 - null
 - `java.util.Collections.EMPTY_LIST`, `EMPTY_MAP`, `EMPTY_SET`
 - The value type is or implements a known immutable type:
 - `Boolean`, `Byte`, `Character`, `Double`, `Float`, `Integer`, `Long`, `Short`
 - `java.lang.Enum`, `StackTraceElement`, `String`
 - `java.io.File`, `java.nio.file.Path`
 - `java.math.BigDecimal`, `BigInteger`, `MathContext`
 - `java.net.InetAddress`, `InetSocketAddress`, `URI`, `URL`
 - `java.security.Permission`
 - `java.util.Currency`, `Locale`, `TimeZone`, `UUID`
 - The value type is annotated with `@org.wildfly.clustering.web.annotation.Immutable`
- **<use-jk/>**
- Previously, the instance-id of the node handling a given request was appended to the `jsessionid` (for use by load balancers such as `mod_jk`, `mod_proxy_balancer`, `mod_cluster`, etc.) depending on the value specified for **<use-jk/>**. In the new implementation, the instance-id, if defined, is always appended to the `jsessionid`.

- **<max-unreplicated-interval/>**

Previously, this configuration option was an optimization that would prevent the replicate of a session's timestamp if no session attribute was changed. While this sounds nice, in practice it doesn't prevent any RPCs, since session access requires cache transaction RPCs regardless of whether any session attributes changed. In the new implementation, the timestamp of a session is replicated on every request. This prevents stale session meta data following failover.

- **<snapshot-mode/>**

Previously, one could configure **<snapshot-mode/>** as `INSTANT` or `INTERVAL`. Infinispan's replication queue renders this configuration option obsolete.

- **<snapshot-interval/>**

Only relevant for **<snapshot-mode>INTERVAL</snapshot-mode>**. See above.

- **<session-notification-policy/>**

Previously, the value defined by this attribute defined a policy for triggering session events. In the new implementation, this behaviour is spec-driven and not configurable.



24.4.7 Other Specifications and Frameworks

Remote JNDI Clients

WildFly 10's default JNDI Provider URL has changed, which means that external applications, using JNDI to lookup remote resources, for instance an EJB or a JMS Queue, may need to change the value for the JNDI **InitialContext** environment's property named **java.naming.provider.url**. The default URL scheme is now **http-remoting**, and the default URL port is now **8080**.

As an example, considering the application server host is **localhost**, then clients previously accessing JBoss AS7 would use

```
java.naming.factory.initial=org.jboss.naming.remote.client.InitialContextFactory
java.naming.provider.url=remote://localhost:4447
```

while clients now accessing WildFly should use instead

```
java.naming.factory.initial=org.jboss.naming.remote.client.InitialContextFactory
java.naming.provider.url=http-remoting://localhost:8080
```

88

The specification which aimed to standardise deployment tasks got very little adoption, due to much more "feature rich" proprietary solutions already included in every vendor application server. It was no surprise that JSR-88 support was dropped from Java EE 7, and WildFly followed that and dropped support too.

A JSR-88 deployment plan is identified by a XML descriptor named **deployment-plan.xml**, bundled in a zip/jar archive.

Module Dependencies

Applications defining dependencies to WildFly modules, through the application's package MANIFEST.MF or jboss-deployment-structure.xml, may be referencing missing modules. When migrating an application, relying on such functionality, the presence of the referenced modules should be validated in advance.




25 How do I migrate my application to WildFly from other application servers

25.1 Choose from the list below:

- [How do I migrate my application from WebLogic to WildFly](#)
- [How do I migrate my application from WebSphere to WildFly](#)

25.2 How do I migrate my application from WebLogic to WildFly

The purpose of this guide is to document the application changes that are needed to successfully run and deploy WebLogic applications on WildFly.

 Feel free to add content in any way you prefer. You do not need to follow the template below. This is a work in progress.

- [Introduction](#)
 - [About this Guide](#)
 -

25.2.1 Introduction


About this Guide

The purpose of this document is to guide you through the planning process and migration of fairly simple and standard Oracle WebLogic applications to WildFly. ○



25.3 How do I migrate my application from WebSphere to WildFly

The purpose of this guide is to document the application changes that are needed to successfully run and deploy WebLogic applications on WildFly.

 Feel free to add content in any way you prefer. You do not need to follow the template below. This is a work in progress.

- [Introduction](#)
 - [About this Guide](#)

25.3.1 Introduction

About this Guide

The purpose of this document is to guide you through the planning process and migration of fairly simple and standard Oracle WebLogic applications to WildFly.



26 Implicit module dependencies for deployments

As explained in the [Class Loading in WildFly](#) article, WildFly 8 is based on module classloading. A class within a module B isn't visible to a class within a module A, unless module B adds a dependency on module A. Module dependencies can be explicitly (as explained in that classloading article) or can be "implicit". This article will explain what implicit module dependencies mean and how, when and which modules are added as implicit dependencies.

26.1 What's an implicit module dependency?

Consider an application deployment which contains EJBs. EJBs typically need access to classes from the `javax.ejb.*` package and other Java EE API packages. The jars containing these packages are already shipped in WildFly and are available as "modules". The module which contains the `javax.ejb.*` classes has a specific name and so does the module which contains all the Java EE API classes. For an application to be able to use these classes, it has to add a dependency on the relevant modules. Forcing the application developers to add module dependencies like these (i.e. dependencies which can be "inferred") isn't a productive approach. Hence, whenever an application is being deployed, the deployers within the server, which are processing this deployment "implicitly" add these module dependencies to the deployment so that these classes are visible to the deployment at runtime. This way the application developer doesn't have to worry about adding them explicitly. How and when these implicit dependencies are added is explained in the next section.



26.2 How and when is an implicit module dependency added?

When a deployment is being processed by the server, it goes through a chain of "deployment processors". Each of these processors will have a way to check if the deployment meets a certain criteria and if it does, the deployment processor adds a implicit module dependency to that deployment. Let's take an example - Consider (again) an EJB3 deployment which has the following class:

MySuperDuperBean.java

```
@Stateless
public class MySuperDuperBean {
    ...
}
```

As can be seen, we have a simple @Stateless EJB. When the deployment containing this class is being processed, the EJB deployment processor will see that the deployment contains a class with the @Stateless annotation and thus identifies this as a EJB deployment. **This is just one of the several ways, various deployment processors can identify a deployment of some specific type.** The EJB deployment processor will then add an implicit dependency on the Java EE API module, so that all the Java EE API classes are visible to the deployment.

Some subsystems will always add a API classes, even if the trigger condition is not met. These are listed separately below.

In the next section, we'll list down the implicit module dependencies that are added to a deployment, by various deployers within WildFly.

26.3 Which are the implicit module dependencies?

Subsystem responsible for adding the implicit dependency	Dependencies that are always added	Dependencies that are added if a trigger condition is met
Core Server	<ul style="list-style-type: none"> • javax.api • sun.jdk • org.jboss.vfs 	



Batch Subsystem	<ul style="list-style-type: none">• javax.batch.api	
EE Subsystem	<ul style="list-style-type: none">• javaee.api	
EJB3 subsystem		<ul style="list-style-type: none">• javaee.api
JAX-RS (Resteasy) subsystem	<ul style="list-style-type: none">• javax.xml.bind.api	<ul style="list-style-type: none">• org.jboss.resteasy.resteasy-atom-provider• org.jboss.resteasy.resteasy-cdi• org.jboss.resteasy.resteasy-jaxrs• org.jboss.resteasy.resteasy-jaxb-provider• org.jboss.resteasy.resteasy-jackson-provider• org.jboss.resteasy.resteasy-jsapi• org.jboss.resteasy.resteasy-multipart-provider• org.jboss.resteasy.async-http-servlet-30
JCA subsystem	<ul style="list-style-type: none">• javax.resource.api	<ul style="list-style-type: none">• javax.jms.api• javax.validation.api• org.jboss.logging• org.jboss.ironjacamar.api• org.jboss.ironjacamar.impl• org.hibernate.validator
JPA (Hibernate) subsystem	<ul style="list-style-type: none">• javax.persistence.api	<ul style="list-style-type: none">• javaee.api• org.jboss.as.jpa• org.hibernate



Logging Subsystem	<ul style="list-style-type: none">• org.jboss.logging• org.apache.commons.logging• org.apache.log4j• org.slf4j• org.jboss.logging.jul-to-slf4j-stub	
SAR Subsystem		<ul style="list-style-type: none">• org.jboss.logging• org.jboss.modules
Security Subsystem	<ul style="list-style-type: none">• org.picketbox	
Web Subsystem		<ul style="list-style-type: none">• javax.ee.api• com.sun.jsf-impl• org.hibernate.validator• org.jboss.as.web• org.jboss.logging
Web Services Subsystem	<ul style="list-style-type: none">• org.jboss.ws.api• org.jboss.ws.spi	
Weld (CDI) Subsystem		<ul style="list-style-type: none">• javax.persistence.api• javax.ee.api• org.javassist• org.jboss.interceptor• org.jboss.as.weld• org.jboss.logging• org.jboss.weld.core• org.jboss.weld.api• org.jboss.weld.spi



27 RS Reference Guide


This page outlines the three options you have for deploying JAX-RS applications in WildFly 8. These three methods are specified in the JAX-RS 1.1 specification in section 2.3.2.

27.1 Subclassing `javax.ws.rs.core.Application` and using `@ApplicationPath`

This is the easiest way and does not require any xml configuration. Simply include a subclass of `javax.ws.rs.core.Application` in your application, and annotate it with the path that you want your JAX-RS classes to be available. For example:

```
@ApplicationPath("/mypath")
public class MyApplication extends Application {
}
```

This will make your JAX-RS resources available under `/mywebappcontext/mypath`.

 Note that that the path is `/mypath` not `/mypath/*`




27.2 Subclassing `javax.ws.rs.core.Application` and using `web.xml`

If you do not wish to use `@ApplicationPath` but still need to subclass `Application` you can set up the JAX-RS mapping in `web.xml`:

```
public class MyApplication extends Application {  
}
```

```
<servlet-mapping>  
  <servlet-name>com.acme.MyApplication</servlet-name>  
  <url-pattern>/hello/*</url-pattern>  
</servlet-mapping>
```

This will make your JAX-RS resources available under `/mywebappcontext/hello`.


 You can also use this approach to override an application path set with the `@ApplicationPath` annotation.

27.3 Using `web.xml`

If you don't want to subclass `Application` you can set the JAX-RS mapping in `web.xml` as follows:

```
<servlet-mapping>  
  <servlet-name>javax.ws.rs.core.Application</servlet-name>  
  <url-pattern>/hello/*</url-pattern>  
</servlet-mapping>
```

This will make your JAX-RS resources available under `/mywebappcontext/hello`.

 Note that you only have to add the mapping, not the corresponding servlet. The server is responsible for adding the corresponding servlet automatically.



28 JNDI Reference

28.1 Overview

WildFly offers several mechanisms to retrieve components by name. Every WildFly instance has its own local JNDI namespace (`java:`) which is unique per JVM. The layout of this namespace is primarily governed by the Java EE specification. Applications which share the same WildFly instance can use this namespace to intercommunicate. In addition to local JNDI, a variety of mechanisms exist to access remote components.

- Client JNDI - This is a mechanism by which remote components can be accessed using the JNDI APIs, but **without network round-trips**. This approach is the most efficient, and **removes a potential single point of failure**. For this reason, it is highly recommended to use Client JNDI over traditional remote JNDI access. However, to make this possible, it does require that all names follow a strict layout, so user customizations are not possible. Currently only access to remote EJBs is supported via the `ejb:` namespace. Future revisions will likely add a JMS client JNDI namespace.
- Traditional Remote JNDI - This is a more familiar approach to EE application developers, where the client performs a remote component name lookup against a server, and a proxy/stub to the component is serialized as part of the name lookup and returned to the client. The client then invokes a method on the proxy which results in another remote network call to the underlying service. In a nutshell, traditional remote JNDI involves two calls to invoke an EE component, whereas Client JNDI requires one. It does however allow for customized names, and for a centralised directory for multiple application servers. This centralized directory is, however, *a single point of failure*.
- EE Application Client / Server-To-Server Delegation - This approach is where local names are bound as an *alias* to a remote name using one of the above mechanisms. This is useful in that it allows applications to only ever reference standard portable Java EE names in both code and deployment descriptors. It also allows for the application to be unaware of network topology details/ This can even work with Java SE clients by using the little known EE Application Client feature. This feature allows you to run an extremely minimal AS server around your application, so that you can take advantage of certain core services such as naming and injection.

28.2 Local JNDI

The Java EE platform specification defines the following JNDI contexts:

- `java:comp` - The namespace is scoped to the current component (i.e. EJB)
- `java:module` - Scoped to the current module
- `java:app` - Scoped to the current application
- `java:global` - Scoped to the application server

In addition to the standard namespaces, WildFly also provides the following two global namespaces:



- java:jboss
- java:/

⚠ Only entries within the `java:jboss/exported` context are accessible over remote JNDI.

⚠ For web deployments `java:comp` is aliased to `java:module`, so EJB's deployed in a war do not have their own `comp` namespace.

28.2.1 Binding entries to JNDI

There are several methods that can be used to bind entries into JNDI in WildFly.

Using a deployment descriptor

For Java EE applications the recommended way is to use a [deployment descriptor](#) to create the binding. For example the following `web.xml` binds the string "Hello World" to `java:global/mystring` and the string "Hello Module" to `java:comp/env/hello` (any non absolute JNDI name is relative to `java:comp/env` context).

```
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
  version="3.1">
  <env-entry>
    <env-entry-name>java:global/mystring</env-entry-name>
    <env-entry-type>java.lang.String</env-entry-type>
    <env-entry-value>Hello World</env-entry-value>
  </env-entry>
  <env-entry>
    <env-entry-name>hello</env-entry-name>
    <env-entry-type>java.lang.String</env-entry-type>
    <env-entry-value>Hello Module</env-entry-value>
  </env-entry>
</web-app>
```

For more details, see the [Java EE Platform Specification](#).




Programmatically

Java EE Applications

Standard Java EE applications may use the standard JNDI API, included with Java SE, to bind entries in the global namespaces (the standard `java:comp`, `java:module` and `java:app` namespaces are read-only, as mandated by the Java EE Platform Specification).


```
InitialContext initialContext = new InitialContext();
initialContext.bind("java:global/a", 100);
```

 There is no need to unbind entries created programmatically, since WildFly tracks which bindings belong to a deployment, and the bindings are automatically removed when the deployment is undeployed.

WildFly Modules and Extensions

With respect to code in WildFly Modules/Extensions, which is executed out of a Java EE application context, using the standard JNDI API may result in a `UnsupportedOperationException` if the target namespace uses a `WritableServiceBasedNamingStore`. To work around that, the `bind()` invocation needs to be wrapped using WildFly proprietary APIs:

```
InitialContext initialContext = new InitialContext();
WritableServiceBasedNamingStore.pushOwner(serviceTarget);
try {
    initialContext.bind("java:global/a", 100);
} finally {
    WritableServiceBasedNamingStore.popOwner();
}
```

 The `ServiceTarget` removes the bind when uninstalled, thus using one out of the module/extension domain usage should be avoided, unless entries are removed using `unbind()`.



Naming Subsystem Configuration

It is also possible to bind to one of the three global namespaces using configuration in the naming subsystem. This can be done by either editing the `standalone.xml/domain.xml` file directly, or through the management API.

Four different types of bindings are supported:

- Simple - A primitive or `java.net.URL` entry (default is `java.lang.String`).
- Object Factory - This allows to specify the `javax.naming.spi.ObjectFactory` that is used to create the looked up value.
- External Context - An external context to federate, such as an LDAP Directory Service
- Lookup - The allows to create JNDI aliases, when this entry is looked up it will lookup the target and return the result.

An example `standalone.xml` might look like:

```
<subsystem xmlns="urn:jboss:domain:naming:2.0" >
  <bindings>
    <simple name="java:global/a" value="100" type="int" />
    <simple name="java:global/jbossDocs" value="https://docs.jboss.org" type="java.net.URL" />
    <object-factory name="java:global/b" module="com.acme" class="org.acme.MyObjectFactory" />
    <external-context name="java:global/federation/ldap/example"
class="javax.naming.directory.InitialDirContext" cache="true">
      <environment>
        <property name="java.naming.factory.initial" value="com.sun.jndi.ldap.LdapCtxFactory" />
        <property name="java.naming.provider.url" value="ldap://ldap.example.com:389" />
        <property name="java.naming.security.authentication" value="simple" />
        <property name="java.naming.security.principal" value="uid=admin,ou=system" />
        <property name="java.naming.security.credentials" value="secret" />
      </environment>
    </external-context>
    <lookup name="java:global/c" lookup="java:global/b" />
  </bindings>
</subsystem>
```

The CLI may also be used to bind an entry. As an example:

```
/subsystem=naming/binding=java\:global\mybinding:add(binding-type=simple, type=long,
value=1000)
```



WildFly's Administrator Guide includes a section describing in detail the Naming subsystem configuration.



28.2.2 Retrieving entries from JNDI

Resource Injection

For Java EE applications the recommended way to lookup a JNDI entry is to use `@Resource` injection:

```
@Resource(lookup = "java:global/mystring")
private String myString;

@Resource(name = "hello")
private String hello;

@Resource
ManagedExecutorService executor;
```

Note that `@Resource` is more than a JNDI lookup, it also binds an entry in the component's JNDI environment. The new bind JNDI name is defined by `@Resource`'s `name` attribute, which value, if unspecified, is the Java type concatenated with `/` and the field's name, for instance `java.lang.String/myString`. More, similar to when using deployment descriptors to bind JNDI entries, unless the name is an absolute JNDI name, it is considered relative to `java:comp/env`. For instance, with respect to the field named `myString` above, the `@Resource`'s `lookup` attribute instructs WildFly to lookup the value in `java:global/mystring`, bind it in `java:comp/env/java.lang.String/myString`, and then inject such value into the field.

With respect to the field named `hello`, there is no `lookup` attribute value defined, so the responsibility to provide the entry's value is delegated to the deployment descriptor. Considering that the deployment descriptor was the `web.xml` previously shown, which defines an environment entry with same `hello` name, then WildFly inject the valued defined in the deployment descriptor into the field.

The `executor` field has no attributes specified, so the bind's name would default to `java:comp/env/javax.enterprise.concurrent.ManagedExecutorService/executor`, but there is no such entry in the deployment descriptor, and when that happens it's up to WildFly to provide a default value or null, depending on the field's Java type. In this particular case WildFly would inject the default instance of a managed executor service, the value in `java:comp/DefaultManagedExecutorService`, as mandated by the EE Concurrency Utilities 1.0 Specification (JSR 236).



Standard Java SE JNDI API

Java EE applications may use, without any additional configuration needed, the standard JNDI API to lookup an entry from JNDI:

```
String myString = (String) new InitialContext().lookup("java:global/mystring");
```

or simply

```
String myString = InitialContext.doLookup("java:global/mystring");
```

28.3 Remote JNDI

WildFly supports two different types of remote JNDI. The old jnp based JNDI implementation used in JBoss AS versions prior to 7.x is no longer supported.

28.3.1 remote:

The `remote:` protocol uses the WildFly remoting protocol to lookup items from the servers local JNDI. To use it, you must have the appropriate jars on the class path, if you are maven user can be done simply by adding the following to your `pom.xml`:

```
<dependency>
  <groupId>org.wildfly</groupId>
  <artifactId>wildfly-ejb-client-bom</artifactId>
  <version>8.0.0.Final</version>
  <type>pom</type>
  <scope>compile</scope>
</dependency>
```

If you are not using maven a shaded jar that contains all required classes can be found in the `bin/client` directory of WildFly's distribution.

```
final Properties env = new Properties();
env.put(Context.INITIAL_CONTEXT_FACTORY,
org.jboss.naming.remote.client.InitialContextFactory.class.getName());
env.put(Context.PROVIDER_URL, "remote://localhost:4447");
remoteContext = new InitialContext(env);
```



28.3.2 ejb:

The `ejb:` namespace is provided by the `jboss-ejb-client` library. This protocol allows you to look up EJB's, using their application name, module name, `ejb` name and interface type.

This is a client side JNDI implementation. Instead of looking up an EJB on the server the lookup name contains enough information for the client side library to generate a proxy with the EJB information. When you invoke a method on this proxy it will use the current EJB client context to perform the invocation. If the current context does not have a connection to a server with the specified EJB deployed then an error will occur. Using this protocol it is possible to look up EJB's that do not actually exist, and no error will be thrown until the proxy is actually used. The exception to this is stateful session beans, which need to connect to a server when they are created in order to create the session bean instance on the server.

Some examples are:

```
ejb:myapp/myejbjar/MyEjbName!com.test.MyRemoteInterface
ejb:myapp/myejbjar/MyStatefulName!comp.test.MyStatefulRemoteInterface?stateful
```

The first example is a lookup of a singleton, stateless or EJB 2.x home interface. This lookup will not hit the server, instead a proxy will be generated for the remote interface specified in the name. The second example is for a stateful session bean, in this case the JNDI lookup will hit the server, in order to tell the server to create the SFSB session.

For more details on how the server connections are configured, please see [EJB invocations from a remote client using JNDI](#).


28.4 Local JNDI

The Java EE platform specification defines the following JNDI contexts:


- `java:comp` - The namespace is scoped to the current component (i.e. EJB)
- `java:module` - Scoped to the current module
- `java:app` - Scoped to the current application
- `java:global` - Scoped to the application server

In addition to the standard namespaces, WildFly also provides the following two global namespaces:

- `java:jboss`
- `java:/`

 Only entries within the `java:jboss/exported` context are accessible over remote JNDI.



 For web deployments `java:comp` is aliased to `java:module`, so EJB's deployed in a war do not have their own comp namespace.

28.4.1 Binding entries to JNDI

There are several methods that can be used to bind entries into JNDI in WildFly.

Using a deployment descriptor

For Java EE applications the recommended way is to use a [deployment descriptor](#) to create the binding. For example the following `web.xml` binds the string "Hello World" to `java:global/mystring` and the string "Hello Module" to `java:comp/env/hello` (any non absolute JNDI name is relative to `java:comp/env` context).

```
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
  version="3.1">
  <env-entry>
    <env-entry-name>java:global/mystring</env-entry-name>
    <env-entry-type>java.lang.String</env-entry-type>
    <env-entry-value>Hello World</env-entry-value>
  </env-entry>
  <env-entry>
    <env-entry-name>hello</env-entry-name>
    <env-entry-type>java.lang.String</env-entry-type>
    <env-entry-value>Hello Module</env-entry-value>
  </env-entry>
</web-app>
```

For more details, see the [Java EE Platform Specification](#).




Programmatically

Java EE Applications

Standard Java EE applications may use the standard JNDI API, included with Java SE, to bind entries in the global namespaces (the standard `java:comp`, `java:module` and `java:app` namespaces are read-only, as mandated by the Java EE Platform Specification).


```
InitialContext initialContext = new InitialContext();
initialContext.bind("java:global/a", 100);
```

 There is no need to unbind entries created programmatically, since WildFly tracks which bindings belong to a deployment, and the bindings are automatically removed when the deployment is undeployed.

WildFly Modules and Extensions

With respect to code in WildFly Modules/Extensions, which is executed out of a Java EE application context, using the standard JNDI API may result in a `UnsupportedOperationException` if the target namespace uses a `WritableServiceBasedNamingStore`. To work around that, the `bind()` invocation needs to be wrapped using WildFly proprietary APIs:

```
InitialContext initialContext = new InitialContext();
WritableServiceBasedNamingStore.pushOwner(serviceTarget);
try {
    initialContext.bind("java:global/a", 100);
} finally {
    WritableServiceBasedNamingStore.popOwner();
}
```

 The `ServiceTarget` removes the bind when uninstalled, thus using one out of the module/extension domain usage should be avoided, unless entries are removed using `unbind()`.



Naming Subsystem Configuration

It is also possible to bind to one of the three global namespaces using configuration in the naming subsystem. This can be done by either editing the `standalone.xml/domain.xml` file directly, or through the management API.

Four different types of bindings are supported:

- Simple - A primitive or `java.net.URL` entry (default is `java.lang.String`).
- Object Factory - This allows to specify the `javax.naming.spi.ObjectFactory` that is used to create the looked up value.
- External Context - An external context to federate, such as an LDAP Directory Service
- Lookup - The allows to create JNDI aliases, when this entry is looked up it will lookup the target and return the result.

An example `standalone.xml` might look like:

```
<subsystem xmlns="urn:jboss:domain:naming:2.0" >
  <bindings>
    <simple name="java:global/a" value="100" type="int" />
    <simple name="java:global/jbossDocs" value="https://docs.jboss.org" type="java.net.URL" />
    <object-factory name="java:global/b" module="com.acme" class="org.acme.MyObjectFactory" />
    <external-context name="java:global/federation/ldap/example"
class="javax.naming.directory.InitialDirContext" cache="true">
      <environment>
        <property name="java.naming.factory.initial" value="com.sun.jndi.ldap.LdapCtxFactory" />
        <property name="java.naming.provider.url" value="ldap://ldap.example.com:389" />
        <property name="java.naming.security.authentication" value="simple" />
        <property name="java.naming.security.principal" value="uid=admin,ou=system" />
        <property name="java.naming.security.credentials" value="secret" />
      </environment>
    </external-context>
    <lookup name="java:global/c" lookup="java:global/b" />
  </bindings>
</subsystem>
```

The CLI may also be used to bind an entry. As an example:

```
/subsystem=naming/binding=java\:global\mybinding:add(binding-type=simple, type=long,
value=1000)
```



WildFly's Administrator Guide includes a section describing in detail the Naming subsystem configuration.



28.4.2 Retrieving entries from JNDI

Resource Injection

For Java EE applications the recommended way to lookup a JNDI entry is to use `@Resource` injection:

```
@Resource(lookup = "java:global/mystring")
private String myString;

@Resource(name = "hello")
private String hello;

@Resource
ManagedExecutorService executor;
```

Note that `@Resource` is more than a JNDI lookup, it also binds an entry in the component's JNDI environment. The new bind JNDI name is defined by `@Resource`'s `name` attribute, which value, if unspecified, is the Java type concatenated with `/` and the field's name, for instance `java.lang.String/myString`. More, similar to when using deployment descriptors to bind JNDI entries, unless the name is an absolute JNDI name, it is considered relative to `java:comp/env`. For instance, with respect to the field named `myString` above, the `@Resource`'s `lookup` attribute instructs WildFly to lookup the value in `java:global/mystring`, bind it in `java:comp/env/java.lang.String/myString`, and then inject such value into the field.

With respect to the field named `hello`, there is no `lookup` attribute value defined, so the responsibility to provide the entry's value is delegated to the deployment descriptor. Considering that the deployment descriptor was the `web.xml` previously shown, which defines an environment entry with same `hello` name, then WildFly inject the valued defined in the deployment descriptor into the field.

The `executor` field has no attributes specified, so the bind's name would default to `java:comp/env/javax.enterprise.concurrent.ManagedExecutorService/executor`, but there is no such entry in the deployment descriptor, and when that happens it's up to WildFly to provide a default value or null, depending on the field's Java type. In this particular case WildFly would inject the default instance of a managed executor service, the value in `java:comp/DefaultManagedExecutorService`, as mandated by the EE Concurrency Utilities 1.0 Specification (JSR 236).



Standard Java SE JNDI API

Java EE applications may use, without any additional configuration needed, the standard JNDI API to lookup an entry from JNDI:

```
String myString = (String) new InitialContext().lookup("java:global/mystring");
```

or simply

```
String myString = InitialContext.doLookup("java:global/mystring");
```

28.5 Remote JNDI Reference

28.5.1 Remote JNDI

WildFly supports two different types of remote JNDI. The old jnp based JNDI implementation used in JBoss AS versions prior to 7.x is no longer supported.

remote:

The `remote:` protocol uses the WildFly remoting protocol to lookup items from the servers local JNDI. To use it, you must have the appropriate jars on the class path, if you are maven user can be done simply by adding the following to your `pom.xml`:

```
<dependency>
  <groupId>org.wildfly</groupId>
  <artifactId>wildfly-ejb-client-bom</artifactId>
  <version>8.0.0.Final</version>
  <type>pom</type>
  <scope>compile</scope>
</dependency>
```

If you are not using maven a shaded jar that contains all required classes can be found in the `bin/client` directory of WildFly's distribution.

```
final Properties env = new Properties();
env.put(Context.INITIAL_CONTEXT_FACTORY,
org.jboss.naming.remote.client.InitialContextFactory.class.getName());
env.put(Context.PROVIDER_URL, "remote://localhost:4447");
remoteContext = new InitialContext(env);
```



ejb:

The `ejb:` namespace is provided by the `jboss-ejb-client` library. This protocol allows you to look up EJB's, using their application name, module name, `ejb` name and interface type.

This is a client side JNDI implementation. Instead of looking up an EJB on the server the lookup name contains enough information for the client side library to generate a proxy with the EJB information. When you invoke a method on this proxy it will use the current EJB client context to perform the invocation. If the current context does not have a connection to a server with the specified EJB deployed then an error will occur. Using this protocol it is possible to look up EJB's that do not actually exist, and no error will be thrown until the proxy is actually used. The exception to this is stateful session beans, which need to connect to a server when they are created in order to create the session bean instance on the server.

Some examples are:

```
ejb:myapp/myejbjar/MyEjbName!com.test.MyRemoteInterface  
ejb:myapp/myejbjar/MyStatefulName!comp.test.MyStatefulRemoteInterface?stateful
```

The first example is a lookup of a singleton, stateless or EJB 2.x home interface. This lookup will not hit the server, instead a proxy will be generated for the remote interface specified in the name. The second example is for a stateful session bean, in this case the JNDI lookup will hit the server, in order to tell the server to create the SFSB session.

For more details on how the server connections are configured, please see [EJB invocations from a remote client using JNDI](#).

28.5.2 Remote JNDI Access

WildFly supports two different types of remote JNDI.




http-remoting:

The `http-remoting` protocol implementation is provided by JBoss Remote Naming project, and uses http upgrade to lookup items from the servers local JNDI. To use it, you must have the appropriate jars on the class path, if you are maven user can be done simply by adding the following to your `pom.xml` dependencies:

```
<dependency>
  <groupId>org.wildfly</groupId>
  <artifactId>wildfly-ejb-client-bom</artifactId>
  <version>8.0.0.Final</version>
  <type>pom</type>
</dependency>
```

If you are not using maven a shaded jar that contains all required classes can be found in the `bin/client` directory of WildFly's distribution.

```
final Properties env = new Properties();
env.put(Context.INITIAL_CONTEXT_FACTORY,
"org.jboss.naming.remote.client.InitialContextFactory");
env.put(Context.PROVIDER_URL, "http-remoting://localhost:8080");
// the property below is required ONLY if there is no ejb client configuration loaded (such as a
// jboss-ejb-client.properties in the class path) and the context will be used to lookup EJBs
env.put("jboss.naming.client.ejb.context", true);
InitialContext remoteContext = new InitialContext(env);
RemoteCalculator ejb = (RemoteCalculator)
remoteContext.lookup("wildfly-http-remoting-ejb/CalculatorBean!"
+ RemoteCalculator.class.getName());
```

 The `http-remoting` client assumes JNDI names in remote lookups are relative to `java:jboss/exported` namespace, a lookup of an absolute JNDI name will fail.



ejb:

The `ejb:` namespace implementation is provided by the `jboss-ejb-client` library, and allows the lookup of EJB's using their application name, module name, `ejb` name and interface type. To use it, you must have the appropriate jars on the class path, if you are maven user can be done simply by adding the following to your `pom.xml` dependencies:

```
<dependency>
  <groupId>org.wildfly</groupId>
  <artifactId>wildfly-ejb-client-bom</artifactId>
  <version>8.0.0.Final</version>
  <type>pom</type>
</dependency>
```

If you are not using maven a shaded jar that contains all required classes can be found in the `bin/client` directory of WildFly's distribution.

This is a client side JNDI implementation. Instead of looking up an EJB on the server the lookup name contains enough information for the client side library to generate a proxy with the EJB information. When you invoke a method on this proxy it will use the current EJB client context to perform the invocation. If the current context does not have a connection to a server with the specified EJB deployed then an error will occur. Using this protocol it is possible to look up EJB's that do not actually exist, and no error will be thrown until the proxy is actually used. The exception to this is stateful session beans, which need to connect to a server when they are created in order to create the session bean instance on the server.

```
final Properties env = new Properties();
env.put(Context.URL_PKG_PREFIXES, "org.jboss.ejb.client.naming");
InitialContext remoteContext = new InitialContext(env);
MyRemoteInterface myRemote = (MyRemoteInterface)
remoteContext.lookup("ejb:myapp/myejbjar/MyEjbName!com.test.MyRemoteInterface");
MyStatefulRemoteInterface myStatefulRemote = (MyStatefulRemoteInterface)
remoteContext.lookup("ejb:myapp/myejbjar/MyStatefulName!comp.test.MyStatefulRemoteInterface?state");
```

The first example is a lookup of a singleton, stateless or EJB 2.x home interface. This lookup will not hit the server, instead a proxy will be generated for the remote interface specified in the name. The second example is for a stateful session bean, in this case the JNDI lookup will hit the server, in order to tell the server to create the SFSB session.



For more details on how the server connections are configured, including the **required** jboss `ejb` client setup, please see [EJB invocations from a remote client using JNDI](#).



29 JPA Reference Guide

- [Introduction](#)
- [Update your Persistence.xml for Hibernate 5.0](#)
- [Entity manager](#)
- [Application-managed entity manager](#)
- [Container-managed entity manager](#)
- [Persistence Context](#)
- [Transaction-scoped Persistence Context](#)
- [Extended Persistence Context](#)
 - [Extended Persistence Context Inheritance](#)
- [Entities](#)
- [Deployment](#)
- [Troubleshooting](#)
- [Using the Hibernate 5.x JPA persistence provider](#)
- [Hibernate ORM 3.x integration is not included](#)
- [Using the Infinispan second level cache](#)
- [Replacing the current Hibernate 5.x jars with a newer version](#)
- [Using Hibernate Search](#)
- [Packaging the Hibernate JPA persistence provider with your application](#)
- [Using OpenJPA](#)
- [Using EclipseLink](#)
- [Using DataNucleus](#)
- [Native Hibernate use](#)
- [Injection of Hibernate Session and SessionFactory](#)
- [Injection of Hibernate Session and SessionFactory](#)
- [Hibernate properties](#)
- [Persistence unit properties](#)
- [Determine the persistence provider module](#)
- [Binding EntityManagerFactory/EntityManager to JNDI](#)
- [Community](#)
 - [People who have contributed to the WildFly JPA layer:](#)



29.1 Introduction

The WildFly JPA subsystem implements the JPA 2.1 container-managed requirements. Deploys the persistence unit definitions, the persistence unit/context annotations and persistence unit/context references in the deployment descriptor. JPA Applications use the Hibernate (version 5) persistence provider, which is included with WildFly. The JPA subsystem uses the standard SPI (`javax.persistence.spi.PersistenceProvider`) to access the Hibernate persistence provider and some additional extensions as well.

During application deployment, JPA use is detected (e.g. `persistence.xml` or `@PersistenceContext/Unit` annotations) and injects Hibernate dependencies into the application deployment. This makes it easy to deploy JPA applications.

In the remainder of this documentation, "entity manager" refers to an instance of the `javax.persistence.EntityManager` class. [Javadoc for the JPA interfaces](#) and [JPA 2.1 specification](#).

The index of the Hibernate documentation is at <http://hibernate.org/orm/documentation/5.0/>.

29.2 Update your Persistence.xml for Hibernate 5.0

The persistence provider class name in Hibernate 4.3.0 (and greater) is `org.hibernate.jpa.HibernatePersistenceProvider`.

Instead of specifying:

```
<provider>org.hibernate.ejb.HibernatePersistence</provider>
```

Switch to:

```
<provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
```

Or remove the persistence provider class name from your `persistence.xml` (so the default provider will be used).

29.3 Entity manager

The entity manager is similar to the Hibernate Session class; applications use it to create/read/update/delete data (and related operations). Applications can use application-managed or container-managed entity managers. Keep in mind that the entity manager is not expected to be thread safe (don't inject it into a servlet class variable which is visible to multiple threads).



29.4 Application-managed entity manager

Application-managed entity managers provide direct access to the underlying persistence provider (`org.hibernate.ejb.HibernatePersistence`). The scope of the application-managed entity manager is from when the application creates it and lasts until the app closes it. Use the `@PersistenceUnit` annotation to inject a persistence unit into a `javax.persistence.EntityManagerFactory`. The `EntityManagerFactory` can return an application-managed entity manager.

29.5 Container-managed entity manager

Container-managed entity managers auto-magically manage the underlying persistence provider for the application. Container-managed entity managers may use transaction-scoped persistence contexts or extended persistence contexts. The container-managed entity manager will create instances of the underlying persistence provider as needed. Every time that a new underlying persistence provider (`org.hibernate.ejb.HibernatePersistence`) instance is created, a new persistence context is also created (as an implementation detail of the underlying persistence provider).

29.6 Persistence Context

The JPA persistence context contains the entities managed by the persistence provider. The persistence context acts like a first level (transactional) cache for interacting with the datasource. Loaded entities are placed into the persistence context before being returned to the application. Entities changes are also placed into the persistence context (to be saved in the database when the transaction commits).



29.7 Transaction-scoped Persistence Context

The transaction-scoped persistence context coordinates with the (active) JTA transaction. When the transaction commits, the persistence context is flushed to the datasource (entity objects are detached but may still be referenced by application code). All entity changes that are expected to be saved to the datasource, must be made during a transaction. Entities read outside of a transaction will be detached when the entity manager invocation completes. Example transaction-scoped persistence context is below.

```
@Stateful // will use container managed transactions
public class CustomerManager {
    @PersistenceContext(unitName = "customerPU") // default type is
    PersistenceContextType.TRANSACTION
    EntityManager em;
    public customer createCustomer(String name, String address) {
        Customer customer = new Customer(name, address);
        em.persist(customer); // persist new Customer when JTA transaction completes (when method
ends).
        // internally:
        // 1. Look for existing "customerPU" persistence context in active
JTA transaction and use if found.
        // 2. Else create new "customerPU" persistence context (e.g.
instance of org.hibernate.ejb.HibernatePersistence)
        // and put in current active JTA transaction.
        return customer; // return Customer entity (will be detached from the persistence
context when caller gets control)
    } // Transaction.commit will be called, Customer entity will be persisted to the database and
"customerPU" persistence context closed
}
```

29.8 Extended Persistence Context

The (ee container managed) extended persistence context can span multiple transactions and allows data modifications to be queued up (like a shopping cart), without an active JTA transaction (to be applied during the next JTA TX). The Container-managed extended persistence context can only be injected into a stateful session bean.

```
@PersistenceContext(type = PersistenceContextType.EXTENDED, unitName = "inventoryPU")
EntityManager em;
```



29.8.1 Extended Persistence Context Inheritance

JPA 2.0 specification section 7.6.2.1

If a stateful session bean instantiates a stateful session bean (executing in the same EJB container instance) which also has such an extended persistence context, the extended persistence context of the first stateful session bean is inherited by the second stateful session bean and bound to it, and this rule recursively applies—independently of whether transactions are active or not at the point of the creation of the stateful session beans.

By default, the current stateful session bean being created, will (**deeply**) inherit the extended persistence context from any stateful session bean executing in the current Java thread. The **deep** inheritance of extended persistence context includes walking multiple levels up the stateful bean call stack (inheriting from parent beans). The **deep** inheritance of extended persistence context includes sibling beans. For example, parentA references child beans beanBwithXPC & beanCwithXPC. Even though parentA doesn't have an extended persistence context, beanBwithXPC & beanCwithXPC will share the same extended persistence context.

Some other EE application servers, use **shallow** inheritance, where stateful session bean only inherit from the parent stateful session bean (if there is a parent bean). Sibling beans do not share the same extended persistence context unless their (common) parent bean also has the same extended persistence context.

Applications can include a (top-level) **jboss-all.xml** deployment descriptor that specifies either the (default) **DEEP** extended persistence context inheritance or **SHALLOW**.

The WF/docs/schema/jboss-jpa_1_0.xsd describes the **jboss-jpa** deployment descriptor that may be included in the **jboss-all.xml**. Below is an example of using **SHALLOW** extended persistence context inheritance:

```
<jboss>
  <jboss-jpa xmlns="http://www.jboss.com/xml/ns/javaee">
    <extended-persistence inheritance="SHALLOW"/>
  </jboss-jpa>
</jboss>
```

Below is an example of using **DEEP** extended persistence inheritance:

```
<jboss>
  <jboss-jpa xmlns="http://www.jboss.com/xml/ns/javaee">
    <extended-persistence inheritance="DEEP"/>
  </jboss-jpa>
</jboss>
```



The AS console/cli can change the **default** extended persistence context setting (DEEP or SHALLOW). The following cli commands will read the current JPA settings and enable SHALLOW extended persistence context inheritance for applications that do not include the **jboss-jpa** deployment descriptor:

```
./jboss-cli.sh
cd subsystem=jpa
:read-resource
:write-attribute(name=default-extended-persistence-inheritance,value="SHALLOW")
```

29.9 Entities

JPA allows use of your (pojo) plain old Java class to represent a database table row.

```
@PersistenceContext EntityManager em;
Integer bomPk = getIndexKeyValue();
BillOfMaterials bom = em.find(BillOfMaterials.class, bomPk); // read existing table row into
BillOfMaterials class

BillOfMaterials createdBom = new BillOfMaterials("..."); // create new entity
em.persist(createdBom); // createdBom is now managed and will be saved to database when the
current JTA transaction completes
```

The entity lifecycle is managed by the underlying persistence provider.

- **New (transient):** an entity is new if it has just been instantiated using the new operator, and it is not associated with a persistence context. It has no persistent representation in the database and no identifier value has been assigned.
- **Managed (persistent):** a managed entity instance is an instance with a persistent identity that is currently associated with a persistence context.
- **Detached:** the entity instance is an instance with a persistent identity that is no longer associated with a persistence context, usually because the persistence context was closed or the instance was evicted from the context.
- **Removed:** a removed entity instance is an instance with a persistent identity, associated with a persistence context, but scheduled for removal from the database.



29.10 Deployment

The persistence.xml contains the persistence unit configuration (e.g. datasource name) and as described in the JPA 2.0 spec (section 8.2), the jar file or directory whose META-INF directory contains the persistence.xml file is termed the root of the persistence unit. In Java EE environments, the root of a persistence unit must be one of the following (quoted directly from the JPA 2.0 specification):

"

- an EJB-JAR file
- the WEB-INF/classes directory of a WAR file
- a jar file in the WEB-INF/lib directory of a WAR file
- a jar file in the EAR library directory
- an application client jar file

The persistence.xml can specify either a JTA datasource or a non-JTA datasource. The JTA datasource is expected to be used within the EE environment (even when reading data without an active transaction). If a datasource is not specified, the default-datasource will instead be used (must be configured).

NOTE: Java Persistence 1.0 supported use of a jar file in the root of the EAR as the root of a persistence unit. This use is no longer supported. Portable applications should use the EAR library directory for this case instead.

"

Question: Can you have a EAR/META-INF/persistence.xml?

Answer: No, the above may deploy but it could include other archives also in the EAR, so you may have deployment issues for other reasons. Better to put the persistence.xml in an EAR/lib/somePuJar.jar.

29.11 Troubleshooting

The **org.jboss.as.jpa** logging can be enabled to get the following information:

- INFO - when persistence.xml has been parsed, starting of persistence unit service (per deployed persistence.xml), stopping of persistence unit service
- DEBUG - informs about entity managers being injected, creating/reusing transaction scoped entity manager for active transaction
- TRACE - shows how long each entity manager operation took in milliseconds, application searches for a persistence unit, parsing of persistence.xml

To enable TRACE, open the as/standalone/configuration/standalone.xml (or as/domain/configuration/domain.xml) file. Search for **<subsystem xmlns="urn:jboss:domain:logging:1.0">** and add the **org.jboss.as.jpa** category. You need to change the console-handler level from **INFO** to **TRACE**.



```
<subsystem xmlns="urn:jboss:domain:logging:1.0">
  <console-handler name="CONSOLE">
    <level name="TRACE" />
    ...
  </console-handler>

  </periodic-rotating-file-handler>
  <logger category="com.arjuna">
    <level name="WARN" />
  </logger>

  <logger category="org.jboss.as.jpa">
    <level name="TRACE" />
  </logger>

  <logger category="org.apache.tomcat.util.modeler">
    <level name="WARN" />
  </logger>
  ...
</subsystem>
```

To see what is going on at the JDBC level, enable **jboss.jdbc.spy** TRACE and add `spy="true"` to the `datasource`.

```
<datasource jndi-name="java:jboss/datasources/..." pool-name="..." enabled="true" spy="true">
  <logger category="jboss.jdbc.spy">
    <level name="TRACE" />
  </logger>
</datasource>
```

To troubleshoot issues with the Hibernate second level cache, try enabling trace for **org.hibernate.SQL + org.hibernate.cache.infinispan + org.infinispan:**



```
<subsystem xmlns="urn:jboss:domain:logging:1.0">
  <console-handler name="CONSOLE">
    <level name="TRACE" />
    ...
  </console-handler>

  </periodic-rotating-file-handler>
  <logger category="com.arjuna">
    <level name="WARN" />
  </logger>

  <logger category="org.hibernate.SQL">
    <level name="TRACE" />
  </logger>

  <logger category="org.hibernate">
    <level name="TRACE" />
  </logger>
  <logger category="org.infinispan">
    <level name="TRACE" />
  </logger>

  <logger category="org.apache.tomcat.util.modeler">
    <level name="WARN" />
  </logger>
  ...
</subsystem>
```

29.12 Using the Hibernate 5.x JPA persistence provider

Hibernate 5.x is packaged with WildFly and is the default persistence provider.

29.13 Hibernate ORM 3.x integration is not included

The Hibernate 3.x integration is removed from WildFly, please use a newer version of Hibernate.

29.14 Using the Infinispan second level cache

To enable the second level cache with Hibernate 5.x, just set the **hibernate.cache.use_second_level_cache** property to true, as is done in the following example (also set the [shared-cache-mode](#) accordingly). By default the application server uses Infinispan as the cache provider for **JPA applications**, so you don't need specify anything on top of that. The Infinispan version that is included in WildFly is expected to work with the Hibernate version that is included with WildFly. Example persistence.xml settings:



```
<?xml version="1.0" encoding="UTF-8"?><persistence
xmlns="http://java.sun.com/xml/ns/persistence" version="1.0">
<persistence-unit name="2lc_example_pu">
  <description>example of enabling the second level cache.</description>
  <jta-data-source>java:jboss/datasources/mydatasource</jta-data-source>
  <shared-cache-mode>ENABLE_SELECTIVE</shared-cache-mode>
  <properties>
    <property name="hibernate.cache.use_second_level_cache" value="true"/>
  </properties>
</persistence-unit>
</persistence>
```

Here is an example of enabling the second level cache for a Hibernate native API hibernate.cfg.xml file:

```
<property name="hibernate.cache.region.factory_class"
value="org.jboss.as.jpa.hibernate5.infinispan.InfinispanRegionFactory"/>
<property name="hibernate.cache.infinispan.cachemanager"
value="java:jboss/infinispan/container/hibernate"/>
<property name="hibernate.transaction.manager_lookup_class"
value="org.hibernate.transaction.JBossTransactionManagerLookup"/>
<property name="hibernate.cache.use_second_level_cache" value="true"/>
```

The Hibernate native API application will also need a MANIFEST.MF:

```
Dependencies: org.infinispan,org.hibernate
```

[Infinispan Hibernate/JPA second level cache provider documentation](#) contains advanced configuration information but you should bear in mind that when Hibernate runs within WildFly 8, some of those configuration options, such as region factory, are not needed. Moreover, the application server providers you with option of selecting a different cache container for Infinispan via **hibernate.cache.infinispan.container** persistence property. To reiterate, this property is not mandatory and a default container is already deployed for by the application server to host the second level cache.

Here is an example of what the Hibernate cache settings may currently be in your standalone.xml:

```
<cache-container name="hibernate" default-cache="local-query" module="org.hibernate.infinispan">
  <local-cache name="entity">
    <transaction mode="NON_XA"/>
    <eviction strategy="LRU" max-entries="10000"/>
    <expiration max-idle="100000"/>
  </local-cache>
  <local-cache name="local-query">
    <eviction strategy="LRU" max-entries="10000"/>
    <expiration max-idle="100000"/>
  </local-cache>
  <local-cache name="timestamps"/>
</cache-container>
```



Below is an example of customizing the "entity", "immutable-entity", "local-query", "pending-puts", "timestamps" cache configuration may look like:

```
<cache-container name="hibernate" module="org.hibernate.infinispan"
default-cache="immutable-entity">
  <local-cache name="entity">
    <transaction mode="NONE" />
    <eviction max-entries="-1" />
    <expiration max-idle="120000" />
  </local-cache>
  <local-cache name="immutable-entity">
    <transaction mode="NONE" />
    <eviction max-entries="-1" />
    <expiration max-idle="120000" />
  </local-cache>
  <local-cache name="local-query">
    <eviction max-entries="-1" />
    <expiration max-idle="300000" />
  </local-cache>
  <local-cache name="pending-puts">
    <transaction mode="NONE" />
    <eviction strategy="NONE" />
    <expiration max-idle="60000" />
  </local-cache>
  <local-cache name="timestamps">
    <transaction mode="NONE" />
    <eviction strategy="NONE" />
  </local-cache>
</cache-container>
```

Persistence.xml to use the above custom settings:

```
<properties>
  <property name="hibernate.cache.use_second_level_cache" value="true" />
  <property name="hibernate.cache.use_query_cache" value="true" />
  <property name="hibernate.cache.infinispan.immutable-entity.cfg" value="immutable-entity" />
  <property name="hibernate.cache.infinispan.timestamps.cfg" value="timestamps" />
  <property name="hibernate.cache.infinispan.pending-puts.cfg" value="pending-puts" />
</properties>
```



29.15 Replacing the current Hibernate 5.x jars with a newer version

Just update the current `wildfly/modules/system/layers/base/org/hibernate/main` folder to contain the newer version (after stopping your WildFly server instance).

1. Delete *.index files in `wildfly/modules/system/layers/base/org/hibernate/main` and `wildfly/modules/system/layers/base/org/hibernate/envers/main` folders.
2. Backup the current contents of `wildfly/modules/system/layers/base/org/hibernate` in case you make a mistake.
3. Remove the older jars and copy new Hibernate jars into `wildfly/modules/system/layers/base/org/hibernate/main` + `wildfly/modules/system/layers/base/org/hibernate/envers/main`.
4. Update the `wildfly/modules/system/layers/base/org/hibernate/main/module.xml` + `wildfly/modules/system/layers/base/org/hibernate/envers/main/module.xml` to name the jars that you copied in.
5. Also update the hibernate-infinispan jars in `wildfly/modules/system/layers/base/org/hibernate/infinispan`.

29.16 Using Hibernate Search

WildFly 10 includes Hibernate Search. If you want to use the bundled version of Hibernate Search - which requires to use the default Hibernate ORM 5 persistence provider - this will be automatically enabled. Having this enabled means that, provided your application includes any entity which is annotated with **org.hibernate.search.annotations.Indexed**, the module **org.hibernate.search.orm:main** will be made available to your deployment; this will also include the required version of Apache Lucene.

If you do not want this module to be exposed to your deployment, set the persistence property **wildfly.jpa.hibernate.search.module** to either **none** to not automatically inject any Hibernate Search module, or to any other module identifier to inject a different module.

For example you could set **wildfly.jpa.hibernate.search.module=org.hibernate.search.orm:5.4.0.Alpha1** to use the experimental version 5.4.0.Alpha1 instead of the provided module; in this case you'll have to download and add the custom modules to the application server as other versions are not included.

When setting **wildfly.jpa.hibernate.search.module=none** you might also opt to include Hibernate Search and its dependencies within your application but we highly recommend the modules approach.



29.17 Packaging the Hibernate JPA persistence provider with your application

WildFly 8 allows the packaging of Hibernate 4.x (or greater) persistence provider jars with the application. The JPA deployer will detect the presence of a persistence provider in the application and

jboss.as.jpa.providerModule needs to be set to **application**.<?xml version="1.0" encoding="UTF-8"?>

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence" version="1.0">
```

```
<persistence-unit name="myOwnORMVersion_pu">
```

```
<description>Hibernate 4 Persistence Unit.</description>
```

```
<jta-data-source>java:jboss/datasources/PlannerDS</jta-data-source>
```

```
<properties>
```

```
  <property name="jboss.as.jpa.providerModule" value="application" />
```

```
</properties>
```

```
</persistence-unit>
```

```
</persistence>
```





29.18 Using OpenJPA

You need to copy the OpenJPA jars (e.g. openjpa-all.jar serp.jar) into the WildFly modules/system/layers/base/org/apache/openjpa/main folder and update modules/system/layers/base/org/apache/openjpa/main/module.xml to include the same jar file names that you copied in.

```
<module xmlns="urn:jboss:module:1.1" name="org.apache.openjpa">
  <resources>
    <resource-root path="jipijapa-openjpa-1.0.1.Final.jar" />
    <resource-root path="openjpa-all.jar">
      <filter>
        <exclude path="javax/**" />
      </filter>
    </resource-root>
    <resource-root path="serp.jar" />
  </resources>

  <dependencies>
    <module name="javax.api" />
    <module name="javax.annotation.api" />
    <module name="javax.enterprise.api" />
    <module name="javax.persistence.api" />
    <module name="javax.transaction.api" />
    <module name="javax.validation.api" />
    <module name="javax.xml.bind.api" />
    <module name="org.apache.commons.collections" />
    <module name="org.apache.commons.lang" />
    <module name="org.jboss.as.jpa.spi" />
    <module name="org.jboss.logging" />
    <module name="org.jboss.vfs" />
    <module name="org.jboss.jandex" />
  </dependencies>
</module>
```

29.19 Using EclipseLink

You need to copy the EclipseLink jar (e.g. eclipselink-2.6.0.jar or eclipselink.jar as in the example below) into the WildFly modules/system/layers/base/org/eclipse/persistence/main folder and update modules/system/layers/base/org/eclipse/persistence/main/module.xml to include the EclipseLink jar (take care to use the jar name that you copied in). If you happen to leave the EclipseLink version number in the jar name, the module.xml should reflect that.



```
<module xmlns="urn:jboss:module:1.1" name="org.eclipse.persistence">
  <resources>
    <resource-root path="jipijapa-eclipselink-10.0.0.Final.jar" />
    <resource-root path="eclipselink.jar">
      <filter>
        <exclude path="javax/**" />
      </filter>
    </resource-root>
  </resources>

  <dependencies>
    <module name="asm.asm" />
    <module name="javax.api" />
    <module name="javax.annotation.api" />
    <module name="javax.enterprise.api" />
    <module name="javax.persistence.api" />
    <module name="javax.transaction.api" />
    <module name="javax.validation.api" />
    <module name="javax.xml.bind.api" />
    <module name="javax.ws.rs.api" />
    <module name="org.antlr" />
    <module name="org.apache.commons.collections" />
    <module name="org.dom4j" />
    <module name="org.jboss.as.jpa.spi" />
    <module name="org.jboss.logging" />
    <module name="org.jboss.vfs" />
  </dependencies>
</module>
```

As a workaround for [issue id=414974](#), set (WildFly) system property "eclipselink.archive.factory" to value "org.jipijapa.eclipselink.JBossArchiveFactoryImpl" via `jboss-cli.sh` command (WildFly server needs to be running when this command is issued):

```
jboss-cli.sh --connect
'/system-property=eclipselink.archive.factory:add(value=org.jipijapa.eclipselink.JBossArchiveFacto
```

. The following shows what the `standalone.xml` (or your WildFly configuration you are using) file might look like after updating the system properties:

```
<system-properties>
  ...
  <property name="eclipselink.archive.factory"
value="org.jipijapa.eclipselink.JBossArchiveFactoryImpl" />
</system-properties>
```

You should then be able to deploy applications with `persistence.xml` that include;

```
<provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
```



Also refer to page [how to use EclipseLink with WildFly guide here](#).

29.20 Using DataNucleus

Read the [how to use DataNucleus with WildFly guide here](#).

29.21 Native Hibernate use

Applications that use the Hibernate API directly, are referred to here as native Hibernate applications. Native Hibernate applications, can choose to use the Hibernate jars included with WildFly or they can package their own copy of the Hibernate jars. Applications that utilize JPA will automatically have the Hibernate classes injected onto the application deployment classpath. Meaning that JPA applications, should expect to use the Hibernate jars included in WildFly.

Example MANIFEST.MF entry to add dependency for Hibernate native applications:

```
Manifest-Version: 1.0
...
Dependencies: org.hibernate
```

If you use the Hibernate native api in your application and also use the JPA api to access the same entities (from the same Hibernate session/EntityManager), you could get surprising results (e.g. `HibernateSession.saveOrUpdate(entity)` is different than `EntityManager.merge(entity)`). Each entity should be managed by either Hibernate native API or JPA code.

29.22 Injection of Hibernate Session and SessionFactory

You can inject a `org.hibernate.Session` and `org.hibernate.SessionFactory` directly, just as you can do with `EntityManagers` and `EntityManagerFactories`.

```
import org.hibernate.Session;
import org.hibernate.SessionFactory;
@Stateful public class MyStatefulBean ... {
    @PersistenceContext(unitName="crm") Session session1;
    @PersistenceContext(unitName="crm2", type=EXTENDED) Session extendedpc;
    @PersistenceUnit(unitName="crm") SessionFactory factory;
}
```



29.23 Hibernate properties

WildFly automatically sets the following Hibernate (5.x) properties (if not already set in persistence unit definition):

Property	Purpose
hibernate.id.new_generator_mappings =true	New applications should let the default to true, older applications with existing data might need set to false (see note below). It really depends on whether your application uses the <code>@GeneratedValue(AUTO)</code> which will generate new key values for newly created entities. The application can override this value (in the persistence.xml)
hibernate.transaction.jta.platform = instance of <code>org.hibernate.service.jta.platform.spi.JtaPlatform</code> interface	The transaction manager, used for transaction and transaction synchronization registry is passed into Hibernate via this class.
hibernate.ejb.resource_scanner = instance of <code>org.hibernate.ejb.packaging.Scanner</code> interface	Instance of entity scanning class is passed in that knows how to use the AS annotation indexes (for faster deployment).
hibernate.transaction.manager_lookup_class	This property is removed if found in the persistence.xml (could conflict with JtaPlatform)
hibernate.session_factory_name = qualified persistence unit name	Is set to the application name or persistence unit name (application can specify a different value but it needs to be unique across all application deployments on the AS instance).
hibernate.session_factory_name_is_jndi = false	only set if the application didn't specify a value for <code>hibernate.session_factory_name</code>



hibernate.ejb.entitymanager_factory_name = qualified persistence unit name	Is set to the application name persistence unit name (application can specify a different value but it needs to be unique across all application deployments on the AS instance).
hibernate.query.jpql_strict_compliance =true	
hibernate.auto_quote_keyword =false	
hibernate.implicit_naming_strategy =org.hibernate.boot.model.naming.ImplicitNamingStrategyJpaCompliantImpl	

In Hibernate 4.x (and greater), if **new_generator_mappings** is **true**:

- @GeneratedValue(AUTO) maps to org.hibernate.id.enhanced.SequenceStyleGenerator
- @GeneratedValue(TABLE) maps to org.hibernate.id.enhanced.TableGenerator
- @GeneratedValue(SEQUENCE) maps to org.hibernate.id.enhanced.SequenceStyleGenerator

In Hibernate 4.x (and greater), if **new_generator_mappings** is **false**:

- @GeneratedValue(AUTO) maps to Hibernate "native"
- @GeneratedValue(TABLE) maps to org.hibernate.id.MultipleHiLoPerTableGenerator
- @GeneratedValue(SEQUENCE) to Hibernate "seqhilo"

29.24 Persistence unit properties

The following properties are supported in the persistence unit definition (in the persistence.xml file):

Property	Purpose
jboss.as.jpa.providerModule	name of the persistence provider module (default is org.hibernate). Should be application , if a persistence provider is packaged with the application. See note below about some module names that are built in (based on the provider).
jboss.as.jpa.adapterModule	name of the integration classes that help WildFly to work with the persistence provider.
jboss.as.jpa.adapterClass	class name of the integration adapter.
jboss.as.jpa.managed	set to false to disable container managed JPA access to the persistence unit. The default is true , which enables container managed JPA access to the persistence unit. This is typically set to false for Seam 2.x + Spring applications.



jboss.as.jpa.classtransformer	set to false to disable class transformers for the persistence unit. The default is true , which allows class enhancing/rewriting. Hibernate also needs persistence unit property hibernate.ejb.use_class_enhancer to be true, for class enhancing to be enabled.
wildfly.jpa.default-unit	set to true to choose the default persistence unit in an application. This is useful if you inject a persistence context without specifying the unitName (<code>@PersistenceContext EntityManager em</code>) but have multiple persistence units specified in your persistence.xml.
wildfly.jpa.twophasebootstrap	persistence providers (like Hibernate ORM 4.3.x via <code>EntityManagerFactoryBuilder</code>), allow a two phase persistence unit bootstrap, which improves JPA integration with CDI. Setting the wildfly.jpa.twophasebootstrap hint to false, disables the two phase bootstrap (for the persistence unit that contains the hint).
wildfly.jpa.allowdefaultdatasourceuse	set to false to prevent persistence unit from using the default data source. Defaults to true. This is only important for persistence units that do not specify a datasource.
jboss.as.jpa.deferdetach	Controls whether transaction scoped persistence context used in non-JTA transaction thread, will detach loaded entities after each <code>EntityManager</code> invocation or when the persistence context is closed (e.g. business method ends). Defaults to false (entities are cleared after <code>EntityManager</code> invocation) and if set to true, the detach is deferred until the context is closed.
wildfly.jpa.hibernate.search.module	Controls which version of Hibernate Search to include on classpath. Only makes sense when using Hibernate as JPA implementation. The default is auto ; other valid values are none or a full module identifier to use an alternative version.
jboss.as.jpa.scopedname	Specify the qualified (application scoped) persistence unit name to be used. By default, this is internally set to the application name + persistence unit name. The <code>hibernate.cache.region_prefix</code> will default to whatever you set <code>jboss.as.jpa.scopedname</code> to. Make sure you set the <code>jboss.as.jpa.scopedname</code> value to a value not already in use by other applications deployed on the same application server instance.



29.25 Determine the persistence provider module

As mentioned above, if the `jboss.as.jpa.providerModule` property is not specified, the provider module name is determined by the `provider` name specified in the `persistence.xml`. The mapping is:

Provider Name	Module name
blank	<code>org.hibernate</code>
<code>org.hibernate.ejb.HibernatePersistence</code>	<code>org.hibernate</code>
<code>org.hibernate.ogm.jpa.HibernateOgmPersistence</code>	<code>org.hibernate.ogm</code>
<code>oracle.toplink.essentials.PersistenceProvider</code>	<code>oracle.toplink</code>
<code>oracle.toplink.essentials.ejb.cmp3.EntityManagerFactoryProvider</code>	<code>oracle.toplink</code>
<code>org.eclipse.persistence.jpa.PersistenceProvider</code>	<code>org.eclipse.persistence</code>
<code>org.datanucleus.api.jpa.PersistenceProviderImpl</code>	<code>org.datanucleus</code>
<code>org.datanucleus.store.appengine.jpa.DatastorePersistenceProvider</code>	<code>org.datanucleus:appengine</code>
<code>org.apache.openjpa.persistence.PersistenceProviderImpl</code>	<code>org.apache.openjpa</code>



29.26 Binding EntityManagerFactory/EntityManager to JNDI

By default WildFly does **not** bind the entity manager factory to JNDI. However, you can explicitly configure this in the `persistence.xml` of your application by setting the `jboss.entity.manager.factory.jndi.name` hint. The value of that property should be the JNDI name to which the entity manager factory should be bound.

You can also bind a container managed (transaction scoped) entity manager to JNDI as well, } via hint `jboss.entity.manager.jndi.name`{. As a reminder, a transaction scoped entity manager (persistence context), acts as a proxy that always gets an unique underlying entity manager (at the persistence provider level).

Here's an example:

`persistence.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
  xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="
    http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
  <persistence-unit name="myPU">
    <!-- If you are running in a production environment, add a managed
      data source, the example data source is just for proofs of concept! -->
    <jta-data-source>java:jboss/datasources/ExampleDS</jta-data-source>
    <properties>
      <!-- Bind entity manager factory to JNDI at java:jboss/myEntityManagerFactory -->
      <property name="jboss.entity.manager.factory.jndi.name"
value="java:jboss/myEntityManagerFactory" />
      <property name="jboss.entity.manager.jndi.name" value="java:/myEntityManager"/>
    </properties>
  </persistence-unit>
</persistence>
```

```
@Stateful
public class ExampleSFSB {
  public void createSomeEntityWithTransactionScopedEM(String name) {
    Context context = new InitialContext();
    javax.persistence.EntityManager entityManager = (javax.persistence.EntityManager)
context.lookup("java:/myEntityManager");
    SomeEntity someEntity = new SomeEntity();
    someEntity.setName(name);    entityManager.persist(name);
  }
}
```



29.27 Community

Many thanks to the community, for reporting issues, solutions and code changes. A number of people have been answering Wildfly forum questions related to JPA usage. I would like to thank them for this, as well as those reporting issues. For those of you that haven't downloaded the AS source code and started hacking patches together. I would like to encourage you to start by reading [Hacking on WildFly](#). You will find that it is very easy to find your way around the WildFly/JPA/* source tree and make changes. Also, new for WildFly, is the JipiJapa project that contains additional integration code that makes EE JPA application deployments work better. The following list of contributors should grow over time, I hope to see more of you listed here.

29.27.1 People who have contributed to the WildFly JPA layer:

- [Carlo de Wolf](#) (lead of the EJB3 project)
- [Steve Ebersole](#) (lead of the Hibernate ORM project)
- [Stuart Douglas](#) (lead of the Seam Persistence project, WildFly project team member/commmitter)
- [Jaikiran Pai](#) (Active member of JBoss forums and JBoss EJB3 project team member)
- [Strong Liu](#) (leads the productization effort of Hibernate in the EAP product)
- [Scott Marlow](#) (lead of the WildFly container JPA sub-project)
- [Antti Laisi](#) (**OpenJPA integration changes**)
- [Galder Zamarreño](#) (Infinispan 2lc documentation)
- [Sanne Grinovero](#) (lead of the Hibernate Search project)
- [Paul Ferraro](#) (Infinispan 2lc integration)



30 OSGi

WildFly does not include support for OSGi, such functionality is now responsibility of JBoss OSGi project.

JBoss OSGi 2.5.0.Final will provide OSGi support for WildFly 10.

Release progress can be tracked via [JBOSGI-786](#).



31 Remote EJB invocations via JNDI - EJB client API or remote-naming project

31.1 Purpose

WildFly provides EJB client API project as well as remote-naming project for invoking on remote objects exposed via JNDI. This article explains which approach to use when and what the differences and scope of each of these projects is.

31.2 History

Previous versions of JBoss AS (versions < WildFly 8) used JNP project (<http://anonsvn.jboss.org/repos/jbossas/projects/naming/>) as the JNDI naming implementation. Developers of client applications of previous versions of JBoss AS will be familiar with the `jnp:// PROVIDER_URL` URL they used to use in their applications for communicating with the JNDI server on the JBoss server.

Starting WildFly 8, the JNP project is *not* used. Neither on the server side nor on the client side. The client side of the JNP project has now been replaced by `jboss-remote-naming` project (<https://github.com/jbossas/jboss-remote-naming>). There were various reasons why the JNP client was replaced by `jboss-remote-naming` project. One of them was the JNP project did not allow fine grained security configurations while communicating with the JNDI server. The `jboss-remote-naming` project is backed by the `jboss-remoting` project (<https://github.com/jboss-remoting/jboss-remoting>) which allows much more and better control over security.

31.3 Overview

Now that we know that for remote client JNDI communication with WildFly 8 requires `jboss-remote-naming` project, let's quickly see what the code looks like.

31.3.1 Client code relying on `jndi.properties` in classpath

```
void doLookup() {
    // Create an InitialContext using the javax.naming.* API
    Context ctx = new InitialContext();
    ctx.lookup("foo/bar");
    ...
}
```




As you can see, there's not much here in terms of code. We first create a `InitialContext` (<http://download.oracle.com/javase/6/docs/api/javax/naming/InitialContext.html>) which as per the API will look for a `jndi.properties` in the classpath of the application. We'll see what our `jndi.properties` looks like, later. Once the `InitialContext` is created, we just use it to do a lookup on a JNDI name which we know is bound on the server side. We'll come back to the details of this lookup string in a while.

Let's now see what the `jndi.properties` in our client classpath looks like:

```
java.naming.factory.initial=org.jboss.naming.remote.client.InitialContextFactory
java.naming.provider.url=http-remoting://localhost:8080
```

Those 2 properties are important for `jboss-remote-naming` project to be used for communicating with the WildFly server. The first property tells the JNDI API which initial context factory to use. In this case we are pointing it to the `InitialContextFactory` class supplied by the `jboss-remote-naming` project. The other property is the `PROVIDER_URL`. Developers familiar with previous JBoss AS versions would remember that they used `jnp://localhost:1099` (just an example). In WildFly, the URI protocol scheme for `jboss-remote-naming` project is `remote://`. The rest of the `PROVIDER_URL` part is the server hostname or IP and the port on which the remoting connector is exposed on the server side. By default the `http-remoting` connector port in WildFly 8 is 8080. That's what we have used in our example. The hostname we have used is `localhost` but that should point to the server IP or hostname where the server is running.

 JNP client project in previous AS versions allowed a comma separated list for `PROVIDER_URL` value, so that if one of the server isn't accessible then the JNDI API would use the next available server. The `jboss-remote-naming` project has similar support starting 1.0.3.Final version of that project (which is available in a WildFly release **after** 7.1.1.Final).

WildFly 8 can use the `PROVIDER_URL` like:

```
java.naming.provider.url=http-remoting://server1:8080,http-remoting://server2:8080
```

So we saw how to setup the JNDI properties in the `jndi.properties` file. The JNDI API also allows you to pass these properties to the constructor of the `InitialContext` class (please check the javadoc of that class for more details). Let's quickly see what the code would look like:

```
void doLookup() {
    Properties jndiProps = new Properties();
    jndiProps.put(Context.INITIAL_CONTEXT_FACTORY,
"org.jboss.naming.remote.client.InitialContextFactory");
    jndiProps.put(Context.PROVIDER_URL, "http-remoting://localhost:8080");
    // create a context passing these properties
    Context ctx = new InitialContext(jndiProps);
    // lookup
    ctx.lookup("foo/bar");
    ...
}
```




That's it! You can see that the values that we pass to those properties are the same as what we did via the `jndi.properties`. It's upto the client application to decide which approach they want to follow.

31.3.2 How does remoting naming work

We have so far had an overview of how the client code looks like when using the `jboss-remote-naming` (henceforth referred to as `remote-naming` - too tired of typing `jboss-remote-naming` everytime 😊) project. Let's now have a brief look at how the `remote-naming` project internally establishes the communication with the server and allows JNDI operations from the client side.

Like previously mentioned, `remote-naming` internally uses `jboss-remoting` project. When the client code creates an `InitialContext` backed by the `org.jboss.naming.remote.client.InitialContextFactory` class, the `org.jboss.naming.remote.client.InitialContextFactory` internally looks for the `PROVIDER_URL` (and other) properties that are applicable for that context (*doesn't* matter whether it comes from the `jndi.properties` file or whether passed explicitly to the constructor of the `InitialContext`). Once it identifies the server and port to connect to, the `remote-naming` project internally sets up a connection using the `jboss-remoting` APIs with the `remoting` connector which is exposed on that port.

We previously mentioned that `remote-naming`, backed by `jboss-remoting` project, has increased support for security configurations. Starting WildFly 8, every service including the `http` `remoting` connector (which listens by default on port 8080), is secured (see <https://community.jboss.org/wiki/AS710Beta1-SecurityEnabledByDefault> for details). This means that when trying to do JNDI operations like a lookup, the client has to pass appropriate user credentials. In our examples so far we haven't passed any username/pass or any other credentials while creating the `InitialContext`. That was just to keep the examples simple. But let's now take the code a step further and see one of the ways how we pass the user credentials. Let's at the moment just assume that the `remoting` connector on port 8080 is accessible to a user named "peter" whose password is expected to be "lois".



Note: The server side configurations for the `remoting` connector to allow "peter" to access the connector, is out of the scope of this documentation. The WildFly 8 documentation already has chapters on how to set that up.



```
void doLookup() {
    Properties jndiProps = new Properties();
    jndiProps.put(Context.INITIAL_CONTEXT_FACTORY,
"org.jboss.naming.remote.client.InitialContextFactory");
    jndiProps.put(Context.PROVIDER_URL, "http-remoting://localhost:8080");
    // username
    jndiProps.put(Context.SECURITY_PRINCIPAL, "peter");
    // password
    jndiProps.put(Context.SECURITY_CREDENTIALS, "lois");
    // create a context passing these properties
    Context ctx = new InitialContext(jndiProps);
    // lookup
    ctx.lookup("foo/bar");
    ...
}
```

The code is similar to our previous example, except that we now have added 2 additional properties that are passed to the InitialContext constructor. The first is http://docs.oracle.com/javase/6/docs/api/javax/naming/Context.html#SECURITY_PRINCIPAL which passes the username (peter in this case) and the second is http://docs.oracle.com/javase/6/docs/api/javax/naming/Context.html#SECURITY_CREDENTIALS which passes the password (lois in this case). Of course the same properties can be configured in the jndi.properties file (read the javadoc of the Context class for appropriate properties to be used in the jndi.properties). This is one way of passing the security credentials for JNDI communication with WildFly. There are some other ways to do this too. But we won't go into those details here for two reasons. One, it's outside the scope of this article and two (which is kind of the real reason) I haven't looked fully at the remote-naming implementation details to see what other ways are allowed.

31.3.3 JNDI operations allowed using remote-naming project

So far we have mainly concentrated on how the naming context is created and what it internally does when an instance is created. Let's now take this one step further and see what kind of operations are allowed for a JNDI context backed by the remote-naming project.

The JNDI Context has various methods <http://docs.oracle.com/javase/6/docs/api/javax/naming/Context.html> that are exposed for JNDI operations. One important thing to note in case of remote-naming project is that, the project's scope is to allow a client to communicate with the JNDI backend exposed by the server. As such, the remote-naming project does **not** support many of the methods that are exposed by the javax.naming.Context class. The remote-naming project only supports the read-only kind of methods (like the lookup() method) and does not support any write kind of methods (like the bind() method). The client applications are expected to use the remote-naming project mainly for lookups of JNDI objects. Neither WildFly 8 nor remote-naming project allows writing/binding to the JNDI server from a remote application.



31.3.4 prerequisites of remotely accessible JNDI objects

On the server side, the JNDI can contain numerous objects that are bound to it. However, *not* all of those are exposed remotely. The two conditions that are to be satisfied by the objects bound to JNDI, to be remotely accessible are:

- 1) Such objects should be bound under the `java:jboss/exported/` namespace. For example,
`java:jboss/exported/foo/bar`
- 2) Objects bound to the `java:jboss/exported/` namespace are expected to be serializable. This allows the objects to be sent over the wire to the remote clients

Both these conditions are important and are required for the objects to be remotely accessible via JNDI.

31.3.5 JNDI lookup strings for remote clients backed by the remote-naming project

In our examples, so far, we have been consistently using `"foo/bar"` as the JNDI name to lookup from a remote client using the remote-naming project. There's a bit more to understand about the JNDI name and how it maps to the JNDI name that's bound on the server side.

First of all, the JNDI names used while using the remote-naming project are **always** relative to the `java:jboss/exported/` namespace. So in our examples, we are using `"foo/bar"` JNDI name for the lookup, that actually is (internally) `"java:jboss/exported/foo/bar"`. The remote-naming project expects it to **always** be relative to the `"java:jboss/exported/"` namespace. Once connected with the server side, the remote-naming project will lookup for `"foo/bar"` JNDI name under the `"java:jboss/exported/"` namespace of the server.



Note: Since the JNDI name that you use on the client side is **always** relative to `java:jboss/exported` namespace, you **shouldn't** be prefixing the `java:jboss/exported/` string to the JNDI name. For example, if you use the following JNDI name:

```
ctx.lookup("java:jboss/exported/helloworld");
```

then remote-naming will translate it to

```
ctx.lookup("java:jboss/exported/java:jboss/exported/helloworld");
```

and as a result, will fail during lookup.

The remote-naming implementation perhaps should be smart enough to strip off the `java:jboss/exported/` namespace prefix if supplied. But let's not go into that here.



31.3.6 How does remote-naming project implementation transfer the JNDI objects to the clients

When a lookup is done on a JNDI string, the remote-naming implementation internally uses the connection to the remoting connector (which it has established based on the properties that were passed to the `InitialContext`) to communicate with the server. On the server side, the implementation then looks for the JNDI name under the `java:jboss/exported/` namespace. Assuming that the JNDI name is available, under that namespace, the remote-naming implementation then passes over the object bound at that address to the client. This is where the requirement about the JNDI object being serializable comes into picture. remote-naming project internally uses jboss-marshalling project to marshal the JNDI object over to the client. On the client side the remote-naming implementation then unmarshals the object and returns it to the client application.

So literally, each lookup backed by the remote-naming project entails a server side communication/interaction and then marshalling/unmarshalling of the object graph. This is very important to remember. We'll come back to this later, to see why this is important when it comes to using EJB client API project for doing EJB lookups ([EJB invocations from a remote client using JNDI](#)) as against using remote-naming project for doing the same thing.

31.4 Summary

That pretty much covers whatever is important to know, in the remote-naming project, for a typical client application. Don't close the browser yet though, since we haven't yet come to the part of EJB invocations from a remote client using the remote-naming project. In fact, the motivation behind writing this article was to explain why *not* to use remote-naming project (in most cases) for doing EJB invocations against WildFly server.

Those of you who don't have client applications doing remote EJB invocations, can just skip the rest of this article if you aren't interested in those details.

31.5 Remote EJB invocations backed by the remote-naming project

In previous sections of this article we saw that whatever is exposed in the `java:jboss/exported/` namespace is accessible remotely to the client applications under the relative JNDI name. Some of you might already have started thinking about exposing remote views of EJBs under that namespace.

It's important to note that WildFly server side already by default exposes the remote views of a EJB under the `java:jboss/exported/` namespace (although it isn't logged in the server logs). So assuming your server side application has the following stateless bean:



```
package org.myapp.ejb;

@Stateless
@Remote(Foo.class)
public class FooBean implements Foo {
    ...
    public String sayBar() {
        return "Baaaaaaaar";
    }
}
```

Then the "Foo" remote view is exposed under the `java:jboss/exported/` namespace under the following JNDI name scheme (which is similar to that mandated by EJB3.1 spec for `java:global/namespace`): `[app-name]`

`app-name/module-name/bean-name!bean-interface`

where,

`app-name` = the name of the `.ear` (without the `.ear` suffix) or the application name configured via `application.xml` deployment descriptor. If the application isn't packaged in a `.ear` then there will be **no** `app-name` part to the JNDI string.

`module-name` = the name of the `.jar` or `.war` (without the `.jar/.war` suffix) in which the bean is deployed or the `module-name` configured in `web.xml/ejb-jar.xml` of the deployment. The module name is mandatory part in the JNDI string.

`bean-name` = the name of the bean which by default is the simple name of the bean implementation class. Of course it can be overridden either by using the "name" attribute of the bean defining annotation (`@Stateless(name="blah")` in this case) or even the `ejb-jar.xml` deployment descriptor.

`bean-interface` = the fully qualified class name of the interface being exposed by the bean.

So in our example above, let's assume the bean is packaged in a `myejbmodule.jar` which is within a `myapp.ear`. So the JNDI name for the Foo remote view under the `java:jboss/exported/` namespace would be:

```
java:jboss/exported/myapp/myejbmodule/FooBean!org.myapp.ejb.Foo
```

That's where WildFly will **automatically** expose the remote views of the EJBs under the `java:jboss/exported/` namespace, **in addition to** the `java:global/` `java:app/` `java:module/` namespaces mandated by the EJB 3.1 spec.



Note that only the `java:jboss/exported/` namespace is available to remote clients.

So the next logical question would be, are these remote views of EJBs accessible and invocable using the remote-naming project on the client application. The answer is yes! Let's quickly see the client code for invoking our `FooBean`. Again, let's just use "peter" and "lois" as username/pass for connecting to the remoting connector.



```
void doBeanLookup() {
    ...
    Properties jndiProps = new Properties();
    jndiProps.put(Context.INITIAL_CONTEXT_FACTORY,
"org.jboss.naming.remote.client.InitialContextFactory");
    jndiProps.put(Context.PROVIDER_URL, "http-remoting://localhost:8080");
    // username
    jndiProps.put(Context.SECURITY_PRINCIPAL, "peter");
    // password
    jndiProps.put(Context.SECURITY_CREDENTIALS, "lois");
    // This is an important property to set if you want to do EJB invocations via the
remote-naming project
    jndiProps.put("jboss.naming.client.ejb.context", true);
    // create a context passing these properties
    Context ctx = new InitialContext(jndiProps);
    // lookup the bean      Foo
    beanRemoteInterface = (Foo) ctx.lookup("myapp/myejbmodule/FooBean!org.myapp.ejb.Foo");
    String bar = beanRemoteInterface.sayBar();
    System.out.println("Remote Foo bean returned " + bar);
    ctx.close();
    // after this point the beanRemoteInterface is not longer valid!
}
```

As you can see, most of the code is similar to what we have been seeing so far for setting up a JNDI context backed by the remote-naming project. The only parts that change are:

- 1) An additional `"jboss.naming.client.ejb.context"` property that is added to the properties passed to the `InitialContext` constructor.
- 2) The JNDI name used for the lookup
- 3) And subsequently the invocation on the bean interface returned by the lookup.

Let's see what the `"jboss.naming.client.ejb.context"` does. In WildFly, remote access/invocations on EJBs is facilitated by the JBoss specific EJB client API, which is a project on its own <https://github.com/jbossas/jboss-ejb-client>. So no matter, what mechanism you use (remote-naming or core EJB client API), the invocations are ultimately routed through the EJB client API project. In this case too, the remote-naming internally uses EJB client API to handle EJB invocations. From a EJB client API project perspective, for successful communication with the server, the project expects a `EJBClientContext` backed by (atleast one) `EJBReceiver(s)`. The `EJBReceiver` is responsible for handling the EJB invocations. One type of a `EJBReceiver` is a `RemotingConnectionEJBReceiver` which internally uses `jboss-remoting` project to communicate with the remote server to handle the EJB invocations. Such a `EJBReceiver` expects a connection backed by the `jboss-remoting` project. Of course to be able to connect to the server, such a `EJBReceiver` would have to know the server address, port, security credentials and other similar parameters. If you were using the core EJB client API, then you would have configured all these properties via the `jboss-ejb-client.properties` or via programatic API usage as explained here [EJB invocations from a remote client using JNDI](#). But in the example above, we are using remote-naming project and are *not* directly interacting with the EJB client API project.



If you look closely at what's being passed, via the JNDI properties, to the remote-naming project and if you remember the details that we explained in a previous section about how the remote-naming project establishes a connection to the remote server, you'll realize that these properties are indeed the same as what the `RemotingConnectionEJBReceiver` would expect to be able to establish the connection to the server. Now this is where the `"jboss.naming.client.ejb.context"` property comes into picture. When this is set to true and passed to the `InitialContext` creation (either via `jndi.properties` or via the constructor of that class), the remote-naming project internally will do whatever is necessary to setup a `EJBClientContext`, containing a `RemotingConnectionEJBReceiver` which is created using the **same** remoting connection that is created by and being used by remote-naming project for its own JNDI communication usage. So effectively, the `InitialContext` creation via the remote-naming project has now internally triggered the creation of a `EJBClientContext` containing a `EJBReceiver` capable of handling the EJB invocations (remember, no remote EJB invocations are possible without the presence of a `EJBClientContext` containing a `EJBReceiver` which can handle the EJB).

So we now know the importance of the `"jboss.naming.client.ejb.context"` property and its usage. Let's move on the next part in that code, the JNDI name. Notice that we have used the JNDI name relative to the `java:jboss/exported/` namespace while doing the lookup. And since we know that the Foo view is exposed on that JNDI name, we cast the returned object back to the Foo interface. Remember that we earlier explained how each lookup via remote-naming triggers a server side communication and a marshalling/unmarshalling process. This applies for EJB views too. In fact, the remote-naming project has no clue (since that's not in the scope of that project to know) whether it's an EJB or some random object.

Once the unmarshalled object is returned (which actually is a proxy to the bean), the rest is straightforward, we just invoke on that returned object. Now since the remote-naming implementation has done the necessary setup for the `EJBClientContext` (due to the presence of `"jboss.naming.client.ejb.context"` property), the invocation on that proxy will internally use the `EJBClientContext` (the proxy is smart enough to do that) to interact with the server and return back the result. We won't go into the details of how the EJB client API handles the communication/invocation.

Long story short, using the remote-naming project for doing remote EJB invocations against WildFly is possible!

31.6 Why use the EJB client API approach then?

I can guess that some of you might already question why/when would one use the EJB client API style lookups as explained in the [EJB invocations from a remote client using JNDI](#) article instead of just using (what appears to be a simpler) remote-naming style lookups.

Before we answer that, let's understand a bit about the EJB client project. The EJB client project was implemented keeping in mind various optimizations and features that would be possible for handling remote invocations. One such optimization was to avoid doing unnecessary server side communication(s) which would typically involve network calls, marshalling/unmarshalling etc... The easiest place where this optimization can be applied, is to the EJB lookup. Consider the following code (let's ignore how the context is created):



```
ctx.lookup("foo/bar");
```

Now `foo/bar` JNDI name could potentially point to **any** type of object on the server side. The `jndi` name itself won't have the type/semantic information of the object bound to that name on the server side. If the context was setup using the remote-naming project (like we have seen earlier in our examples), then the only way for remote-naming to return an object for that lookup operation is to communicate with the server and marshal/unmarshal the object bound on the server side. And that's exactly what it does (remember, we explained this earlier).

The EJB client API project on the other hand optimizes this lookup. In order to do so, it expects the client application to let it know that a EJB is being looked up. It does this, by expecting the client application to use the JNDI name of the format `"ejb:"` namespace and also expecting the client application to setup the JNDI context by passing the `"org.jboss.ejb.client.naming"` value for the `Context.URL_PKG_PREFIXES` property.

Example:

```
final Properties jndiProperties = new Properties();
jndiProperties.put(Context.URL_PKG_PREFIXES, "org.jboss.ejb.client.naming");
// create the context
final Context context = new InitialContext(jndiProperties);

// lookup
Foo beanProxy = context.lookup("ejb:myapp/myejbmodule//FooBean!org.myapp.ejb.Foo");
String bar = beanProxy.sayBar();
```

More details about such code can be found here [EJB invocations from a remote client using JNDI](#)

When a client application looks up anything under the `"ejb:"` namespace, it is a clear indication (for the EJB client API project) to know that the client is looking up an EJB. That's where it steps in to do the necessary optimizations that might be applicable. So unlike, in the case of remote-naming project (which has no clue about the semantics of the object being looked up), the EJB client API project does **not** trigger a server side communication or a marshal/unmarshal process when you do lookup for a remote view of a stateless bean (it's important to note that we have specifically mentioned stateless bean here, we'll come to that later). Instead, the EJB client API just returns a `java.lang.reflect.Proxy` instance of the remote view type that's being looked up. This not just saves a network call, marshalling/unmarshalling step but it also means that you can create an EJB proxy even when the server isn't up yet. Later on, when the invocation on the proxy happens, the EJB client API *does* communicate with the server to carry out the invocation.



31.6.1 Is the lookup optimization applicable for all bean types?

In the previous section we (intentionally) mentioned that the lookup optimization by the EJB client API project happens for stateless beans. This kind of optimization is **not** possible for stateful beans because in case of stateful beans, a lookup is expected to create a session for that stateful bean and for session creation we do have to communicate with the server since the server is responsible for creating that session.

That's exactly why the EJB client API project expects the JNDI name lookup string for stateful beans to include the "?stateful" string at the end of the JNDI name:

```
context.lookup("ejb:myapp/myejbmodule//StatefulBean!org.myapp.ejb.Counter?stateful");
```

Notice the use of "?stateful" in that JNDI name. See [EJB invocations from a remote client using JNDI](#) for more details about such lookup.

The presence of "?stateful" in the JNDI name lookup string is a directive to the EJB client API to let it know that a stateful bean is being looked up and it's necessary to communicate with the server and create a session during that lookup.

So as you can see, we have managed to optimize certain operations by using the EJB client API for EJB lookup/invocation as against using the remote-naming project. There are other EJB client API implementation details (and probably more might be added) which are superior when it is used for remote EJB invocations in client applications as against remote-naming project which doesn't have the intelligence to carry out such optimizations for EJB invocations. *That's why the remote-naming project for remote EJB invocations is considered "deprecated"*. Note that if you want to use remote-naming for looking up and invoking on non-EJB remote objects then you are free to do so. In fact, that's why that project has been provided. You can even use the remote-naming project for EJB invocations (like we just saw), if you are fine with *not* wanting the optimizations that the EJB client API can do for you or if you have other restrictions that force you to use that project.



31.6.2 Restrictions for EJB's

If the remote-naming is used there are some restrictions as there is no full support of the ejb-client features.

- No loadbalancing, if the URL contains multiple "remote://" servers there is no loadbalancing, the first available server will be used and only in case it is not longer available there will be a failover to the next available one.
- No cluster support. As a cluster needs to be defined in the `jboss-ejb-client.properties` this feature can not be used and there is no cluster node added
- No client side interceptor. The `EJBContext.getCurrent()` can not be used and it is not possible to add a client interceptor
- No `UserTransaction` support
- All proxies become invalid if `.close()` for the related `InitialContext` is invoked, or the `InitialContext` is not longer referenced and gets garbage-collected. In this case the underlying `EJBContext` is destroyed and the connections are closed.
- It is not possible to use remote-naming if the client is an application deployed on another JBoss instance



32 Scoped EJB client contexts

32.1 Overview

WildFly 8 introduced the EJB client API for managing remote EJB invocations. The EJB client API works off `EJBClientContext(s)`. An `EJBClientContext` can potentially contain any number of EJB receivers. An EJB receiver is a component which knows how to communicate with a server which is capable of handling the EJB invocation. Typically EJB remote applications can be classified into:

- A remote client which runs as a standalone Java application
- A remote client which runs within another WildFly 8 instance

Depending on the kind of remote client, from an EJB client API point of view, there can potentially be more than 1 `EJBClientContext(s)` within a JVM.

In case of standalone applications, typically a single `EJBClientContext` (backed by any number of EJB receivers) exists. However this isn't mandatory. Certain standalone applications can potentially have more than one `EJBClientContext(s)` and a EJB client context selector will be responsible for returning the appropriate context.

In case of remote clients which run within another WildFly 8 instance, each deployed application will have a corresponding EJB client context. Whenever that application invokes on another EJB, the corresponding EJB client context will be used for finding the right EJB receiver and letting it handle the invocation.

32.2 Potential shortcomings of a single EJB client context

In the Overview section we briefly looked at the different types of remote clients. Let's focus on the standalone remote clients (the ones that don't run within another WildFly 8 instance) for some of the next sections. Like mentioned earlier, typically a remote standalone client has just one EJB client context backed by any number of EJB receivers. Consider this example:

```
public class MyApplication {

    public static void main(String args[]) {

        final javax.naming.Context ctxOne = new javax.naming.InitialContext();
        final MyBeanInterface beanOne = ctxOne.lookup("ejb:app/module/distinct/bean!interface");
        beanOne.doSomething();
        ...
    }
}
```



Now, we have seen in this other chapter [EJB invocations from a remote client using JNDI](#) that the JNDI lookups are (typically) backed by `jboss-ejb-client.properties` file which is used to setup the EJB client context and the EJB receivers. Let's assume we have a `jboss-ejb-client.properties` with the relevant receivers configurations. These configurations include the security credentials that will be used to create a EJB receiver which connects to the AS7 server. Now when the above code is invoked, the EJB client API looks for the EJB client context to pick a EJB receiver, to pass on the EJB invocation request. Since we just have a single EJB client context, that context is used by the above code to invoke the bean.

Now let's consider a case where the user application wants to invoke on the bean more than once, but wants to connect to the WildFly 8 server using different security credentials. Let's take a look at the following code:

```
public class MyApplication {

    public static void main(String args[]) {

        // let's say we want to use "foo" security credential while connecting to the AS7 server
        for invoking on this bean instance
        final javax.naming.Context ctxOne = new javax.naming.InitialContext();
        final MyBeanInterface beanOne = ctxOne.lookup("ejb:app/module/distinct/bean!interface");
        beanOne.doSomething();
        ...

        // let's say we want to use "bar" security credential while connecting to the AS7 server
        for invoking on this bean instance
        final javax.naming.Context ctxTwo = new javax.naming.InitialContext();
        final MyBeanInterface beanTwo = ctxTwo.lookup("ejb:app/module/distinct/bean!interface");
        beanTwo.doSomething();
        ...

    }
}
```

So we have the same application, which wants to connect to the same server instance for invoking the EJB(s) hosted on that server, but wants to use two different credentials while connecting to the server. Remember, the client application has a single EJB client context which can have at most 1 EJB receiver for each server instance. Which effectively means that the above code will end up using just one credential to connect to the server. So there was no easy way to have the above code working.

That was one of the use cases which prompted the <https://issues.jboss.org/browse/EJBCLIENT-34> feature request. The proposal was to introduce a way, where you can have more control over the EJB client contexts and their association with JNDI contexts which are typically used for EJB invocations.



32.3 Scoped EJB client contexts

Developers familiar with earlier versions of JBoss AS would remember that for invoking an EJB, you would typically create a JNDI context passing it the PROVIDER_URL which would point to the target server. That way any invocation done on EJB proxies looked up using that JNDI context, would end up on that server. If we look back at the example above, we'll realize that, we are ultimately aiming for a similar functionality through <https://issues.jboss.org/browse/EJBCLIENT-34>. We want the user applications to have more control over which EJB receiver gets used for a specific invocation.

Before we introduced <https://issues.jboss.org/browse/EJBCLIENT-34> feature, the EJB client context was typically scoped to the client application. As part of <https://issues.jboss.org/browse/EJBCLIENT-34> we now allow the EJB client contexts to be scoped with the JNDI contexts. Consider the following example:

```
public class MyApplication {

    public static void main(String args[]) {

        // let's say we want to use "foo" security credential while connecting to the AS7 server
        for invoking on this bean instance
        final Properties ejbClientContextPropsOne = getPropsForEJBClientContextOne():
        final javax.naming.Context ctxOne = new
        javax.naming.InitialContext(ejbClientContextPropsOne);
        final MyBeanInterface beanOne = ctxOne.lookup("ejb:app/module/distinct/bean!interface");
        beanOne.doSomething();
        ...
        closeContext(ctxOne); // read on the entire article to understand more about closing
        scoped EJB client contexts

        // let's say we want to use "bar" security credential while connecting to the AS7 server
        for invoking on this bean instance
        final Properties ejbClientContextPropsTwo = getPropsForEJBClientContextTwo():
        final javax.naming.Context ctxTwo = new
        javax.naming.InitialContext(ejbClientContextPropsTwo);
        final MyBeanInterface beanTwo = ctxTwo.lookup("ejb:app/module/distinct/bean!interface");
        beanTwo.doSomething();
        ...
        closeContext(ctxTwo); // read on the entire article to understand more about closing
        scoped EJB client contexts
    }
}
```

Notice any difference between this code and the earlier one? We now create and pass EJB client context specific properties to the JNDI context. So what do the EJB client context properties look like? The properties are the same that you would pass through the `jboss-ejb-client.properties` file, except for one additional property which is required to scope the EJB client context to the JNDI context. The name of the property is:

```
org.jboss.ejb.client.scoped.context
```



which is expected to have a value true. This property lets the EJB client API know that it has to create an EJB client context (backed by EJB receiver(s)) and that created context is then scoped/visible to only that JNDI context which created it. Lookup and invocation on any EJB proxies looked up using this JNDI context will only know of the EJB client context associated with this JNDI context. This effectively means that the other JNDI contexts which the application uses to lookup and invoke on EJBs will *not* know about the other scoped EJB client contexts at all.

JNDI contexts which aren't scoped to an EJB client context (for example, not passing the `org.jboss.ejb.client.scoped.context` property) will fallback to the default behaviour of using the "current" EJB client context which typically is the one tied to the entire application.

This scoping of the EJB client context helps the user applications to have more control over which JNDI context "talks to" which server and connects to that server in "what way". This gives the user applications the flexibility that was associated with the JNP based JNDI invocations prior to WildFly 8 versions.



IMPORTANT: It is very important to remember that scoped EJB client contexts which are scoped to the JNDI contexts are NOT fire and forget kind of contexts. What that means is the application program which is using these contexts is solely responsible for managing their lifecycle and the application itself is responsible for closing the context at the right moment. After closing the context the proxies which are bound to this context are no longer valid and any invocation will throw an Exception. Not closing the context will end in resource problems as the underlying physical connection will stay open.

Read the rest of the sections in this article to understand more about the lifecycle management of such scoped contexts.


32.4 Lifecycle management of scoped EJB client contexts

Like you saw in the previous sections, in case of scoped EJB client contexts, the EJB client context is tied to the JNDI context. It's very important to understand how the lifecycle of the EJB client context works in such cases. Especially since any EJB client context is almost always backed by connections to the server. Not managing the EJB client context lifecycle correctly can lead to connection leaks in some cases.

When you create a scoped EJB client context, the EJB client context connects to the server(s) listed in the JNDI properties. An internal implementation detail of this logic includes the ability of the EJB client context to cache connections based on certain internal algorithm it uses. The algorithm itself isn't publicly documented (yet) since the chances of it changing or even removal shouldn't really affect the client application and instead it's supposed to be transparent to the client application.

The connections thus created for an EJB client context are kept open as long as the EJB client context is open. This allows the EJB client context to be usable for EJB invocations. The connections associated with the EJB client context are closed when the EJB client context itself is closed.



 The connections that were manually added by the application to the EJB client context are **not** managed by the EJB client context. i.e. they won't be opened (obviously) nor closed by the EJB client API when the EJB client context is closed.

32.4.1 How to close EJB client contexts?

The answer to that is simple. Use the `close()` method on the appropriate EJB client context.

32.4.2 How to close scoped EJB client contexts?

The answer is the same, use the `close()` method on the EJB client context. But the real question is how do you get the relevant scoped EJB client context which is associated with a JNDI context. Before we get to that, it's important to understand how the `ejb:` JNDI namespace that's used for EJB lookups and how the JNDI context (typically the `InitialContext` that you see in the client code) are related. The JNDI API provided by Java language allows "URL context factory" to be registered in the JNDI framework (see this for details <http://docs.oracle.com/javase/jndi/tutorial/provider/url/factory.html>). Like that documentation states, the URL context factory can be used to resolve URL strings during JNDI lookup. That's what the `ejb:` prefix is when you do a remote EJB lookup. The `ejb:` URL string is backed by a URL context factory.

Internally, when a lookup happens for a `ejb:` URL string, a relevant `javax.naming.Context` is created for that `ejb:` lookup. Let's see some code for better understanding:

```
// JNDI context "A"
Context jndiCtx = new InitialContext(props);
// Now let's lookup a EJB
MyBean bean = jndiCtx.lookup("ejb:app/module/distinct/bean!interface");
```

So we first create a JNDI context and then use it to lookup an EJB. The bean lookup using the `ejb:` JNDI name, although, is just one statement, involves a few more things under the hood. What's actually happening when you lookup that string is that a separate `javax.naming.Context` gets created for the `ejb:` URL string. This new `javax.naming.Context` is then used to lookup the rest of the string in that JNDI name.

Let's break up that one line into multiple statements to understand better:

```
// Remember, the ejb: is backed by a URL context factory which returns a Context for the ejb:
// URL (that's why it's called a context factory)
final Context ejbNamingContext = (Context) jndiCtx.lookup("ejb:");
// Use the returned EJB naming context to lookup the rest of the JNDI string for EJB
final MyBean bean = ejbNamingContext.lookup("app/module/distinct/bean!interface");
```



As you see above, we split up that single statement into a couple of statements for explaining the details better. So as you can see when the `ejb:` URL string is parsed in a JNDI name, it gets hold of a `javax.naming.Context` instance. This instance is different from the one which was used to do the lookup (`jndiCtx` in this example). This is an important detail to understand (for reasons explained later). Now this returned instance is used to lookup the rest of the JNDI string ("`app/module/distinct/bean!interface`"), which then returns the EJB proxy. Irrespective of whether the lookup is done in a single statement or multiple parts, the code works the same. i.e. an instance of `javax.naming.Context` gets created for the `ejb:` URL string.

So why am I explaining all this when the section is titled "How to close scoped EJB client contexts"? The reason is because client applications dealing with scoped EJB client contexts which are associated with a JNDI context would expect the following code to close the associated EJB client context, but will be surprised that it won't:

```
final Properties props = new Properties();
// mark it for scoped EJB client context
props.put("org.jboss.ejb.client.scoped.context", "true");
// add other properties
props.put(...);
...
Context jndiCtx = new InitialContext(props);
try {
    final MyBean bean = jndiCtx.lookup("ejb:app/module/distinct/bean!interface");
    bean.doSomething();
} finally {
    jndiCtx.close();
}
```

Applications expect that the call to `jndiCtx.close()` will effectively close the EJB client context associated with the JNDI context. That doesn't happen because as explained previously, the `javax.naming.Context` backing the `ejb:` URL string is a different instance than the one the code is closing. The JNDI implementation in Java, only just closes the context on which the close was called. As a result, the other `javax.naming.Context` that backs the `ejb:` URL string is still not closed, which effectively means that the scoped EJB client context is not closed too which then ultimately means that the connection to the server(s) in the EJB client context are not closed too.

So now let's see how this can be done properly. We know that the `ejb:` URL string lookup returns us a `javax.naming.Context`. All we have to do is keep a reference to this instance and close it when we are done with the EJB invocations. So here's how it's going to look:



```
final Properties props = new Properties();
// mark it for scoped EJB client context
props.put("org.jboss.ejb.client.scoped.context", "true");
// add other properties
props.put(...);
...
Context jndiCtx = new InitialContext(props);
Context ejbRootNamingContext = (Context) jndiCtx.lookup("ejb:");
try {
    final MyBean bean = ejbRootNamingContext.lookup("app/module/distinct/bean!interface"); //
the rest of the EJB jndi string
    bean.doSomething();
} finally {
    try {
        // close the EJB naming JNDI context
        ejbRootNamingContext.close();
    } catch (Throwable t) {
        // log and ignore
    }
    try {
        // also close our other JNDI context since we are done with it too
        jndiCtx.close();
    } catch (Throwable t) {
        // log and ignore
    }
}
}
```

As you see, we changed the code to first do a lookup on just the "ejb:" string to get hold of the EJB naming context and then used that `ejbRootNamingContext` instance to lookup the rest of the EJB JNDI name to get hold of the EJB proxy. Then when it was time to close the context, we closed the `ejbRootNamingContext` (as well as the other JNDI context). Closing the `ejbRootNamingContext` ensures that the scoped EJB client context associated with that JNDI context is closed too. Effectively, this closes the connection(s) to the server(s) within that EJB client context.



Can that code be simplified a bit?

If you are using that JNDI context only for EJB invocations, then yes you can get rid of some instances and code from the above code. You can change that code to:

```
final Properties props = new Properties();
// mark it for scoped EJB client context
props.put("org.jboss.ejb.client.scoped.context", "true");
// add other properties
props.put(...);
...
Context.ejbRootNamingContext = (Context) new InitialContext(props).lookup("ejb:");
try {
    final MyBean bean =.ejbRootNamingContext.lookup("app/module/distinct/bean!interface"); //
the rest of the EJB jndi string
    bean.doSomething();
} finally {
    try {
        // close the EJB naming JNDI context
       .ejbRootNamingContext.close();
    } catch (Throwable t) {
        // log and ignore
    }
}
```

Notice that we no longer hold a reference to 2 JNDI contexts and instead just keep track of the `ejbRootNamingContext` which is actually the root JNDI context for our "ejb:" URL string. Of course, this means that you can only use this context for EJB lookups or any other EJB related JNDI lookups. So it depends on your application and how it's coded.



32.4.3 Can't the scoped EJB client context be automatically closed by the EJB client API when the JNDI context is no longer in scope (i.e. on GC)?

That's one of the common questions that gets asked. No, the EJB client API can't take that decision. i.e. it cannot automatically go ahead and close the scoped EJB client context by itself when the associated JNDI context is eligible for GC. The reason is simple as illustrated by the following code:

```
void doEJBInvocation() {
    final MyBean bean = lookupEJB();
    bean.doSomething();
    bean.doSomeOtherThing();
    ... // do some other work
    bean.keepDoingSomething();
}

MyBean lookupEJB() {
    final Properties props = new Properties();
    // mark it for scoped EJB client context
    props.put("org.jboss.ejb.client.scoped.context", "true");
    // add other properties
    props.put(...);
    ...
    Context ejbRootNamingContext = (Context) new InitialContext(props).lookup("ejb:");
    final MyBean bean = ejbRootNamingContext.lookup("app/module/distinct/bean!interface"); //
    rest of the EJB jndi string
    return bean;
}
```

As you can see, the `doEJBInvocation()` method first calls a `lookupEJB()` method which does a lookup of the bean using a JNDI context and then returns the bean (proxy). The `doEJBInvocation()` then uses that returned proxy and keeps doing the invocations on the bean. As you might have noticed, the JNDI context that was used for lookup (i.e. the `ejbRootNamingContext`) is eligible for GC. If the EJB client API had closed the scoped EJB client context associated with that JNDI context, when that JNDI context was garbage collected, then the subsequent EJB invocations on the returned EJB (proxy) would start failing in `doEJBInvocation()` since the EJB client context is no longer available.

That's the reason why the EJB client API doesn't automatically close the EJB client context.



33 Spring applications development and migration guide

This document details the main points that need to be considered by Spring developers that wish to develop new applications or to migrate existing applications to be run into WildFly 8.

33.1 Dependencies and Modularity

WildFly 8 has a modular class loading strategy, different from previous versions of JBoss AS, which enforces a better class loading isolation between deployments and the application server itself. A detailed description can be found in the documentation dedicated to [class loading in WildFly 8](#).

This reduces significantly the risk of running into a class loading conflict and allows applications to package their own dependencies if they choose to do so. This makes it easier for Spring applications that package their own dependencies - such as logging frameworks or persistence providers to run on WildFly 8.

At the same time, this does not mean that duplications and conflicts cannot exist on the classpath. Some module dependencies are implicit, depending on the type of deployment as shown [here](#).

33.2 Persistence usage guide

Depending on the strategy being used, Spring applications can be:

- native Hibernate applications;
- JPA-based applications;
- native JDBC applications;

33.3 Native Spring/Hibernate applications

Applications that use the Hibernate API directly with Spring (i.e. through either one of `LocalSessionFactoryBean` or `AnnotationSessionFactoryBean`) may use a version of Hibernate 3 packaged inside the application. Hibernate 4 (which is provided through the 'org.hibernate' module of WildFly 8) is not supported by Spring 3.0 and Spring 3.1 (and may be supported by Spring 3.2 as described in [SPR-8096](#)), so adding this module as a dependency is not a solution.

33.4 based applications

Spring applications using JPA may choose between:

- using a server-deployed persistence unit;
- using a Spring-managed persistence unit.



33.4.1 Using server-deployed persistence units

Applications that use a server-deployed persistence unit must observe the typical Java EE rules in what concerns dependency management, i.e. the `javax.persistence` classes and persistence provider (Hibernate) are contained in modules which are added automatically by the application when the persistence unit is deployed.

In order to use the server-deployed persistence units from within Spring, either the persistence context or the persistence unit need to be registered in JNDI via `web.xml` as follows:

```
<persistence-context-ref>
  <persistence-context-ref-name>persistence/petclinic-em</persistence-unit-ref-name>
  <persistence-unit-name>petclinic</persistence-unit-name>
</persistence-context-ref>
```

or, respectively:

```
<persistence-unit-ref>
  <persistence-unit-ref-name>persistence/petclinic-emf</persistence-unit-ref-name>
  <persistence-unit-name>petclinic</persistence-unit-name>
</persistence-unit-ref>
```

When doing so, the persistence context or persistence unit are available to be looked up in JNDI, as follows:

```
<jee:jndi-lookup id="entityManager" jndi-name="java:comp/env/persistence/petclinic-em"
  expected-type="javax.persistence.EntityManager" />
```

or

```
<jee:jndi-lookup id="entityManagerFactory" jndi-name="java:comp/env/persistence/petclinic-emf"
  expected-type="javax.persistence.EntityManagerFactory" />
```

JNDI binding

JNDI binding via `persistence.xml` properties is not supported in WildFly 8.



33.4.2 Using Spring-managed persistence units

Spring applications running in WildFly 8 may also create persistence units on their own, using the `LocalContainerEntityManagerFactoryBean`. This is what these applications need to consider:

Placement of the persistence unit definitions

When the application server encounters a deployment that has a file named `META-INF/persistence.xml` (or, for that matter, `WEB-INF/classes/META-INF/persistence.xml`), it will attempt to create a persistence unit based on what is provided in the file. In most cases, such definition files are not compliant with the Java EE requirements, mostly because required elements such as the `datasource` of the persistence unit are supposed to be provided by the Spring context definitions, which will fail the deployment of the persistence unit, and consequently of the entire deployment.

Spring applications can easily avoid this type of conflict, by using a feature of the `LocalContainerEntityManagerFactoryBean` which is designed for this purpose. Persistence unit definition files can exist in other locations than `META-INF/persistence.xml` and the location can be indicated through the `persistenceXmlLocation` property of the factory bean class.

Assuming that the persistence unit is in the `META-INF/jpa-persistence.xml`, the corresponding definition can be:

```
<bean id="entityManagerFactory"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
    <property name="persistenceXmlLocation"
value="classpath*:META-INF/jpa-persistence.xml"/>
    <!-- other definitions -->
</bean>
```



33.4.3 Managing dependencies

Since the `LocalContainerEntityManagerFactoryBean` and the corresponding `HibernateJpaVendorAdapter` are based on Hibernate 3, it is required to use that version with the application. Therefore, the Hibernate 3 jars must be included in the deployment. At the same time, due the presence of `@PersistenceUnit` or `@PersistenceContext` annotations on the application classes, the application server will automatically add the 'org.hibernate' module as a dependency.

This can be avoided by instructing the server to exclude the module from the deployment's list of dependencies. In order to do so, include a `META-INF/jboss-deployment-structure.xml` or, for web applications, `WEB-INF/jboss-deployment-structure.xml` with the following content:

```
<jboss-deployment-structure xmlns="urn:jboss:deployment-structure:1.0">
  <deployment>
    <exclusions>
      <module name="org.hibernate" />
    </exclusions>
  </deployment>
</jboss-deployment-structure>
```



34 Sharing sessions between wars in an ear

Undertow allows you to share sessions between wars in an ear, if it is explicitly configured to do so. Note that if you use this feature your applications may not be portable, as this is not a standard servlet feature.

In order to enable this you must include a `shared-session-config` element in the `jboss-all.xml` file in the META-INF directory of the ear:

```
<jboss xmlns="urn:jboss:1.0">
  <shared-session-config xmlns="urn:jboss:shared-session-config:1.0">
    <session-config>
      <cookie-config>
        <path>/</path>
      </cookie-config>
    </session-config>
  </shared-session-config>
</jboss>
```

This element is used to configure the shared session manager that will be used by all wars in the ear. For full details of all the options provided by this file please see the schema at

https://github.com/wildfly/wildfly/blob/master/undertow/src/main/resources/schema/shared-session-config_1_0.xsd, however in general it mimics the options that are available in `jboss-web.xml` for configuring the session.



35 Webservices reference guide

The Web Services functionalities of WildFly are provided by the JBossWS project integration.

The latest project documentation is available [here](#).

This section covers the most relevant topics for the JBossWS version available on WildFly 9.

- [JAX-WS User Guide](#)
- [JAX-WS Tools](#)
 - [wsconsume](#)
 - [wsprovide](#)
- [Advanced User Guide](#)
 - [Predefined client and endpoint configurations](#)
 - [Authentication](#)
 - [Apache CXF integration](#)
 - [WS-Addressing](#)
 - [WS-Security](#)
 - [WS-Trust and STS](#)
 - [ActAs WS-Trust Scenario](#)
 - [OnBehalfOf WS-Trust Scenario](#)
 - [SAML Bearer Assertion Scenario](#)
 - [SAML Holder-Of-Key Assertion Scenario](#)
 - [WS-Reliable Messaging](#)
 - [SOAP over JMS](#)
 - [HTTP Proxy](#)
 - [WS-Discovery](#)
 - [WS-Policy](#)
 - [Published WSDL customization](#)
- [JBoss Modules and WS applications](#)

35.1 WS User Guide

The [Java API for XML-Based Web Services \(JAX-WS / JSR-224\)](#) defines the mapping between WSDL and Java as well as the classes to be used for accessing webservices and publishing them. JBossWS implements the latest JAX-WS specification, hence users can reference it for any vendor agnostic webservice usage need. Below is a brief overview of the most basic functionalities.



35.1.1 Web Service Endpoints

JAX-WS simplifies the development model for a web service endpoint a great deal. In short, an endpoint implementation bean is annotated with JAX-WS annotations and deployed to the server. The server automatically generates and publishes the abstract contract (i.e. wsdl+schema) for client consumption. All marshalling/unmarshalling is delegated to [JAXB](#).

Plain old Java Object (POJO)

Let's take a look at simple POJO endpoint implementation. All endpoint associated metadata is provided via [JSR-181](#) annotations

```
@WebService
@SOAPBinding(style = SOAPBinding.Style.RPC)
public class JSEBean01
{
    @WebMethod
    public String echo(String input)
    {
        ...
    }
}
```

The endpoint as a web application

A JAX-WS java service endpoint (JSE) is deployed as a web application. Here is a sample *web.xml* descriptor:

```
<web-app ...>
  <servlet>
    <servlet-name>TestService</servlet-name>
    <servlet-class>org.jboss.test.ws.jaxws.samples.jsr181pojo.JSEBean01</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>TestService</servlet-name>
    <url-pattern>/*</url-pattern>
  </servlet-mapping>
</web-app>
```



Packaging the endpoint

A JSR-181 java service endpoint (JSE) is packaged as a web application in a *war* file.

```
<war warfile="${build.dir}/libs/jbossws-samples-jsr181pojo.war"
webxml="${build.resources.dir}/samples/jsr181pojo/WEB-INF/web.xml">
  <classes dir="${build.dir}/classes">
    <include name="org/jboss/test/ws/samples/jsr181pojo/JSEBean01.class" />
  </classes>
</war>
```

Note, only the endpoint implementation bean and web.xml are required.

Accessing the generated WSDL

A successfully deployed service endpoint will show up in the WildFly management console. You can get the deployed endpoint wsdl address there too.



Note, it is also possible to generate the abstract contract off line using JBossWS tools. For details of that please see Bottom-Up (Java to WSDL).



EJB3 Stateless Session Bean (SLSB)

The JAX-WS programming model supports the same set of annotations on EJB3 stateless session beans as on POJO endpoints.

```
@Stateless
@Remote(EJB3RemoteInterface.class)
@RemoteBinding(jndiBinding = "/ejb3/EJB3EndpointInterface")

@WebService
@SOAPBinding(style = SOAPBinding.Style.RPC)
public class EJB3Bean01 implements EJB3RemoteInterface
{
    @WebMethod
    public String echo(String input)
    {
        ...
    }
}
```

Above you see an EJB-3.0 stateless session bean that exposes one method both on the remote interface and as an endpoint operation.

Packaging the endpoint

A JSR-181 EJB service endpoint is packaged as an ordinary ejb deployment.

```
<jar jarfile="${build.dir}/libs/jbossws-samples-jsr181ejb.jar">
  <fileset dir="${build.dir}/classes">
    <include name="org/jboss/test/ws/samples/jsr181ejb/EJB3Bean01.class"/>
    <include name="org/jboss/test/ws/samples/jsr181ejb/EJB3RemoteInterface.class"/>
  </fileset>
</jar>
```

Accessing the generated WSDL

A successfully deployed service endpoint will show up in the WildFly management console. You can get the deployed endpoint wsdl address there too.

i Note, it is also possible to generate the abstract contract off line using JBossWS tools. For details of that please see Bottom-Up (Java to WSDL).



Endpoint Provider

JAX-WS services typically implement a native Java service endpoint interface (SEI), perhaps mapped from a WSDL port type, either directly or via the use of annotations.

Java SEIs provide a high level Java-centric abstraction that hides the details of converting between Java objects and their XML representations for use in XML-based messages. However, in some cases it is desirable for services to be able to operate at the XML message level. The Provider interface offers an alternative to SEIs and may be implemented by services wishing to work at the XML message level.

A Provider based service instances invoke method is called for each message received for the service.

```
@WebServiceProvider(wsdlLocation = "WEB-INF/wsdl/Provider.wsdl")
@ServiceMode(value = Service.Mode.PAYLOAD)
public class ProviderBeanPayload implements Provider<Source>
{
    public Source invoke(Source req)
    {
        // Access the entire request PAYLOAD and return the response PAYLOAD
    }
}
```

Note, `Service.Mode.PAYLOAD` is the default and does not have to be declared explicitly. You can also use `Service.Mode.MESSAGE` to access the entire SOAP message (i.e. with `MESSAGE` the Provider can also see SOAP Headers)

The abstract contract for a provider endpoint cannot be derived/generated automatically. Therefore it is necessary to specify the `wsdlLocation` with the `@WebServiceProvider` annotation.

35.1.2 Web Service Clients

Service

`Service` is an abstraction that represents a WSDL service. A WSDL service is a collection of related ports, each of which consists of a port type bound to a particular protocol and available at a particular endpoint address.

For most clients, you will start with a set of stubs generated from the WSDL. One of these will be the service, and you will create objects of that class in order to work with the service (see "static case" below).



Service Usage

Static case

Most clients will start with a WSDL file, and generate some stubs using JBossWS tools like *wsconsume*. This usually gives a mass of files, one of which is the top of the tree. This is the service implementation class.

The generated implementation class can be recognised as it will have two public constructors, one with no arguments and one with two arguments, representing the wsdl location (a `java.net.URL`) and the service name (a `javax.xml.namespace.QName`) respectively.

Usually you will use the no-argument constructor. In this case the WSDL location and service name are those found in the WSDL. These are set implicitly from the `@WebServiceClient` annotation that decorates the generated class.

The following code snippet shows the generated constructors from the generated class:

```
// Generated Service Class

@WebServiceClient(name="StockQuoteService", targetNamespace="http://example.com/stocks",
wsdlLocation="http://example.com/stocks.wsdl")
public class StockQuoteService extends javax.xml.ws.Service
{
    public StockQuoteService()
    {
        super(new URL("http://example.com/stocks.wsdl"), new QName("http://example.com/stocks",
"StockQuoteService"));
    }

    public StockQuoteService(String wsdlLocation, QName serviceName)
    {
        super(wsdlLocation, serviceName);
    }

    ...
}
```

Section [Dynamic Proxy](#) explains how to obtain a port from the service and how to invoke an operation on the port. If you need to work with the XML payload directly or with the XML representation of the entire SOAP message, have a look at [Dispatch](#).

Dynamic case

In the dynamic case, when nothing is generated, a web service client uses `Service.create` to create Service instances, the following code illustrates this process.

```
URL wsdlLocation = new URL("http://example.org/my.wsdl");
QName serviceName = new QName("http://example.org/sample", "MyService");
Service service = Service.create(wsdlLocation, serviceName);
```



Handler Resolver

JAX-WS provides a flexible plug-in framework for message processing modules, known as handlers, that may be used to extend the capabilities of a JAX-WS runtime system. Handler Framework describes the handler framework in detail. A `Service` instance provides access to a `HandlerResolver` via a pair of `getHandlerResolver` / `setHandlerResolver` methods that may be used to configure a set of handlers on a per-service, per-port or per-protocol binding basis.

When a `Service` instance is used to create a proxy or a `Dispatch` instance then the handler resolver currently registered with the service is used to create the required handler chain. Subsequent changes to the handler resolver configured for a `Service` instance do not affect the handlers on previously created proxies, or `Dispatch` instances.

Executor

`Service` instances can be configured with a `java.util.concurrent.Executor`. The executor will then be used to invoke any asynchronous callbacks requested by the application. The `setExecutor` and `getExecutor` methods of `Service` can be used to modify and retrieve the executor configured for a service.

Dynamic Proxy

You can create an instance of a client proxy using one of `getPort` methods on the `Service`.

```
/**
 * The getPort method returns a proxy. A service client
 * uses this proxy to invoke operations on the target
 * service endpoint. The serviceEndpointInterface
 * specifies the service endpoint interface that is supported by
 * the created dynamic proxy instance.
 */
public <T> T getPort(QName portName, Class<T> serviceEndpointInterface)
{
    ...
}

/**
 * The getPort method returns a proxy. The parameter
 * serviceEndpointInterface specifies the service
 * endpoint interface that is supported by the returned proxy.
 * In the implementation of this method, the JAX-WS
 * runtime system takes the responsibility of selecting a protocol
 * binding (and a port) and configuring the proxy accordingly.
 * The returned proxy should not be reconfigured by the client.
 */
public <T> T getPort(Class<T> serviceEndpointInterface)
{
    ...
}
```



The service endpoint interface (SEI) is usually generated using tools. For details see [Top Down \(WSDL to Java\)](#)

A generated static Service usually also offers typed methods to get ports. These methods also return dynamic proxies that implement the SEI.

```
@WebServiceClient(name = "TestEndpointService", targetNamespace = "http://org.jboss.ws/wsref",
    wsdlLocation = "http://localhost.localdomain:8080/jaxws-samples-webserviceref?wsdl")

public class TestEndpointService extends Service
{
    ...

    public TestEndpointService(URL wsdlLocation, QName serviceName) {
        super(wsdlLocation, serviceName);
    }

    @WebEndpoint(name = "TestEndpointPort")
    public TestEndpoint getTestEndpointPort()
    {
        return (TestEndpoint)super.getPort(TESTENDPOINTPORT, TestEndpoint.class);
    }
}
```

WebServiceRef

The `@WebServiceRef` annotation is used to declare a reference to a Web service. It follows the resource pattern exemplified by the `javax.annotation.Resource` annotation in [JSR-250](#).

There are two uses to the `WebServiceRef` annotation:

1. To define a reference whose type is a generated service class. In this case, the type and value element will both refer to the generated service class type. Moreover, if the reference type can be inferred by the field/method declaration the annotation is applied to, the type and value elements MAY have the default value (`Object.class`, that is). If the type cannot be inferred, then at least the type element MUST be present with a non-default value.
2. To define a reference whose type is a SEI. In this case, the type element MAY be present with its default value if the type of the reference can be inferred from the annotated field/method declaration, but the value element MUST always be present and refer to a generated service class type (a subtype of `javax.xml.ws.Service`). The `wsdlLocation` element, if present, overrides the WSDL location information specified in the `WebService` annotation of the referenced generated service class.

```
public class EJB3Client implements EJB3Remote
{
    @WebServiceRef
    public TestEndpointService service4;

    @WebServiceRef
    public TestEndpoint port3;
```




Dispatch

XMLWeb Services use XML messages for communication between services and service clients. The higher level JAX-WS APIs are designed to hide the details of converting between Java method invocations and the corresponding XML messages, but in some cases operating at the XML message level is desirable. The Dispatch interface provides support for this mode of interaction.

Dispatch supports two usage modes, identified by the constants

`javax.xml.ws.Service.Mode.MESSAGE` and `javax.xml.ws.Service.Mode.PAYLOAD` respectively:

Message In this mode, client applications work directly with protocol-specific message structures. E.g., when used with a SOAP protocol binding, a client application would work directly with a SOAP message.

Message Payload In this mode, client applications work with the payload of messages rather than the messages themselves. E.g., when used with a SOAP protocol binding, a client application would work with the contents of the SOAP Body rather than the SOAP message as a whole.

Dispatch is a low level API that requires clients to construct messages or message payloads as XML and requires an intimate knowledge of the desired message or payload structure. Dispatch is a generic class that supports input and output of messages or message payloads of any type.

```
Service service = Service.create(wsdlURL, serviceName);
Dispatch dispatch = service.createDispatch(portName, StreamSource.class, Mode.PAYLOAD);

String payload = "<ns1:ping xmlns:ns1='http://oneway.samples.jaxws.ws.test.jboss.org/'/>";
dispatch.invokeOneWay(new StreamSource(new StringReader(payload)));

payload = "<ns1:feedback xmlns:ns1='http://oneway.samples.jaxws.ws.test.jboss.org/'/>";
Source retObj = (Source)dispatch.invoke(new StreamSource(new StringReader(payload)));
```



Asynchronous Invocations

The `BindingProvider` interface represents a component that provides a protocol binding for use by clients, it is implemented by proxies and is extended by the `Dispatch` interface.

`BindingProvider` instances may provide asynchronous operation capabilities. When used, asynchronous operation invocations are decoupled from the `BindingProvider` instance at invocation time such that the response context is not updated when the operation completes. Instead a separate response context is made available using the `Response` interface.

```
public void testInvokeAsync() throws Exception
{
    URL wsdlURL = new URL("http://" + getServerHost() + ":8080/jaxws-samples-asynchronous?wsdl");
    QName serviceName = new QName(targetNS, "TestEndpointService");
    Service service = Service.create(wsdlURL, serviceName);
    TestEndpoint port = service.getPort(TestEndpoint.class);
    Response response = port.echoAsync("Async");
    // access future
    String retStr = (String) response.get();
    assertEquals("Async", retStr);
}
```

Oneway Invocations

`@Oneway` indicates that the given web method has only an input message and no output. Typically, a oneway method returns the thread of control to the calling application prior to executing the actual business method.

```
@WebService (name="PingEndpoint")
@SOAPBinding(style = SOAPBinding.Style.RPC)
public class PingEndpointImpl
{
    private static String feedback;

    @WebMethod
    @Oneway
    public void ping()
    {
        log.info("ping");
        feedback = "ok";
    }

    @WebMethod
    public String feedback()
    {
        log.info("feedback");
        return feedback;
    }
}
```



Timeout Configuration

There are two properties to configure the http connection timeout and client receive time out:

```
public void testConfigureTimeout() throws Exception
{
    //Set timeout until a connection is established
    ((BindingProvider)port).getRequestContext().put("javax.xml.ws.client.connectionTimeout",
"6000");

    //Set timeout until the response is received
    ((BindingProvider) port).getRequestContext().put("javax.xml.ws.client.receiveTimeout",
"1000");

    port.echo("testTimeout");
}
```

35.1.3 Common API

This sections describes concepts that apply equally to Web Service Endpoints and Web Service Clients.

Handler Framework

The handler framework is implemented by a JAX-WS protocol binding in both client and server side runtimes. Proxies, and Dispatch instances, known collectively as binding providers, each use protocol bindings to bind their abstract functionality to specific protocols.

Client and server-side handlers are organized into an ordered list known as a handler chain. The handlers within a handler chain are invoked each time a message is sent or received. Inbound messages are processed by handlers prior to binding provider processing. Outbound messages are processed by handlers after any binding provider processing.

Handlers are invoked with a message context that provides methods to access and modify inbound and outbound messages and to manage a set of properties. Message context properties may be used to facilitate communication between individual handlers and between handlers and client and service implementations. Different types of handlers are invoked with different types of message context.

Logical Handler

Handlers that only operate on message context properties and message payloads. Logical handlers are protocol agnostic and are unable to affect protocol specific parts of a message. Logical handlers are handlers that implement `javax.xml.ws.handler.LogicalHandler`.



Protocol Handler

Handlers that operate on message context properties and protocol specific messages. Protocol handlers are specific to a particular protocol and may access and change protocol specific aspects of a message.

Protocol handlers are handlers that implement any interface derived from

`javax.xml.ws.handler.Handler` except `javax.xml.ws.handler.LogicalHandler`.

Service endpoint handlers

On the service endpoint, handlers are defined using the `@HandlerChain` annotation.

```
@WebService
@HandlerChain(file = "jaxws-server-source-handlers.xml")
public class SOAPEndpointSourceImpl
{
    ...
}
```

The location of the handler chain file supports 2 formats

1. An absolute `java.net.URL` in externalForm. (ex: <http://myhandlers.foo.com/handlerfile1.xml>)
2. A relative path from the source file or class file. (ex: `bar/handlerfile1.xml`)

Service client handlers

On the client side, handler can be configured using the `@HandlerChain` annotation on the SEI or dynamically using the API.

```
Service service = Service.create(wsdlURL, serviceName);
Endpoint port = (Endpoint)service.getPort(Endpoint.class);

BindingProvider bindingProvider = (BindingProvider)port;
List<Handler> handlerChain = new ArrayList<Handler>();
handlerChain.add(new LogHandler());
handlerChain.add(new AuthorizationHandler());
handlerChain.add(new RoutingHandler());
bindingProvider.getBinding().setHandlerChain(handlerChain); // important!
```



Message Context

`MessageContext` is the super interface for all JAX-WS message contexts. It extends `Map<String, Object>` with additional methods and constants to manage a set of properties that enable handlers in a handler chain to share processing related state. For example, a handler may use the `put` method to insert a property in the message context that one or more other handlers in the handler chain may subsequently obtain via the `get` method.

Properties are scoped as either `APPLICATION` or `HANDLER`. All properties are available to all handlers for an instance of an MEP on a particular endpoint. E.g., if a logical handler puts a property in the message context, that property will also be available to any protocol handlers in the chain during the execution of an MEP instance. `APPLICATION` scoped properties are also made available to client applications (see section 4.2.1) and service endpoint implementations. The default scope for a property is `HANDLER`.

Logical Message Context

Logical Handlers are passed a message context of type `LogicalMessageContext` when invoked. `LogicalMessageContext` extends `MessageContext` with methods to obtain and modify the message payload, it does not provide access to the protocol specific aspects of a message. A protocol binding defines what component of a message are available via a logical message context. The SOAP binding defines that a logical handler deployed in a SOAP binding can access the contents of the SOAP body but not the SOAP headers whereas the XML/HTTP binding defines that a logical handler can access the entire XML payload of a message.

SOAP Message Context

SOAP handlers are passed a `SOAPMessageContext` when invoked. `SOAPMessageContext` extends `MessageContext` with methods to obtain and modify the SOAP message payload.



Fault Handling

An implementation may throw a `SOAPFaultException`

```
public void throwSoapFaultException()
{
    SOAPFactory factory = SOAPFactory.newInstance();
    SOAPFault fault = factory.createFault("this is a fault string!", new QName("http://foo",
"FooCode"));
    fault.setFaultActor("mr. actor");
    fault.addDetail().addChildElement("test");
    throw new SOAPFaultException(fault);
}
```

or an application specific user exception

```
public void throwApplicationException() throws UserException
{
    throw new UserException("validation", 123, "Some validation error");
}
```



In case of the latter, JBossWS generates the required fault wrapper beans at runtime if they are not part of the deployment

35.1.4 WS Annotations

For details, see [JSR-224 - Java API for XML-Based Web Services \(JAX-WS\) 2.2](#)

javax.xml.ws.ServiceMode

The `ServiceMode` annotation is used to specify the mode for a provider class, i.e. whether a provider wants to have access to protocol message payloads (e.g. a SOAP body) or the entire protocol messages (e.g. a SOAP envelope).

javax.xml.ws.WebFault

The `WebFault` annotation is used when mapping WSDL faults to Java exceptions, see section 2.5. It is used to capture the name of the fault element used when marshalling the JAXB type generated from the global element referenced by the WSDL fault message. It can also be used to customize the mapping of service specific exceptions to WSDL faults.



javax.xml.ws.RequestWrapper

The `RequestWrapper` annotation is applied to the methods of an SEI. It is used to capture the JAXB generated request wrapper bean and the element name and namespace for marshalling / unmarshalling the bean. The default value of `localName` element is the `operationName` as defined in `webMethod` annotation and the default value for the `targetNamespace` element is the target namespace of the SEI. When starting from Java, this annotation is used to resolve overloading conflicts in document literal mode. Only the `className` element is required in this case.

javax.xml.ws.ResponseWrapper

The `ResponseWrapper` annotation is applied to the methods of an SEI. It is used to capture the JAXB generated response wrapper bean and the element name and namespace for marshalling / unmarshalling the bean. The default value of the `localName` element is the `operationName` as defined in the `webMethod` appended with "Response" and the default value of the `targetNamespace` element is the target namespace of the SEI. When starting from Java, this annotation is used to resolve overloading conflicts in document literal mode. Only the `className` element is required in this case.

javax.xml.ws.WebServiceClient

The `WebServiceClient` annotation is specified on a generated service class (see 2.7). It is used to associate a class with a specific Web service, identify by a URL to a WSDL document and the qualified name of a `wsdl:service` element.

javax.xml.ws.WebEndpoint

The `WebEndpoint` annotation is specified on the `getPortName()` methods of a generated service class (see 2.7). It is used to associate a get method with a specific `wsdl:port`, identified by its local name (a `NCName`).

javax.xml.ws.WebServiceProvider

The `WebServiceProvider` annotation is specified on classes that implement a strongly typed `javax.xml.ws.Provider`. It is used to declare that a class that satisfies the requirements for a provider (see 5.1) does indeed define a Web service endpoint, much like the `WebService` annotation does for SEI-based endpoints.

The `WebServiceProvider` and `WebService` annotations are mutually exclusive.

javax.xml.ws.BindingType

The `BindingType` annotation is applied to an endpoint implementation class. It specifies the binding to use when publishing an endpoint of this type.

The default binding for an endpoint is the SOAP 1.1/HTTP one.



javax.xml.ws.WebServiceRef

The `WebServiceRef` annotation is used to declare a reference to a Web service. It follows the resource pattern exemplified by the `javax.annotation.Resource` annotation in JSR-250 [JBWS:32]. The `WebServiceRef` annotation is required to be honored when running on the Java EE 5 platform, where it is subject to the common resource injection rules described by the platform specification [JBWS:33].

javax.xml.ws.WebServiceRefs

The `WebServiceRefs` annotation is used to declare multiple references to Web services on a single class. It is necessary to work around the limitation against specifying repeated annotations of the same type on any given class, which prevents listing multiple `javax.xml.ws.WebServiceRef` annotations one after the other. This annotation follows the resource pattern exemplified by the `javax.annotation.Resources` annotation in JSR-250.

Since no name and type can be inferred in this case, each `WebServiceRef` annotation inside a `WebServiceRefs` MUST contain name and type elements with non-default values. The `WebServiceRef` annotation is required to be honored when running on the Java EE 5 platform, where it is subject to the common resource injection rules described by the platform specification.

javax.xml.ws.Action

The `Action` annotation is applied to the methods of a SEI. It used to generate the `wsa:Action` on `wsdl:input` and `wsdl:output` of each `wsdl:operation` mapped from the annotated methods.

javax.xml.ws.FaultAction

The `FaultAction` annotation is used within the `Action` annotation to generate the `wsa:Action` element on the `wsdl:fault` element of each `wsdl:operation` mapped from the annotated methods.

35.1.5 181 Annotations

JSR-181 defines the syntax and semantics of Java Web Service (JWS) metadata and default values.

For details, see [JSR 181 - Web Services Metadata for the Java Platform](#).

javax.jws.WebService

Marks a Java class as implementing a Web Service, or a Java interface as defining a Web Service interface.

javax.jws.WebMethod

Customizes a method that is exposed as a Web Service operation.



javax.jws.OneWay

Indicates that the given web method has only an input message and no output. Typically, a oneway method returns the thread of control to the calling application prior to executing the actual business method. A JSR-181 processor is REQUIRED to report an error if an operation marked `@Oneway` has a return value, declares any checked exceptions or has any INOUT or OUT parameters.

javax.jws.WebParam

Customizes the mapping of an individual parameter to a Web Service message part and XML element.

javax.jws.WebResult

Customizes the mapping of the return value to a WSDL part and XML element.

javax.jws.SOAPBinding

Specifies the mapping of the Web Service onto the SOAP message protocol.

The `SOAPBinding` annotation has a target of `TYPE` and `METHOD`. The annotation may be placed on a method if and only if the `SOAPBinding.style` is `DOCUMENT`. Implementations MUST report an error if the `SOAPBinding` annotation is placed on a method with a `SOAPBinding.style` of `RPC`. Methods that do not have a `SOAPBinding` annotation accept the `SOAPBinding` behavior defined on the type.

javax.jws.HandlerChain

The `@HandlerChain` annotation associates the Web Service with an externally defined handler chain.

It is an error to combine this annotation with the `@SOAPMessageHandlers` annotation.

The `@HandlerChain` annotation MAY be present on the endpoint interface and service implementation bean. The service implementation bean's `@HandlerChain` is used if `@HandlerChain` is present on both.

The `@HandlerChain` annotation MAY be specified on the type only. The annotation target includes `METHOD` and `FIELD` for use by JAX-WS-2.x.

35.2 WS Tools

The JAX-WS tools provided by JBossWS can be used in a variety of ways. First we will look at server-side development strategies, and then proceed to the client.



35.2.1 Server side

When developing a Web Service Endpoint (the server-side) you have the option of starting from Java (*bottom-up development*), or from the abstract contract (WSDL) that defines your service (*top-down development*). If this is a new service (no existing contract), the bottom-up approach is the fastest route; you only need to add a few annotations to your classes to get a service up and running. However, if you are developing a service with an already defined contract, it is far simpler to use the top-down approach, since the provided tool will generate the annotated code for you.

Bottom-up use cases:

- Exposing an already existing EJB3 bean as a Web Service
- Providing a new service, and you want the contract to be generated for you

Top-down use cases:

- Replacing the implementation of an existing Web Service, and you can't break compatibility with older clients
- Exposing a service that conforms to a contract specified by a third party (e.g. a vendor that calls you back using an already defined protocol).
- Creating a service that adheres to the XML Schema and WSDL you developed by hand up front

The following JAX-WS command line tools are included in JBossWS:

Command	Description
wsprovide	Generates JAX-WS portable artifacts, and provides the abstract contract. Used for bottom-up development.
wsconsume	Consumes the abstract contract (WSDL and Schema files), and produces artifacts for both a server and client. Used for top-down and client development

Bottom-Up (Using wsprovide)

The bottom-up strategy involves developing the Java code for your service, and then annotating it using JAX-WS annotations. These annotations can be used to customize the contract that is generated for your service. For example, you can change the operation name to map to anything you like. However, all of the annotations have sensible defaults, so only the `@WebService` annotation is required.

This can be as simple as creating a single class:



```
package echo;

@javax.jws.WebService
public class Echo
{
    public String echo(String input)
    {
        return input;
    }
}
```

A JSE or EJB3 deployment can be built using this class, and it is the only Java code needed to deploy on JBossWS. The WSDL, and all other Java artifacts called "wrapper classes" will be generated for you at deploy time. This actually goes beyond the JAX-WS specification, which requires that wrapper classes be generated using an offline tool. The reason for this requirement is purely a vendor implementation problem, and since we do not believe in burdening a developer with a bunch of additional steps, we generate these as well. However, if you want your deployment to be portable to other application servers, you will unfortunately need to use a tool and add the generated classes to your deployment.

This is the primary purpose of the *wsprovide* tool, to generate portable JAX-WS artifacts. Additionally, it can be used to "provide" the abstract contract (WSDL file) for your service. This can be obtained by invoking *wsprovide* using the "-w" option:

```
$ javac -d . -classpath jboss-jaxws.jar Echo.java
$ wsprovide -w echo.Echo
Generating WSDL:
EchoService.wsdl
Writing Classes:
echo/jaxws/Echo.class
echo/jaxws/EchoResponse.class
```


Inspecting the WSDL reveals a service called *EchoService*:

```
<service name='EchoService'>
  <port binding='tns:EchoBinding' name='EchoPort'>
    <soap:address location='REPLACE_WITH_ACTUAL_URL' />
  </port>
</service>
```

As expected, this service defines one operation, "*echo*":

```
<portType name='Echo'>
  <operation name='echo' parameterOrder='echo'>
    <input message='tns:Echo_echo' />
    <output message='tns:Echo_echoResponse' />
  </operation>
</portType>
```



 Remember that when deploying on JBossWS you do not need to run this tool. You only need it for generating portable artifacts and/or the abstract contract for your service.

Let's create a POJO endpoint for deployment on WildFly. A simple *web.xml* needs to be created:

```
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
  version="2.4">

  <servlet>
    <servlet-name>Echo</servlet-name>
    <servlet-class>echo.Echo</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>Echo</servlet-name>
    <url-pattern>/Echo</url-pattern>
  </servlet-mapping>
</web-app>
```

The *web.xml* and the single class can now be used to create a war:

```
$ mkdir -p WEB-INF/classes
$ cp -rp echo WEB-INF/classes/
$ cp web.xml WEB-INF
$ jar cvf echo.war WEB-INF
added manifest
adding: WEB-INF/(in = 0) (out= 0)(stored 0%)
adding: WEB-INF/classes/(in = 0) (out= 0)(stored 0%)
adding: WEB-INF/classes/echo/(in = 0) (out= 0)(stored 0%)
adding: WEB-INF/classes/echo/Echo.class(in = 340) (out= 247)(deflated 27%)
adding: WEB-INF/web.xml(in = 576) (out= 271)(deflated 52%)
```

The war can then be deployed to the JBoss Application Server. The war can then be deployed to the JBoss Application Server; this will internally invoke *wsprovide*, which will generate the WSDL. If deployment was successful, and you are using the default settings, it should be available in the server management console.

For a portable JAX-WS deployment, the wrapper classes generated earlier could be added to the deployment.

Down (Using *wsconsume*)

The top-down development strategy begins with the abstract contract for the service, which includes the WSDL file and zero or more schema files. The *wsconsume* tool is then used to consume this contract, and produce annotated Java classes (and optionally sources) that define it.



`wconsume` may have problems with symlinks on Unix systems

Using the WSDL file from the bottom-up example, a new Java implementation that adheres to this service can be generated. The "-k" option is passed to `wconsume` to preserve the Java source files that are generated, instead of providing just classes:

```
$ wsconsume -k EchoService.wsdl
echo/Echo.java
echo/EchoResponse.java
echo/EchoService.java
echo/Echo_Type.java
echo/ObjectFactory.java
echo/package-info.java
echo/Echo.java
echo/EchoResponse.java
echo/EchoService.java
echo/Echo_Type.java
echo/ObjectFactory.java
echo/package-info.java
```

The following table shows the purpose of each generated file:

File	Purpose
Echo.java	Service Endpoint Interface
Echo_Type.java	Wrapper bean for request message
EchoResponse.java	Wrapper bean for response message
ObjectFactory.java	JAXB XML Registry
package-info.java	Holder for JAXB package annotations
EchoService.java	Used only by JAX-WS clients

Examining the Service Endpoint Interface reveals annotations that are more explicit than in the class written by hand in the bottom-up example, however, these evaluate to the same contract:



```
@WebService(name = "Echo", targetNamespace = "http://echo/")
public interface Echo {
    @WebMethod
    @WebResult(targetNamespace = "")
    @RequestWrapper(localName = "echo", targetNamespace = "http://echo/", className =
"echo.Echo_Type")
    @ResponseWrapper(localName = "echoResponse", targetNamespace = "http://echo/", className =
"echo.EchoResponse")
    public String echo(
        @WebParam(name = "arg0", targetNamespace = "")
        String arg0);
}
```

The only missing piece (besides for packaging) is the implementation class, which can now be written, using the above interface.

```
package echo;

@javax.jws.WebService(endpointInterface="echo.Echo")
public class EchoImpl implements Echo
{
    public String echo(String arg0)
    {
        return arg0;
    }
}
```

35.2.2 Client Side

Before going to detail on the client-side it is important to understand the decoupling concept that is central to Web Services. Web Services are not the best fit for internal RPC, even though they can be used in this way. There are much better technologies for this (CORBA, and RMI for example). Web Services were designed specifically for interoperable coarse-grained correspondence. There is no expectation or guarantee that any party participating in a Web Service interaction will be at any particular location, running on any particular OS, or written in any particular programming language. So because of this, it is important to clearly separate client and server implementations. The only thing they should have in common is the abstract contract definition. If, for whatever reason, your software does not adhere to this principal, then you should not be using Web Services. For the above reasons, the **recommended methodology for developing a client is to follow *the top-down approach***, even if the client is running on the same server.

Let's repeat the process of the top-down section, although using the deployed WSDL, instead of the one generated offline by *wsprovide*. The reason why we do this is just to get the right value for soap:address. This value must be computed at deploy time, since it is based on container configuration specifics. You could of course edit the WSDL file yourself, although you need to ensure that the path is correct.

Offline version:



```
<service name='EchoService'>
  <port binding='tns:EchoBinding' name='EchoPort'>
    <soap:address location='REPLACE_WITH_ACTUAL_URL' />
  </port>
</service>
```

Online version:

```
<service name="EchoService">
  <port binding="tns:EchoBinding" name="EchoPort">
    <soap:address location="http://localhost.localdomain:8080/echo/Echo" />
  </port>
</service>
```

Using the online deployed version with *wsconsume*:

```
$ wsconsume -k http://localhost:8080/echo/Echo?wsdl
echo/Echo.java
echo/EchoResponse.java
echo/EchoService.java
echo/Echo_Type.java
echo/ObjectFactory.java
echo/package-info.java
echo/Echo.java
echo/EchoResponse.java
echo/EchoService.java
echo/Echo_Type.java
echo/ObjectFactory.java
echo/package-info.java
```

The one class that was not examined in the top-down section, was `EchoService.java`. Notice how it stores the location the WSDL was obtained from.



```
@WebServiceClient(name = "EchoService", targetNamespace = "http://echo/", wsdlLocation =
"http://localhost:8080/echo/Echo?wsdl")
public class EchoService extends Service
{
    private final static URL ECHOSERVICE_WSDL_LOCATION;


    static {
        URL url = null;
        try
        {
            url = new URL("http://localhost:8080/echo/Echo?wsdl");
        }
        catch (MalformedURLException e)
        {
            e.printStackTrace();
        }
        ECHOSERVICE_WSDL_LOCATION = url;
    }

    public EchoService(URL wsdlLocation, QName serviceName)
    {
        super(wsdlLocation, serviceName);
    }

    public EchoService()
    {
        super(ECHOSERVICE_WSDL_LOCATION, new QName("http://echo/", "EchoService"));
    }

    @WebEndpoint(name = "EchoPort")
    public Echo getEchoPort()
    {
        return (Echo)super.getPort(new QName("http://echo/", "EchoPort"), Echo.class);
    }
}
```

As you can see, this generated class extends the main client entry point in JAX-WS, `javax.xml.ws.Service`. While you can use `Service` directly, this is far simpler since it provides the configuration info for you. The only method we really care about is the `getEchoPort()` method, which returns an instance of our Service Endpoint Interface. Any WS operation can then be called by just invoking a method on the returned interface.

 It's not recommended to refer to a remote WSDL URL in a production application. This causes network I/O every time you instantiate the Service Object. Instead, use the tool on a saved local copy, or use the URL version of the constructor to provide a new WSDL location.

All that is left to do, is write and compile the client:



```
import echo.*;

public class EchoClient
{
    public static void main(String args[])
    {
        if (args.length != 1)
        {
            System.err.println("usage: EchoClient <message>");
            System.exit(1);
        }

        EchoService service = new EchoService();
        Echo echo = service.getEchoPort();
        System.out.println("Server said: " + echo.echo(args0));
    }
}
```

It is easy to change the endpoint address of your operation at runtime, setting the `ENDPOINT_ADDRESS_PROPERTY` as shown below:

```
EchoService service = new EchoService();
Echo echo = service.getEchoPort();

/* Set NEW Endpoint Location */
String endpointURL = "http://NEW_ENDPOINT_URL";
BindingProvider bp = (BindingProvider)echo;
bp.getRequestContext().put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY, endpointURL);

System.out.println("Server said: " + echo.echo(args0));
```

35.2.3 wsconsume

`wsconsume` is a command line tool and ant task that "consumes" the abstract contract (WSDL file) and produces portable JAX-WS service and client artifacts.

Command Line Tool

The command line tool has the following usage:



```
usage: wsconsume [options] <wsdl-url>
options:
  -h, --help                Show this help message
  -b, --binding=<file>      One or more JAX-WS or JAXB binding files
  -k, --keep                Keep/Generate Java source
  -c --catalog=<file>       Oasis XML Catalog file for entity resolution
  -j --clientjar=<name>     Create a jar file of the generated artifacts for calling the
webservice
  -p --package=<name>       The target package for generated source
  -w --wsdlLocation=<loc>   Value to use for @WebServiceClient.wsdlLocation
  -o, --output=<directory> The directory to put generated artifacts
  -s, --source=<directory> The directory to put Java source
  -t, --target=<2.0|2.1|2.2> The JAX-WS specification target
  -q, --quiet               Be somewhat more quiet
  -v, --verbose             Show full exception stack traces
  -l, --load-consumer       Load the consumer and exit (debug utility)
  -e, --extension           Enable SOAP 1.2 binding extension
  -a, --additionalHeaders   Enables processing of implicit SOAP headers
  -n, --nocompile           Do not compile generated sources
```



The `wsdlLocation` is used when creating the Service to be used by clients and will be added to the `@WebServiceClient` annotation, for an endpoint implementation based on the generated service endpoint interface you will need to manually add the `wsdlLocation` to the `@WebService` annotation on your web service implementation and not the service endpoint interface.



Examples

Generate artifacts in Java class form only:

```
wsconsume Example.wsdl
```

Generate source and class files:

```
wsconsume -k Example.wsdl
```

Generate source and class files in a custom directory:

```
wsconsume -k -o custom Example.wsdl
```

Generate source and class files in the org.foo package:

```
wsconsume -k -p org.foo Example.wsdl
```

Generate source and class files using multiple binding files:

```
wsconsume -k -b wsdl-binding.xml -b schema1-binding.xml -b schema2-binding.xml
```

Maven Plugin

The `wsconsume` tools is included in the **org.jboss.ws.plugins:jaxws-tools-maven-plugin** plugin. The plugin has two goals for running the tool, `wsconsume` and `wsconsume-test`, which basically do the same during different maven build phases (the former triggers the sources generation during `generate-sources` phase, the latter during the `generate-test-sources` one).

The `wsconsume` plugin has the following parameters:



Attribute	Description	Default
bindingFiles	JAXWS or JAXB binding file	true
classpathElements	Each classpathElement provides a library file to be added to classpath	\${project.compileClasspath} or \${project.testClasspathEler}
catalog	Oasis XML Catalog file for entity resolution	none
targetPackage	The target Java package for generated code.	generated
bindingFiles	One or more JAX-WS or JAXB binding file	none
wsdlLocation	Value to use for @WebServiceClient.wsdlLocation	generated
outputDirectory	The output directory for generated artifacts.	\${project.build.outputDirect} or \${project.build.testOutputD}
sourceDirectory	The output directory for Java source.	\${project.build.directory}/ws
verbose	Enables more informational output about command progress.	false
wsdls	The WSDL files or URLs to consume	n/a
extension	Enable SOAP 1.2 binding extension.	false
encoding	The charset encoding to use for generated sources.	\${project.build.sourceEnco}
argLine	An optional additional argline to be used when running in fork mode; can be used to set endorse dir, enable debugging, etc. Example <code><argLine>-Djava.endorsed.dirs=...</argLine></code>	none
fork	Whether or not to run the generation task in a separate VM.	false
target	A preference for the JAX-WS specification target	Depends on the underlying endorsed dirs if any

Examples

You can use *wsconsume* in your own project build simply referencing the *jaxws-tools-maven-plugin* in the configured plugins in your pom.xml file.

The following example makes the plugin consume the test.wsdl file and generate SEI and wrappers' java sources. The generated sources are then compiled together with the other project classes.



```
<build>
  <plugins>
    <plugin>
      <groupId>org.jboss.ws.plugins</groupId>
      <artifactId>jaxws-tools-maven-plugin</artifactId>
      <version>1.2.0.Beta1</version>
      <configuration>
        <wsdls>
          <wsdl>${basedir}/test.wsdl</wsdl>
        </wsdls>
      </configuration>
      <executions>
        <execution>
          <goals>
            <goal>wsconsume</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

You can also specify multiple wsdl files, as well as force the target package, enable SOAP 1.2 binding and turn the tool's verbose mode on:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.jboss.ws.plugins</groupId>
      <artifactId>jaxws-tools-maven-plugin</artifactId>
      <version>1.2.0.Beta1</version>
      <configuration>
        <wsdls>
          <wsdl>${basedir}/test.wsdl</wsdl>
          <wsdl>${basedir}/test2.wsdl</wsdl>
        </wsdls>
        <targetPackage>foo.bar</targetPackage>
        <extension>true</extension>
        <verbose>true</verbose>
      </configuration>
      <executions>
        <execution>
          <goals>
            <goal>wsconsume</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

Finally, if the `wsconsume` invocation is required for consuming a wsdl to be used in your testsuite only, you might want to use the `wsconsume-test` goal as follows:



```
<build>
  <plugins>
    <plugin>
      <groupId>org.jboss.ws.plugins</groupId>
      <artifactId>jaxws-tools-maven-plugin</artifactId>
      <version>1.2.0.Beta1</version>
      <configuration>
        <wsdls>
          <wsdl>${basedir}/test.wsdl</wsdl>
        </wsdls>
      </configuration>
      <executions>
        <execution>
          <goals>
            <goal>wsconsume-test</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

Plugin stack dependency The plugin itself does not have an explicit dependency to a JBossWS stack, as it's meant for being used with implementations of any supported version of the *JBossWS SPI*. So the user is expected to set a dependency in his own `pom.xml` to the desired *JBossWS* stack version. The plugin will rely on the that for using the proper tooling.

```
<dependencies>
  <dependency>
    <groupId>org.jboss.ws.cxf</groupId>
    <artifactId>jbossws-cxf-client</artifactId>
    <version>4.0.0.GA</version>
  </dependency>
</dependencies>
```

✔ Be careful when using this plugin with the Maven War Plugin as that include any project dependency into the generated application war archive. You might want to set `<scope>provided</scope>` for the *JBossWS* stack dependency to avoid that.


ℹ Up to version 1.1.2.Final, the *artifactId* of the plugin was **maven-jaxws-tools-plugin**.


Ant Task

The *wsconsume* Ant task (*org.jboss.ws.tools.ant.WSConsumeTask*) has the following attributes:



Attribute	Description	Default
fork	Whether or not to run the generation task in a separate VM.	true
keep	Keep/Enable Java source code generation.	false
catalog	Oasis XML Catalog file for entity resolution	none
package	The target Java package for generated code.	generated
binding	A JAX-WS or JAXB binding file	none
wsdlLocation	Value to use for @WebServiceClient.wsdlLocation	generated
encoding	The charset encoding to use for generated sources	n/a
destdir	The output directory for generated artifacts.	"output"
sourcedestdir	The output directory for Java source.	value of destdir
target	The JAX-WS specification target. Allowed values are 2.0, 2.1 and 2.2	
verbose	Enables more informational output about command progress.	false
wsdl	The WSDL file or URL	n/a
extension	Enable SOAP 1.2 binding extension.	false
additionalHeaders	Enables processing of implicit SOAP headers	false

 Users also need to put `streamBuffer.jar` and `stax-ex.jar` to the classpath of the ant task to generate the appropriate artefacts.

 The `wsdlLocation` is used when creating the Service to be used by clients and will be added to the `@WebServiceClient` annotation, for an endpoint implementation based on the generated service endpoint interface you will need to manually add the `wsdlLocation` to the `@WebService` annotation on your web service implementation and not the service endpoint interface.

Also, the following nested elements are supported:

Element	Description	Default
binding	A JAXWS or JAXB binding file	none
jvmarg	Allows setting of custom jvm arguments	



Examples

Generate JAX-WS source and classes in a separate JVM with separate directories, a custom wsdl location attribute, and a list of binding files from foo.wsdl:

```
<wsconsume
  fork="true"
  verbose="true"
  destdir="output"
  sourcedestdir="gen-src"
  keep="true"
  wsdllocation="handEdited.wsdl"
  wsdl="foo.wsdl">
  <binding dir="binding-files" includes="*.xml" excludes="bad.xml" />
</wsconsume>
```

35.2.4 wsprovide

wsprovide is a command line tool, Maven plugin and Ant task that generates portable JAX-WS artifacts for a service endpoint implementation. It also has the option to "provide" the abstract contract for offline usage.



Command Line Tool

The command line tool has the following usage:

```
usage: wsprovide [options] <endpoint class name>
options:
  -h, --help                Show this help message
  -k, --keep                Keep/Generate Java source
  -w, --wsdl                Enable WSDL file generation
  -a, --address <address>  The generated port soap:address in wsdl
  -c, --classpath <path>   The classpath that contains the endpoint
  -o, --output=<directory> The directory to put generated artifacts
  -r, --resource=<directory> The directory to put resource artifacts
  -s, --source=<directory> The directory to put Java source
  -e, --extension           Enable SOAP 1.2 binding extension
  -q, --quiet               Be somewhat more quiet
  -t, --show-traces         Show full exception stack traces
```

Examples

Generating wrapper classes for portable artifacts in the "generated" directory:

```
wsprovide -o generated foo.Endpoint
```

Generating wrapper classes and WSDL in the "generated" directory

```
wsprovide -o generated -w foo.Endpoint
```

Using an endpoint that references other jars

```
wsprovide -o generated -c application1.jar:application2.jar foo.Endpoint
```

Maven Plugin

The *wsprovide* tools is included in the **org.jboss.ws.plugins:jaxws-tools-maven-plugin** plugin. The plugin has two goals for running the tool, *wsprovide* and *wsprovide-test*, which basically do the same during different Maven build phases (the former triggers the sources generation during *process-classes* phase, the latter during the *process-test-classes* one).

The *wsprovide* plugin has the following parameters:



Attribute	Description	Default
testClasspathElements	Each classpathElement provides a library file to be added to classpath	<code>\${project.compileClasspathElements}</code> or <code>\${project.testClasspathElements}</code>
outputDirectory	The output directory for generated artifacts.	<code>\${project.build.outputDirectory}</code> or <code>\${project.build.testOutputDirectory}</code>
resourceDirectory	The output directory for resource artifacts (WSDL/XSD).	<code>\${project.build.directory}/wsprovide/resources</code>
sourceDirectory	The output directory for Java source.	<code>\${project.build.directory}/wsprovide/java</code>
extension	Enable SOAP 1.2 binding extension.	false
generateWsdI	Whether or not to generate WSDL.	false
verbose	Enables more informational output about command progress.	false
portSoapAddress	The generated port soap:address in the WSDL	
endpointClass	Service Endpoint Implementation.	

Examples

You can use *wsprovide* in your own project build simply referencing the *maven-jaxws-tools-plugin* in the configured plugins in your *pom.xml* file.

The following example makes the plugin provide the wsdl file and artifact sources for the specified endpoint class:



```
<build>
  <plugins>
    <plugin>
      <groupId>org.jboss.ws.plugins</groupId>
      <artifactId>jaxws-tools-maven-plugin</artifactId>
      <version>1.2.0.Beta1</version>
      <configuration>
        <verbose>true</verbose>
        <endpointClass>org.jboss.test.ws.plugins.tools.wsprovide.TestEndpoint</endpointClass>
        <generateWsdL>true</generateWsdL>
      </configuration>
      <executions>
        <execution>
          <goals>
            <goal>wsprovide</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

The following example does the same, but is meant for use in your own testsuite:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.jboss.ws.plugins</groupId>
      <artifactId>jaxws-tools-maven-plugin</artifactId>
      <version>1.2.0.Beta1</version>
      <configuration>
        <verbose>true</verbose>
        <endpointClass>org.jboss.test.ws.plugins.tools.wsprovide.TestEndpoint2</endpointClass>
        <generateWsdL>true</generateWsdL>
      </configuration>
      <executions>
        <execution>
          <goals>
            <goal>wsprovide-test</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

Plugin stack dependencyThe plugin itself does not have an explicit dependency to a JBossWS stack, as it's meant for being used with implementations of any supported version of the *JBossWS SPI*. So the user is expected to set a dependency in his own `pom.xml` to the desired *JBossWS* stack version. The plugin will rely on the that for using the proper tooling.



```
<dependencies>
  <dependency>
    <groupId>org.jboss.ws.cxf</groupId>
    <artifactId>jbossws-cxf-client</artifactId>
    <version>5.0.0.CR1</version>
  </dependency>
</dependencies>
```

✔ Be careful when using this plugin with the Maven War Plugin as that include any project dependency into the generated application war archive. You might want to set `<scope>provided</scope>` for the *JBossWS* stack dependency to avoid that.

ℹ Up to version 1.1.2.Final, the *artifactId* of the plugin was **maven-jaxws-tools-plugin**.



Ant Task

The wsprovide ant task (*org.jboss.ws.tools.ant.WSProvideTask*) has the following attributes:

Attribute	Description	Default
fork	Whether or not to run the generation task in a separate VM.	true
keep	Keep/Enable Java source code generation.	false
destdir	The output directory for generated artifacts.	"output"
resourcedestdir	The output directory for resource artifacts (WSDL/XSD).	value of destdir
sourcedestdir	The output directory for Java source.	value of destdir
extension	Enable SOAP 1.2 binding extension.	false
genwsdl	Whether or not to generate WSDL.	false
address	The generated port soap:address in wsdl.	
verbose	Enables more informational output about command progress.	false
sei	Service Endpoint Implementation.	
classpath	The classpath that contains the service endpoint implementation.	."

Examples

Executing wsprovide in verbose mode with separate output directories for source, resources, and classes:

```
<target name="test-wsprovide" depends="init">
  <taskdef name="wsprovide" classname="org.jboss.ws.tools.ant.WSProvideTask">
    <classpath refid="core.classpath"/>
  </taskdef>
  <wsprovide
    fork="false"
    keep="true"
    destdir="out"
    resourcedestdir="out-resource"
    sourcedestdir="out-source"
    genwsdl="true"
    verbose="true"
    sei="org.jboss.test.ws.jaxws.jsr181.soapbinding.DocWrappedServiceImpl">
    <classpath>
      <pathelement path="${tests.output.dir}/classes"/>
    </classpath>
  </wsprovide>
</target>
```



35.3 Advanced User Guide

- [Logging](#)
 - [JAX-WS Handler approach](#)
 - [Apache CXF approach](#)
 - [System property](#)
 - [Manual interceptor addition and logging feature](#)
- [WS-* support](#)
- [Address rewrite](#)
 - [Server configuration options](#)
 - [Dynamic rewrite](#)
- [Configuration through deployment descriptor](#)
 - [context-root element](#)
 - [config-name and config-file elements](#)
 - [property element](#)
 - [port-component element](#)
 - [webservice-description element](#)
- [Schema validation of SOAP messages](#)
- [JAXB Introductions](#)
- [WSDL system properties expansion](#)

35.3.1 Logging

Logging of inbound and outbound messages is a common need. Different approaches are available for achieving that:

WS Handler approach

A portable way of performing logging is writing a simple JAX-WS handler dumping the messages that are passed in it; the handler can be added to the desired client/endpoints (programmatically / using `@HandlerChain` JAX-WS annotation).

The [predefined client and endpoint configuration](#) mechanism allows user to add the logging handler to any client/endpoint or to some of them only (in which case the `@EndpointConfig` annotation / JBossWS API is required though).



Apache CXF approach

Apache CXF also comes with logging interceptors that can be easily used to log messages to the console or configured client/server log files. Those interceptors can be added to client, endpoint and buses in multiple ways:

System property

Setting the `org.apache.cxf.logging.enabled` system property to true causes the logging interceptors to be added to any `Bus` instance being created on the JVM.

i On WildFly, the system property is easily set by adding what follows to the standalone / domain server configuration just after the extensions section:

```
<system-properties>
  <property name="org.apache.cxf.logging.enabled" value="true"/>
</system-properties>
```

Manual interceptor addition and logging feature

Logging interceptors can be selectively added to endpoints using the Apache CXF annotations `@org.apache.cxf.interceptor.InInterceptors` and `@org.apache.cxf.interceptor.OutInterceptors`. The same is achieved on client side by programmatically adding new instances of the logging interceptors to the client or the bus.

Alternatively, Apache CXF also comes with a `org.apache.cxf.feature.LoggingFeature` that can be used on clients and endpoints (either annotating them with `@org.apache.cxf.feature.Features` or directly with `@org.apache.cxf.annotations.Logging`).

Please refer to the [Apache CXF documentation](#) for more details.

35.3.2 * support

JBossWS includes most of the WS-* specification functionalities through the integration with Apache CXF. In particular, the whole WS-Security Policy framework is fully supported, enabling full contract driven configuration of complex features like WS-Security.

In details information available further down in this documentation book.



35.3.3 Address rewrite

JBossWS allows users to configure the `soap:address` attribute in the wsd contract of deployed services.

Server configuration options

The configuration options are part of the [webservices subsystem section](#) of the application server domain model.

```
<subsystem xmlns="urn:jboss:domain:webservices:1.1"
xmlns:javaee="http://java.sun.com/xml/ns/javaee"
  xmlns:jaxwsconfig="urn:jboss:jbossws-jaxws-config:4.0">
  <wsdl-host>localhost</wsdl-host>
  <modify-wsdl-address>true</modify-wsdl-address>
  <!--
  <wsdl-port>8080</wsdl-port>
  <wsdl-secure-port>8443</wsdl-secure-port>
  -->
</subsystem>
```

If the content of `<soap:address>` in the wsd is a valid URL, JBossWS will not rewrite it unless `modify-wsdl-address` is true. If the content of `<soap:address>` is not a valid URL instead, JBossWS will always rewrite it using the attribute values given below. Please note that the variable `${jboss.bind.address}` can be used to set the address which the application is bound to at each startup.

The `wsdl-secure-port` and `wsdl-port` attributes are used to explicitly define the ports to be used for rewriting the SOAP address. If these attributes are not set, the ports will be identified by querying the list of installed connectors. If multiple connectors are found the port of the first connector is used.

Dynamic rewrite

When the application server is bound to multiple addresses or non-trivial real-world network architectures cause request for different external addresses to hit the same endpoint, a static rewrite of the `soap:address` may not be enough. JBossWS allows for both the `soap:address` in the wsd and the wsd address in the console to be rewritten with the host use in the client request. This way, users always get the right wsd address assuming they're connecting to an instance having the endpoint they're looking for. To trigger this behaviour, the `jbossws.undefined.host` value has to be specified for the `wsdl-host` element.

```
<wsdl-host>jbossws.undefined.host</wsdl-host>
<modify-wsdl-address>true</modify-wsdl-address>
```

Of course, when a confidential transport address is required, the addresses are always rewritten using https protocol and the port currently configured for the https/ssl connector.



35.3.4 Configuration through deployment descriptor

The `jboss-webservices.xml` deployment descriptor can be used to provide additional configuration for a given deployment. The expected location of it is:

- `META-INF/jboss-webservices.xml` for EJB webservice deployments
- `WEB-INF/jboss-webservices.xml` for POJO webservice deployments and EJB webservice endpoints bundled in `war` archives

The structure of file is the following (schemas are available [here](#)):

```
<webservices>
  <context-root/>?
  <config-name/>?
  <config-file/>?
  <property>*
    <name/>
    <value/>
  </property>
  <port-component>*
    <ejb-name/>
    <port-component-name/>
    <port-component-uri/>?
    <auth-method/>?
    <transport-guarantee/>?
    <secure-wsdl-access/>?
  </port-component>
  <webservice-description>*
    <webservice-description-name/>
    <wsdl-publish-location/>?
  </webservice-description>
</webservices>
```

context-root element

Element `<context-root>` can be used to customize context root of webservices deployment.

```
<webservices>
  <context-root>foo</context-root>
</webservices>
```



config-name and config-file elements

Elements `<config-name>` and `<config-file>` can be used to associate any endpoint provided in the deployment with a given [endpoint configuration](#). Endpoint configuration are either specified in the referenced config file or in the WildFly domain model (webservices subsystem). For further details on the endpoint configurations and their management in the domain model, please see the related [documentation](#).

```
<webservices>
  <config-name>Standard WSSecurity Endpoint</config-name>
  <config-file>META-INF/custom.xml</config-file>
</webservices>
```

property element

`<property>` elements can be used to setup simple property values to configure the ws stack behavior. Allowed property names and values are mentioned in the guide under related topics.

```
<property>
  <name>prop.name</name>
  <value>prop.value</value>
</property>
```

port-component element

Element `<port-component>` can be used to customize EJB endpoint target URI or to configure security related properties.

```
<webservices>
  <port-component>
    <ejb-name>TestService</ejb-name>
    <port-component-name>TestServicePort</port-component-name>
    <port-component-uri>/*</port-component-uri>
    <auth-method>BASIC</auth-method>
    <transport-guarantee>NONE</transport-guarantee>
    <secure-wsdl-access>true</secure-wsdl-access>
  </port-component>
</webservices>
```



webservice-description element

Element `<webservice-description>` can be used to customize (override) webservice WSDL publish location.

```
<webservises>
  <webservice-description>
    <webservice-description-name>TestService</webservice-description-name>
    <wsdl-publish-location>file:///bar/foo.wsdl</wsdl-publish-location>
  </webservice-description>
</webservises>
```



35.3.5 Schema validation of SOAP messages

Apache CXF has a feature for validating incoming and outgoing SOAP messages on both client and server side. The validation is performed against the relevant schema in the endpoint wsdl contract (server side) or the wsdl contract used for building up the service proxy (client side).

Schema validation can be turned on programmatically on client side

```
((BindingProvider)proxy).getRequestContext().put("schema-validation-enabled", true);
```

or using the `@org.apache.cxf.annotations.SchemaValidation` annotation on server side

```
import javax.jws.WebService;
import org.apache.cxf.annotations.SchemaValidation;

@WebService(...)
@SchemaValidation
public class ValidatingHelloImpl implements Hello {
    ...
}
```

Alternatively, any endpoint and client running in-container can be associated to a JBossWS [predefined configuration](#) having the `schema-validation-enabled` property set to `true` in the referenced config file.

Finally, JBossWS also allows for server-wide (default) setup of schema validation by using the *Standard-Endpoint-Config* and *Standard-Client-Config* special configurations (which apply to any client / endpoint unless a different configuration is specified for them)

```
<subsystem xmlns="urn:jboss:domain:webservices:1.2">
    ...
    <endpoint-config name="Standard-Endpoint-Config">
        <property name="schema-validation-enabled" value="true"/>
    </endpoint-config>
    ...
    <client-config name="Standard-Client-Config">
        <property name="schema-validation-enabled" value="true"/>
    </client-config>
</subsystem>
```



35.3.6 JAXB Introductions

As Kohsuke Kawaguchi wrote on [his blog](#), one common complaint from the JAXB users is the lack of support for binding 3rd party classes. The scenario is this: you are trying to annotate your classes with JAXB annotations to make it XML bindable, but some of the classes are coming from libraries and JDK, and thus you cannot put necessary JAXB annotations on it.

To solve this JAXB has been designed to provide hooks for programmatic introduction of annotations to the runtime.

This is currently leveraged by the JBoss JAXB Introductions project, using which users can define annotations in XML and make JAXB see those as if those were in the class files (perhaps coming from 3rd party libraries).

Take a look at the [JAXB Introductions page](#) on the wiki and at the examples in the sources.

35.3.7 WSDL system properties expansion

See [Published WSDL customization](#)

35.3.8 Predefined client and endpoint configurations

- [Overview](#)
- [Assigning configurations](#)
 - [Endpoint configuration assignment](#)
 - [Endpoint Configuration Deployment Descriptor](#)
 - [Application server configurations](#)
 - [Standard configurations](#)
 - [Handlers classloading](#)
 - [Examples](#)
 - [EndpointConfig annotation](#)
 - [JAXWS Feature](#)
 - [Explicit setup through API](#)
 - [Automatic configuration from default descriptors](#)
 - [Automatic configuration assignment from container setup](#)



Overview

JBossWS permits extra setup configuration data to be predefined and associated with an endpoint or a client. Configurations can include JAX-WS handlers and key/value property declarations that control JBossWS and Apache CXF internals. Predefined configurations can be used for JAX-WS client and JAX-WS endpoint setup.

Configurations can be defined in the webservice subsystem and in an application's deployment descriptor file. There can be many configuration definitions in the webservice subsystem and in an application. Each configuration must have a name that is unique within the server. Configurations defined in an application are local to the application. Endpoint implementations declare the use of a specific configuration through the use of the `org.jboss.ws.api.annotation.EndpointConfig` annotation. An endpoint configuration defined in the webservicess subsystem is available to all deployed applications on the server container and can be referenced by name in the annotation. An endpoint configuration defined in an application must be referenced by both deployment descriptor file name and configuration name by the annotation.

Handlers

Each endpoint configuration may be associated with zero or more PRE and POST handler chains. Each handler chain may include JAXWS handlers. For outbound messages the PRE handler chains are executed before any handler that is attached to the endpoint using the standard means, such as with annotation `@HandlerChain`, and POST handler chains are executed after those objects have executed. For inbound messages the POST handler chains are executed before any handler that is attached to the endpoint using the standard means and the PRE handler chains are executed after those objects have executed.

```
* Server inbound messages
Client --> ... --> POST HANDLER --> ENDPOINT HANDLERS --> PRE HANDLERS --> Endpoint

* Server outbound messages
Endpoint --> PRE HANDLER --> ENDPOINT HANDLERS --> POST HANDLERS --> ... --> Client
```

The same applies for client configurations.

Properties

Key/value properties are used for controlling both some Apache CXF internals and some JBossWS options. Specific supported values are mentioned where relevant in the rest of the documentation.

Assigning configurations

Endpoints and clients are assigned configuration through different means. Users can explicitly require a given configuration or rely on container defaults. The assignment process can be split up as follows:

- Explicit assignment through annotations (for endpoints) or API programmatic usage (for clients)
- Automatic assignment of configurations from default descriptors
- Automatic assignment of configurations from container



Endpoint configuration assignment

The explicit configuration assignment is meant for developer that know in advance their endpoint or client has to be setup according to a specified configuration. The configuration is either coming from a descriptor that is included in the application deployment, or is included in the application server webservices subsystem management model.

Endpoint Configuration Deployment Descriptor

Java EE archives that can contain JAX-WS client and endpoint implementations can also contain predefined client and endpoint configuration declarations. All endpoint/client configuration definitions for a given archive must be provided in a single deployment descriptor file, which must be an implementation of schema [jbossws-jaxws-config](#). Many endpoint/client configurations can be defined in the deployment descriptor file. Each configuration must have a name that is unique within the server on which the application is deployed. The configuration name can't be referred to by endpoint/client implementations outside the application. Here is an example of a descriptor, containing two endpoint configurations:

```
<?xml version="1.0" encoding="UTF-8"?>
<jaxws-config xmlns="urn:jboss:jbossws-jaxws-config:4.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:jvaxee="http://java.sun.com/xml/ns/jvaxee"
xsi:schemaLocation="urn:jboss:jbossws-jaxws-config:4.0 schema/jbossws-jaxws-config_4_0.xsd">
<endpoint-config>
<config-name>org.jboss.test.ws.jaxws.jbws3282.Endpoint4Impl</config-name>
<pre-handler-chains>
<jvaxee:handler-chain>
<jvaxee:handler>
<jvaxee:handler-name>Log Handler</jvaxee:handler-name>
<jvaxee:handler-class>org.jboss.test.ws.jaxws.jbws3282.LogHandler</jvaxee:handler-class>
</jvaxee:handler>
</jvaxee:handler-chain>
</pre-handler-chains>
<post-handler-chains>
<jvaxee:handler-chain>
<jvaxee:handler>
<jvaxee:handler-name>Routing Handler</jvaxee:handler-name>
<jvaxee:handler-class>org.jboss.test.ws.jaxws.jbws3282.RoutingHandler</jvaxee:handler-class>
</jvaxee:handler>
</jvaxee:handler-chain>
</post-handler-chains>
</endpoint-config>
<endpoint-config>
<config-name>EP6-config</config-name>
<post-handler-chains>
<jvaxee:handler-chain>
<jvaxee:handler>
<jvaxee:handler-name>Authorization Handler</jvaxee:handler-name>
<jvaxee:handler-class>org.jboss.test.ws.jaxws.jbws3282.AuthorizationHandler</jvaxee:handler-class>
```

Similarly, client configurations can be specified in descriptors (still implementing the schema mentioned above):



```
<?xml version="1.0" encoding="UTF-8"?>
<jaxws-config xmlns="urn:jboss:jbossws-jaxws-config:4.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:jvaee="http://java.sun.com/xml/ns/jvaee"
xsi:schemaLocation="urn:jboss:jbossws-jaxws-config:4.0 schema/jbossws-jaxws-config_4_0.xsd">
<client-config>
<config-name>Custom Client Config</config-name>
<pre-handler-chains>
<jvaee:handler-chain>
<jvaee:handler>
<jvaee:handler-name>Routing Handler</jvaee:handler-name>
<jvaee:handler-class>org.jboss.test.ws.jaxws.clientConfig.RoutingHandler</jvaee:handler-class>
</jvaee:handler>
<jvaee:handler>
<jvaee:handler-name>Custom Handler</jvaee:handler-name>
<jvaee:handler-class>org.jboss.test.ws.jaxws.clientConfig.CustomHandler</jvaee:handler-class>
</jvaee:handler>
</jvaee:handler-chain>
</pre-handler-chains>
</client-config>
<client-config>
<config-name>Another Client Config</config-name>
<post-handler-chains>
<jvaee:handler-chain>
<jvaee:handler>
<jvaee:handler-name>Routing Handler</jvaee:handler-name>
<jvaee:handler-class>org.jboss.test.ws.jaxws.clientConfig.RoutingHandler</jvaee:handler-class>
</jvaee:handler>
</jvaee:handler-chain>
</post-handler-chains>
</client-config>
</jaxws-config>
```

Application server configurations

WildFly allows declaring JBossWS client and server predefined configurations in the *webservices* subsystem section of the server model. As a consequence it is possible to declare server-wide handlers to be added to the chain of each endpoint or client assigned to a given configuration.

Please refer to the [WildFly documentation](#) for details on managing the *webservices* subsystem such as adding, removing and modifying handlers and properties.

The allowed contents in the *webservices* subsystem are defined by the [schema](#) included in the application server.

Standard configurations

Clients running in-container as well as endpoints are assigned standard configurations by default. The defaults are used unless different configurations are set as described on this page. This enables administrators to tune the default handler chains for client and endpoint configurations. The names of the default client and endpoint configurations, used in the *webservices* subsystem are `Standard-Client-Config` and `Standard-Endpoint-Config` respectively.



Handlers classloading

When setting a server-wide handler, please note the handler class needs to be available through each ws deployment classloader. As a consequence proper module dependencies might need to be specified in the deployments that are going to leverage a given predefined configuration. A shortcut is to add a dependency to the module containing the handler class in one of the modules which are already automatically set as dependencies to any deployment, for instance `org.jboss.ws.spi`.

Examples

JBoss AS 7.2 default configurations

```
<subsystem xmlns="urn:jboss:domain:webservices:2.0">
  <!-- ... -->
  <endpoint-config name="Standard-Endpoint-Config"/>
  <endpoint-config name="Recording-Endpoint-Config">
    <pre-handler-chain name="recording-handlers" protocol-bindings="##SOAP11_HTTP ##SOAP11_HTTP_MTOM
    ##SOAP12_HTTP ##SOAP12_HTTP_MTOM">
      <handler name="RecordingHandler" class="org.jboss.ws.common.invocation.RecordingServerHandler"/>
    </pre-handler-chain>
  </endpoint-config>
  <client-config name="Standard-Client-Config"/>
</subsystem>
```

A configuration file for a deployment specific ws-security endpoint setup

```
<jaxws-config xmlns="urn:jboss:jbossws-jaxws-config:4.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:javasee="http://java.sun.com/xml/ns/javasee"
xsi:schemaLocation="urn:jboss:jbossws-jaxws-config:4.0 schema/jbossws-jaxws-config_4_0.xsd">
  <endpoint-config>
    <config-name>Custom WS-Security Endpoint</config-name>
    <property>
      <property-name>ws-security.signature.properties</property-name>
      <property-value>bob.properties</property-value>
    </property>
    <property>
      <property-name>ws-security.encryption.properties</property-name>
      <property-value>bob.properties</property-value>
    </property>
    <property>
      <property-name>ws-security.signature.username</property-name>
      <property-value>bob</property-value>
    </property>
    <property>
      <property-name>ws-security.encryption.username</property-name>
      <property-value>alice</property-value>
    </property>
    <property>
      <property-name>ws-security.callback-handler</property-name>
      <property-value>org.jboss.test.ws.jaxws.samples.wsse.policy.basic.KeystorePasswordCallback</proper
```

**JBoss AS 7.2 default configurations modified to default to SOAP messages schema-validation on**

```
<subsystem xmlns="urn:jboss:domain:webservices:2.0">
  <!-- ... -->
  <endpoint-config name="Standard-Endpoint-Config">
    <property name="schema-validation-enabled" value="true"/>
  </endpoint-config>
  <!-- ... -->
  <client-config name="Standard-Client-Config">
    <property name="schema-validation-enabled" value="true"/>
  </client-config>
</subsystem>
```

EndpointConfig annotation

Once a configuration is available to a given application, the `org.jboss.ws.api.annotation.EndpointConfig` annotation is used to assign an endpoint configuration to a JAX-WS endpoint implementation. When assigning a configuration that is defined in the webservices subsystem only the configuration name is specified. When assigning a configuration that is defined in the application, the relative path to the deployment descriptor and the configuration name must be specified.

```
@EndpointConfig(configFile = "WEB-INF/my-endpoint-config.xml", configName = "Custom WS-Security
Endpoint")
public class ServiceImpl implements ServiceIface
{
  public String sayHello()
  {
    return "Secure Hello World!";
  }
}
```



JAXWS Feature

The most practical way of setting a configuration is using

`org.jboss.ws.api.configuration.ClientConfigFeature`, a JAXWS Feature extension provided by JBossWS:

```
import org.jboss.ws.api.configuration.ClientConfigFeature;

...

Service service = Service.create(wsdlURL, serviceName);
Endpoint port = service.getPort(Endpoint.class, new
ClientConfigFeature("META-INF/my-client-config.xml", "Custom Client Config"));
port.echo("Kermit");

... or ....

port = service.getPort(Endpoint.class, new ClientConfigFeature("META-INF/my-client-config.xml",
"Custom Client Config"), true); //setup properties too from the configuration
port.echo("Kermit");
... or ...

port = service.getPort(Endpoint.class, new ClientConfigFeature(null, testConfigName)); //reads
from current container configurations if available
port.echo("Kermit");
```

JBossWS parses the specified configuration file. The configuration file must be found as a resource by the classloader of the current thread. The [jbossws-jaxws-config schema](#) defines the descriptor contents and is included in the *jbossws-spi* artifact.

Explicit setup through API

Alternatively, JBossWS API comes with facility classes that can be used for assigning configurations when building a client. JAXWS handlers read from client configurations as follows:



```
import org.jboss.ws.api.configuration.ClientConfigUtil;
import org.jboss.ws.api.configuration.ClientConfigurer;

...

Service service = Service.create(wsdlURL, serviceName);
Endpoint port = service.getPort(Endpoint.class);
BindingProvider bp = (BindingProvider)port;
ClientConfigUtil.setConfigHandlers(bp, "META-INF/my-client-config.xml", "Custom Client Config
1");
port.echo("Kermit");

...

ClientConfigurer configurer = ClientConfigUtil.resolveClientConfigurer();
configurer.setConfigHandlers(bp, "META-INF/my-client-config.xml", "Custom Client Config 2");
port.echo("Kermit");

...

configurer.setConfigHandlers(bp, "META-INF/my-client-config.xml", "Custom Client Config 3");
port.echo("Kermit");

...

configurer.setConfigHandlers(bp, null, "Container Custom Client Config"); //reads from current
container configurations if available
port.echo("Kermit");
```

... similarly, properties are read from client configurations as follows:



```
import org.jboss.ws.api.configuration.ClientConfigUtil;
import org.jboss.ws.api.configuration.ClientConfigurer;

...

Service service = Service.create(wsdlURL, serviceName);
Endpoint port = service.getPort(Endpoint.class);

ClientConfigUtil.setConfigProperties(port, "META-INF/my-client-config.xml", "Custom Client
Config 1");
port.echo("Kermit");

...

ClientConfigurer configurer = ClientConfigUtil.resolveClientConfigurer();
configurer.setConfigProperties(port, "META-INF/my-client-config.xml", "Custom Client Config 2");
port.echo("Kermit");

...

configurer.setConfigProperties(port, "META-INF/my-client-config.xml", "Custom Client Config 3");
port.echo("Kermit");

...

configurer.setConfigProperties(port, null, "Container Custom Client Config"); //reads from
current container configurations if available
port.echo("Kermit");
```

The default `ClientConfigurer` implementation parses the specified configuration file, if any, after having resolved it as a resources using the current thread context classloader. The [jbossws-jaxws-config schema](#) defines the descriptor contents and is included in the `jbossws-spi` artifact.



Automatic configuration from default descriptors

In some cases, the application developer might not be aware of the configuration that will need to be used for its client and endpoint implementation, perhaps because that's a concern of the application deployer. In other cases, explicit usage (compile time dependency) of JBossWS API might not be accepted. To cope with such scenarios, JBossWS allows including default client (`jaxws-client-config.xml`) and endpoint (`jaxws-endpoint-config.xml`) descriptor within the application (in its root), which are parsed for getting configurations any time a configuration file name is not specified.

If the configuration name is also not specified, JBossWS automatically looks for a configuration named the same as

- the endpoint implementation class (full qualified name), in case of JAX-WS endpoints;
- the service endpoint interface (full qualified name), in case of JAX-WS clients.

No automatic configuration name is selected for `Dispatch` clients.

So, for instance, an endpoint implementation class `org.foo.bar.EndpointImpl` for which no pre-defined configuration is explicitly set will cause JBossWS to look for a `org.foo.bar.EndpointImpl` named configuration within a `jaxws-endpoint-config.xml` descriptor in the root of the application deployment. Similarly, on client side, a client proxy implementing `org.foo.bar.Endpoint` interface (SEI) will have the setup read from a `org.foo.bar.Endpoint` named configuration in `jaxws-client-config.xml` descriptor.

Automatic configuration assignment from container setup

JBossWS fall-backs to getting predefined configurations from the container setup whenever no explicit configuration has been provided and the default descriptors are either not available or do not contain relevant configurations. This gives additional control on the JAX-WS client and endpoint setup to administrators, as the container setup can be managed independently from the deployed applications. JBossWS hence accesses the webservices subsystem the same as explained above for explicitly named configuration; the default configuration names used for look are

- the endpoint implementation class (full qualified name), in case of JAX-WS endpoints;
- the service endpoint interface (full qualified name), in case of JAX-WS clients.
`Dispatch` clients are not automatically configured. If no configuration is found using names computed as above, the `Standard-Client-Config` and `Standard-Endpoint-Config` configurations are used for clients and endpoints respectively

35.3.9 Authentication

- [Authentication](#)
 - [Specify the security domain](#)
 - [Use BindingProvider to set principal/credential](#)
 - [Using HTTP Basic Auth for security](#)
- [JASPI Authentication](#)



Authentication

Here the simplest way to authenticate a web service user with JBossWS is explained.

First we secure the access to the SLSB as we would do for normal (non web service) invocations: this can be easily done through the `@RolesAllowed`, `@PermitAll`, `@DenyAll` annotation. The allowed user roles can be set with these annotations both on the bean class and on any of its business methods.

```
@Stateless
@RolesAllowed("friend")
public class EndpointEJB implements EndpointInterface
{
    ...
}
```

Similarly POJO endpoints are secured the same way as we do for normal web applications in web.xml:

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>All resources</web-resource-name>
    <url-pattern>/*</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <role-name>friend</role-name>
  </auth-constraint>
</security-constraint>

<security-role>
  <role-name>friend</role-name>
</security-role>
```



Specify the security domain

Next, specify the security domain for this deployment. This is performed using the `@SecurityDomain` annotation for EJB3 endpoints

```
@Stateless
@SecurityDomain("JBossWS")
@RolesAllowed("friend")
public class EndpointEJB implements EndpointInterface
{
    ...
}
```

or modifying the `jboss-web.xml` for POJO endpoints

```
<jboss-web>
<security-domain>JBossWS</security-domain>
</jboss-web>
```

The security domain as well as its the authentication and authorization mechanisms are defined differently depending on the application server version in use.

Use BindingProvider to set principal/credential

A web service client may use the `javax.xml.ws.BindingProvider` interface to set the username/password combination

```
URL wsdlURL = new
File("resources/jaxws/samples/context/WEB-INF/wsdl/TestEndpoint.wsdl").toURL();
QName qname = new QName("http://org.jboss.ws/jaxws/context", "TestEndpointService");
Service service = Service.create(wsdlURL, qname);
port = (TestEndpoint)service.getPort(TestEndpoint.class);

BindingProvider bp = (BindingProvider)port;
bp.getRequestContext().put(BindingProvider.USERNAME_PROPERTY, "kermit");
bp.getRequestContext().put(BindingProvider.PASSWORD_PROPERTY, "thefrog");
```




Using HTTP Basic Auth for security

To enable HTTP Basic authentication you use the `@WebContext` annotation on the bean class

```
@Stateless
@SecurityDomain("JBossWS")
@RolesAllowed("friend")
@WebContext(contextRoot="/my-cxt", urlPattern="/*", authMethod="BASIC",
transportGuarantee="NONE", secureWSDLAccess=false)
public class EndpointEJB implements EndpointInterface
{
    ...
}
```

For POJO endpoints, we modify the `web.xml` adding the `auth-method` element:

```
<login-config>
  <auth-method>BASIC</auth-method>
  <realm-name>Test Realm</realm-name>
</login-config>
```

JASPI Authentication

A Java Authentication SPI (JASPI) provider can be configured in WildFly security subsystem to authenticate SOAP messages:

```
<security-domain name="jaspi">
  <authentication-jaspi>
    <login-module-stack name="jaas-lm-stack">
      <login-module code="UsersRoles" flag="required">
        <module-option name="usersProperties" value="jbossws-users.properties"/>
        <module-option name="rolesProperties" value="jbossws-roles.properties"/>
      </login-module>
    </login-module-stack>
    <auth-module code="org.jboss.wsf.stack.cxf.jaspi.module.UsernameTokenServerAuthModule"
login-module-stack-ref="jaas-lm-stack"/>
  </authentication-jaspi>
</security-domain>
```



For further information on configuring security domains in WildFly, please refer to [here](#).

Here `org.jboss.wsf.stack.cxf.jaspi.module.UsernameTokenServerAuthModule` is the class implementing `javax.security.auth.message.module.ServerAuthModule`, which delegates to the proper login module to perform authentication using the credentials from WS-Security UsernameToken in the incoming SOAP message. Alternative implementations of `ServerAuthModule` can be implemented and configured.



To enable JASPI authentication, the endpoint deployment needs to specify the security domain to use; that can be done in two different ways:

- Setting the `jaspi.security.domain` property in the `jboss-webservices.xml` descriptor

```
<?xml version="1.1" encoding="UTF-8"?>
<webservices
xmlns="http://www.jboss.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
version="1.2"
xsi:schemaLocation="http://www.jboss.com/xml/ns/javaee">

<property>
<name>jaspi.security.domain</name>
<value>jaspi</value>
</property>


</webservices>
```

- Referencing (through `@EndpointConfig` annotation) an endpoint config that sets the `jaspi.security.domain` property

```
@EndpointConfig(configFile = "WEB-INF/jaxws-endpoint-config.xml", configName =
"jaspiSecurityDomain")
public class ServiceEndpointImpl implements ServiceIface {
```

The `jaspi.security.domain` property is specified as follows in the referenced descriptor:

```
<?xml version="1.0" encoding="UTF-8"?>
<jaxws-config xmlns="urn:jboss:jbossws-jaxws-config:4.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:javaee="http://java.sun.com/xml/ns/javaee"
xsi:schemaLocation="urn:jboss:jbossws-jaxws-config:4.0 schema/jbossws-jaxws-config_4_0.xsd">
<endpoint-config>
<config-name>jaspiSecurityDomain</config-name>
<property>
<property-name>jaspi.security.domain</property-name>
<property-value>jaspi</property-value>
</property>
</endpoint-config>
</jaxws-config>
```

 If the JASPI security domain is specified in both `jboss-webservices.xml` and config file referenced by `@EndpointConfig` annotation, the JASPI security domain specified in `jboss-webservices.xml` will take precedence.



35.3.10 Apache CXF integration

- [JBossWS integration layer with Apache CXF](#)
- [Building WS applications the JBoss way](#)
 - [Portable applications](#)
 - [Direct Apache CXF API usage](#)
- [Bus usage](#)
 - [Creating a Bus instance](#)
 - [Using existing Bus instances](#)
 - [Bus selection strategies for JAXWS clients](#)
 - [Thread bus strategy \(THREAD_BUS\)](#)
 - [New bus strategy \(NEW_BUS\)](#)
 - [Thread context classloader bus strategy \(TCCL_BUS\)](#)
 - [Strategy configuration](#)
- [Server Side Integration Customization](#)
 - [Deployment descriptor properties](#)
 - [WorkQueue configuration](#)
 - [Policy alternative selector](#)
 - [MBean management](#)
 - [Schema validation](#)
 - [Interceptors](#)
 - [Features](#)
 - [WS-Discovery enablement](#)
- [Apache CXF interceptors](#)
- [Apache CXF features](#)
- [Properties driven bean creation](#)
- [HTTPConduit configuration](#)



JBossWS integration layer with Apache CXF

All JAX-WS functionalities provided by JBossWS on top of WildFly are currently served through a proper integration of the JBoss Web Services stack with most of the [Apache CXF](#) project modules.

Apache CXF is an open source services framework. It allows building and developing services using frontend programming APIs (including JAX-WS), with services speaking a variety of protocols such as SOAP and XML/HTTP over a variety of transports such as HTTP and JMS.

The integration layer (*JBossWS-CXF* in short hereafter) is mainly meant for:

- allowing using standard webservices APIs (including JAX-WS) on WildFly; this is performed internally leveraging Apache CXF without requiring the user to deal with it;
- allowing using Apache CXF advanced features (including WS-*) on top of WildFly without requiring the user to deal with / setup / care about the required integration steps for running in such a container.

In order for achieving the goals above, the JBossWS-CXF integration supports the JBoss ws endpoint deployment mechanism and comes with many internal customizations on top of Apache CXF.

In the next sections a list of technical suggestions and notes on the integration is provided; please also refer to the [Apache CXF official documentation](#) for in-depth details on the CXF architecture.



Building WS applications the JBoss way

The Apache CXF client and endpoint configuration as explained in the [Apache CXF official user guide](#) is heavily based on Spring. Apache CXF basically parses Spring `cxf.xml` descriptors; those may contain any basic bean plus specific ws client and endpoint beans which CXF has custom parsers for. Apache CXF can be used to deploy webservice endpoints on any servlet container by including its libraries in the deployment; in such a scenario Spring basically serves as a convenient configuration option, given direct Apache CXF API usage won't be very handy. Similar reasoning applies on client side, where a Spring based descriptor offers a shortcut for setting up Apache CXF internals.

This said, nowadays almost any Apache CXF functionality can be configured and used through direct API usage, without Spring. As a consequence of that and given the considerations in the sections below, the JBossWS integration with Apache CXF does not rely on Spring descriptors.

Portable applications

WildFly is much more than a servlet container; it actually provides users with a fully compliant target platform for Java EE applications.

Generally speaking, *users are encouraged to write portable applications* by relying only on *JAX-WS specification* whenever possible. That would by the way ensure easy migrations to and from other compliant platforms. Being a Java EE container, WildFly already comes with a JAX-WS compliant implementation, which is basically Apache CXF plus the JBossWS-CXF integration layer. So users just need to write their JAX-WS application; *no need for embedding any Apache CXF or any ws related dependency library in user deployments*. Please refer to the [JAX-WS User Guide](#) section of the documentation for getting started.

WS-* usage (including WS-Security, WS-Addressing, WS-ReliableMessaging, ...) should also be configured in the most portable way; that is by *relying on proper WS-Policy assertions* on the endpoint WSDL contracts, so that client and endpoint configuration is basically a matter of setting few ws context properties. The WS-* related sections of this documentation cover all the details on configuring applications making use of WS-* through policies.

As a consequence of the reasoning above, the JBossWS-CXF integration is currently built directly on the Apache CXF API and aims at allowing users to configure webservice clients and endpoints *without Spring descriptors*.

Direct Apache CXF API usage

Whenever users can't really meet their application requirements with JAX-WS plus WS-Policy, it is of course still possible to rely on direct Apache CXF API usage (given that's included in the AS), loosing the Java EE portability of the application. That could be the case of a user needing specific Apache CXF functionalities, or having to consume WS-* enabled endpoints advertised through legacy wsdL contracts without WS-Policy assertions.

On server side, direct Apache CXF API usage might not be always possible or end up being not very easy. For this reason, the JBossWS integration comes with a convenient alternative through customization options in the `jboss-webservices.xml` descriptor described below on this page. Properties can be declared in `jboss-webservices.xml` to control Apache CXF internals like *interceptors*, *features*, etc.



Bus usage

Creating a Bus instance

Most of the Apache CXF features are configurable using the `org.apache.cxf.Bus` class. While for basic JAX-WS usage the user might never need to explicitly deal with `Bus`, using Apache CXF specific features generally requires getting a handle to a `org.apache.cxf.Bus` instance. This can happen on client side as well as in a ws endpoint or handler business code.

New `Bus` instances are produced by the currently configured `org.apache.cxf.BusFactory` implementation the following way:

```
Bus bus = BusFactory.newInstance().createBus();
```

The algorithm for selecting the actual implementation of `BusFactory` to be used leverages the Service API, basically looking for optional configurations in `META-INF/services/...` location using the current thread context classloader. JBossWS-CXF integration comes with its own implementation of `BusFactory`, `org.jboss.wsf.stack.cxf.client.configuration.JBossWSBusFactory`, that allows for seamless setup of JBossWS customizations on top of Apache CXF. So, assuming the JBossWS-CXF libraries are available in the current thread context classloader, the `JBossWSBusFactory` is *automatically* retrieved by the `BusFactory.newInstance()` call above.

JBossWS users willing to explicitly use functionalities of `org.apache.cxf.bus.CXFBusFactory`, get the same API with JBossWS additions through `JBossWSBusFactory`:

```
Map<Class, Object> myExtensions = new HashMap<Class, Object>();  
myExtensions.put(...);  
Bus bus = new JBossWSBusFactory().createBus(myExtensions);
```



Using existing Bus instances

Apache CXF keeps reference to a global default `Bus` instance as well as to a thread default bus for each thread. That is performed through static members in `org.apache.cxf.BusFactory`, which also comes with the following methods in the public API:

```
public static synchronized Bus getDefaultBus()
public static synchronized Bus getDefaultBus(boolean createIfNeeded)
public static synchronized void setDefaultBus(Bus bus)
public static Bus getThreadDefaultBus()
public static Bus getThreadDefaultBus(boolean createIfNeeded)
public static void setThreadDefaultBus(Bus bus)
```

Please note that the default behaviour of `getDefaultBus()` / `getDefaultBus(true)` / `getThreadDefaultBus()` / `getThreadDefaultBus(true)` is to create a new `Bus` instance if that's not set yet. Moreover `getThreadDefaultBus()` and `getThreadDefaultBus(true)` first fallback to retrieving the configured global default bus before actually trying creating a new instance (and the created new instance is set as global default bus if that was not set there yet).

The drawback of this mechanism (which is basically fine in JSE environment) is that when running in WildFly container you need to be careful in order not to (mis)use a bus over multiple applications (assuming the Apache CXF classes are loaded by the same classloader, which is currently the case with WildFly).

Here is a list of general suggestions to avoid problems when running in-container:

- forget about the global default bus; you don't need that, so don't do `getDefaultBus()` / `getDefaultBus(true)` / `setDefaultBus()` in your code;
- avoid `getThreadDefaultBus()` / `getThreadDefaultBus(true)` unless you already know for sure the default bus is already set;
- keep in mind thread pooling whenever you customize a thread default bus instance (for instance adding bus scope interceptors, ...), as that thread and bus might be later reused; so either shutdown the bus when you're done or explicitly remove it from the `BusFactory` thread association.

Finally, remember that each time you explicitly create a new `Bus` instance (`factory.createBus()`) that is set as thread default bus and global default bus if those are not set yet. The `JAXWS Provider` implementation also creates `Bus` instances internally, in particular the `JBossWS` version of `JAXWS Provider` makes sure the default bus is never internally used and instead a new `Bus` is created if required (more details on this in the next paragraph).

Bus selection strategies for JAXWS clients

`JAXWS` clients require an Apache CXF `Bus` to be available; the client is registered within the `Bus` and the `Bus` affects the client behavior (e.g. through the configured CXF interceptors). The way a bus is internally selected for serving a given `JAXWS` client is very important, especially for in-container clients; for this reason, `JBossWS` users can choose the preferred `Bus` selection strategy. The strategy is enforced in the `javax.xml.ws.spi.Provider` implementation from the `JBossWS` integration, being that called whenever a `JAXWS Service` (client) is requested.



Thread bus strategy (THREAD_BUS)

Each time the vanilla JAXWS api is used to create a Bus, the JBossWS-CXF integration will automatically make sure a Bus is currently associated to the current thread in the BusFactory. If that's not the case, a new Bus is created and linked to the current thread (to prevent the user from relying on the default Bus). The Apache CXF engine will then create the client using the current thread Bus.

This is the default strategy, and the most straightforward one in Java SE environments; it lets users automatically reuse a previously created Bus instance and allows using customized Bus that can possibly be created and associated to the thread before building up a JAXWS client.

The drawback of the strategy is that the link between the Bus instance and the thread needs to be eventually cleaned up (when not needed anymore). This is really evident in a Java EE environment (hence when running in-container), as threads from pools (e.g. serving web requests) are re-used.

When relying on this strategy, the safest approach to be sure of cleaning up the link is to surround the JAXWS client with a `try/finally` block as below:

```
try {
    Service service = Service.create(wsdlURL, serviceQName);
    MyEndpoint port = service.getPort(MyEndpoint.class);
    //...
} finally {
    BusFactory.setThreadDefaultBus(null);
    // OR (if you don't need the bus and the client anymore)
    Bus bus = BusFactory.getThreadDefaultBus(false);
    bus.shutdown(true);
}
```

New bus strategy (NEW_BUS)

Another strategy is to have the JAXWS Provider from the JBossWS integration create a new Bus each time a JAXWS client is built. The main benefit of this approach is that a fresh bus won't rely on any formerly cached information (e.g. cached WSDL / schemas) which might have changed after the previous client creation. The main drawback is of course worse performance as the Bus creation takes time.

If there's a bus already associated to the current thread before the JAXWS client creation, that is automatically restored when returning control to the user; in other words, the newly created bus will be used only for the created JAXWS client but won't stay associated to the current thread at the end of the process. Similarly, if the thread was not associated to any bus before the client creation, no bus will be associated to the thread at the end of the client creation.



Thread context classloader bus strategy (TCCL_BUS)

The last strategy is to have the bus created for serving the client be associated to the current thread context classloader (TCCL). That basically means the same Bus instance is shared by JAXWS clients running when the same TCCL is set. This is particularly interesting as each web application deployment usually has its own context classloader, so this strategy is possibly a way to keep the number of created Bus instances bound to the application number in WildFly container.

If there's a bus already associated to the current thread before the JAXWS client creation, that is automatically restored when returning control to the user; in other words, the bus corresponding to the current thread context classloader will be used only for the created JAXWS client but won't stay associated to the current thread at the end of the process. If the thread was not associated to any bus before the client creation, a new bus will be created (and later user for any other client built with this strategy and the same TCCL in place); no bus will be associated to the thread at the end of the client creation.

Strategy configuration

Users can request a given Bus selection strategy to be used for the client being built by specifying one of the following JBossWS features (which extend `javax.xml.ws.WebServiceFeature`):

Feature	Strategy
<code>org.jboss.wsf.stack.cxf.client.UseThreadBusFeature</code>	THREAD_BUS
<code>org.jboss.wsf.stack.cxf.client.UseNewBusFeature</code>	NEW_BUS
<code>org.jboss.wsf.stack.cxf.client.UseTCCLBusFeature</code>	TCCL_BUS

The feature is specified as follows:

```
Service service = Service.create(wsdlURL, serviceQName, new UseThreadBusFeature());
```

If no feature is explicitly specified, the system default strategy is used, which can be modified through the `org.jboss.ws.cxf.jaxws-client.bus.strategy` system property when starting the JVM. The valid values for the property are `THREAD_BUS`, `NEW_BUS` and `TCCL_BUS`. The default is `THREAD_BUS`.

Server Side Integration Customization

The JBossWS-CXF server side integration takes care of internally creating proper Apache CXF structures (including a `Bus` instance, of course) for the provided `ws` deployment. Should the deployment include multiple endpoints, those would all live within the same Apache CXF Bus, which would of course be completely separated by the other deployments' bus instances.

While JBossWS sets sensible defaults for most of the Apache CXF configuration options on server side, users might want to fine tune the `Bus` instance that's created for their deployment; a `jboss-webservices.xml` descriptor can be used for deployment level customizations.



Deployment descriptor properties

The `jboss-webservices.xml` descriptor can be used to [provide property values](#).

```
<webservices xmlns="http://www.jboss.com/xml/ns/javaee" version="1.2">
  ...
  <property>
    <name>...</name>
    <value>...</value>
  </property>
  ...
</webservices>
```

JBossWS-CXF integration comes with a set of allowed property names to control Apache CXF internals.

WorkQueue configuration

Apache CXF uses `WorkQueue` instances for dealing with some operations (e.g. `@Oneway` requests processing). A [WorkQueueManager](#) is installed in the Bus as an extension and allows for adding / removing queues as well as controlling the existing ones.

On server side, queues can be provided by using the `cxf.queue.<queue-name>.*` properties in `jboss-webservices.xml` (e.g. `cxf.queue.default.maxQueueSize` for controlling the max queue size of the default workqueue). At deployment time, the JBossWS integration can add new instances of [AutomaticWorkQueueImpl](#) to the currently configured `WorkQueueManager`; the properties below are used to fill in parameter into the [AutomaticWorkQueueImpl](#) constructor:

Property	Default value
<code>cxf.queue.<queue-name>.maxQueueSize</code>	256
<code>cxf.queue.<queue-name>.initialThreads</code>	0
<code>cxf.queue.<queue-name>.highWaterMark</code>	25
<code>cxf.queue.<queue-name>.lowWaterMark</code>	5
<code>cxf.queue.<queue-name>.dequeueTimeout</code>	120000

Policy alternative selector

The Apache CXF policy engine supports different strategies to deal with policy alternatives. JBossWS-CXF integration currently defaults to the [MaximalAlternativeSelector](#), but still allows for setting different selector implementation using the `cxf.policy.alternativeSelector` property in `jboss-webservices.xml`.



MBean management

Apache CXF allows managing its MBean objects that are installed into the WildFly MBean server. The feature is enabled on a deployment basis through the `cxf.management.enabled` property in `jboss-webservices.xml`. The `cxf.management.installResponseTimeInterceptors` property can also be used to control installation of CXF response time interceptors, which are added by default when enabling MBean management, but might not be desired in some cases. Here is an example:

```
<webservices xmlns="http://www.jboss.com/xml/ns/javaee" version="1.2">
  <property>
    <name>cxf.management.enabled</name>
    <value>true</value>
  </property>
  <property>
    <name>cxf.management.installResponseTimeInterceptors</name>
    <value>>false</value>
  </property>
</webservices>
```

Schema validation

Schema validation of exchanged messages can also be enabled in `jboss-webservices.xml`. Further details available [here](#).

Interceptors

The `jboss-webservices.xml` descriptor also allows specifying the `cxf.interceptors.in` and `cxf.interceptors.out` properties; those allows declaring interceptors to be attached to the Bus instance that's created for serving the deployment.

```
<?xml version="1.1" encoding="UTF-8"?>
<webservices
xmlns="http://www.jboss.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
version="1.2"
xsi:schemaLocation="http://www.jboss.com/xml/ns/javaee">

  <property>
    <name>cxf.interceptors.in</name>
    <value>org.jboss.test.ws.jaxws.cxf.interceptors.BusInterceptor</value>
  </property>
  <property>
    <name>cxf.interceptors.out</name>
    <value>org.jboss.test.ws.jaxws.cxf.interceptors.BusCounterInterceptor</value>
  </property>
</webservices>
```



Features

The `jboss-webservices.xml` descriptor also allows specifying the `cxf.features` property; that allows declaring features to be attached to any endpoint belonging to the Bus instance that's created for serving the deployment.

```
<?xml version="1.1" encoding="UTF-8"?>
<webservices
xmlns="http://www.jboss.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
version="1.2"
xsi:schemaLocation="http://www.jboss.com/xml/ns/javaee">

  <property>
    <name>cxf.features</name>
    <value>org.apache.cxf.feature.FastInfosetFeature</value>
  </property>
</webservices>
```

Discovery enablement

WS-Discovery support can be turned on in `jboss-webservices` for the current deployment. Further details available [here](#).



Apache CXF interceptors

Apache CXF supports declaring interceptors using one of the following approaches:

- Annotation usage on endpoint classes (`@org.apache.cxf.interceptor.InInterceptor`, `@org.apache.cxf.interceptor.OutInterceptor`)
- Direct API usage on client side (through the `org.apache.cxf.interceptor.InterceptorProvider` interface)
- Spring descriptor usage (*cx.xml*)

As the Spring descriptor usage is not supported, the JBossWS integration adds an additional descriptor based approach to avoid requiring modifications to the actual client/endpoint code. Users can declare interceptors within [predefined client and endpoint configurations](#) by specifying a list of interceptor class names for the `cx.xml.interceptors.in` and `cx.xml.interceptors.out` properties.

```
<?xml version="1.0" encoding="UTF-8"?>
<jaxws-config xmlns="urn:jboss:jbossws-jaxws-config:4.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:javaee="http://java.sun.com/xml/ns/javaee"
xsi:schemaLocation="urn:jboss:jbossws-jaxws-config:4.0 schema/jbossws-jaxws-config_4_0.xsd">
<endpoint-config>
<config-name>org.jboss.test.ws.jaxws.cxf.interceptors.EndpointImpl</config-name>
<property>
<property-name>cx.xml.interceptors.in</property-name>
<property-value>org.jboss.test.ws.jaxws.cxf.interceptors.EndpointInterceptor,org.jboss.test.ws.jax
```

A new instance of each specified interceptor class will be added to the client or endpoint the configuration is assigned to. The interceptor classes must have a no-argument constructor.



Apache CXF features

Apache CXF supports declaring features using one of the following approaches:

- Annotation usage on endpoint classes (`@org.apache.cxf.feature.Features`)
- Direct API usage on client side (through extensions of the `org.apache.cxf.feature.AbstractFeature` class)
- Spring descriptor usage (`cxf.xml`)

As the Spring descriptor usage is not supported, the JBossWS integration adds an additional descriptor based approach to avoid requiring modifications to the actual client/endpoint code. Users can declare features within [predefined client and endpoint configurations](#) by specifying a list of feature class names for the `cxf.features` property.

```
<?xml version="1.0" encoding="UTF-8"?>
<jaxws-config xmlns="urn:jboss:jbossws-jaxws-config:4.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:javaee="http://java.sun.com/xml/ns/javaee"
xsi:schemaLocation="urn:jboss:jbossws-jaxws-config:4.0 schema/jbossws-jaxws-config_4_0.xsd">
<endpoint-config>
<config-name>Custom FI Config</config-name>
<property>
<property-name>cxf.features</property-name>
<property-value>org.apache.cxf.feature.FastInfosetFeature</property-value>
</property>
</endpoint-config>
</jaxws-config>
```

A new instance of each specified feature class will be added to the client or endpoint the configuration is assigned to. The feature classes must have a no-argument constructor.



Properties driven bean creation

Sections above explain how to declare CXF interceptors and features through properties either in a client/endpoint predefined configuration or in a `jboss-webservices.xml` descriptor. By getting the feature/interceptor class name only specified, the container simply tries to create a bean instance using the class default constructor. This sets a limitation on the feature/interceptor configuration, unless custom extensions of vanilla CXF classes are provided, with the default constructor setting properties before eventually using the super constructor.

To cope with this issue, JBossWS integration comes with a mechanism for configuring simple bean hierarchies when building them up from properties. Properties can have bean reference values, that is strings starting with `##`. Property reference keys are used to specify the bean class name and the value for for each attribute. So for instance the following properties:

Key	Value
<code>cxf.features</code>	<code>##foo, ##bar</code>
<code>##foo</code>	<code>org.jboss.Foo</code>
<code>##foo.par</code>	<code>34</code>
<code>##bar</code>	<code>org.jboss.Bar</code>
<code>##bar.color</code>	<code>blue</code>

would result into the stack installing two feature instances, the same that would have been created by

```
import org.Bar;
import org.Foo;

...

Foo foo = new Foo();
foo.setPar(34);
Bar bar = new Bar();
bar.setColor("blue");
```

The mechanism assumes that the classes are valid beans with proper getter and setter methods; value objects are cast to the correct primitive type by inspecting the class definition. Nested beans can of course be configured.



HTTPConduit configuration

HTTP transport setup in Apache CXF is achieved through

`org.apache.cxf.transport.http.HTTPConduit` [configurations](#). When running on top of the JBossWS integration, conduits can be programmatically modified using the Apache CXF API as follows:

```
import org.apache.cxf.frontend.ClientProxy;
import org.apache.cxf.transport.http.HTTPConduit;
import org.apache.cxf.transports.http.configuration.HTTPClientPolicy;

//set chunking threshold before using a JAX-WS port client
...
HTTPConduit conduit = (HTTPConduit)ClientProxy.getClient(port).getConduit();
HTTPClientPolicy client = conduit.getClient();

client.setChunkingThreshold(8192);
...
```

Users can also control the default values for the most common HTTPConduit parameters by setting specific system properties; the provided values will override Apache CXF default values.

Property	Description
<code>cxf.client.allowChunking</code>	A boolean to tell Apache CXF whether to allow send messages using chunking.
<code>cxf.client.chunkingThreshold</code>	An integer value to tell Apache CXF the threshold at which switching from non-chunking to chunking mode.
<code>cxf.client.connectionTimeout</code>	A long value to tell Apache CXF how many milliseconds to set the connection timeout to
<code>cxf.client.receiveTimeout</code>	A long value to tell Apache CXF how many milliseconds to set the receive timeout to
<code>cxf.client.connection</code>	A string to tell Apache CXF to use <code>Keep-Alive</code> or <code>close</code> connection type
<code>cxf.tls-client.disableCNCheck</code>	A boolean to tell Apache CXF whether disabling CN host name check or not

The vanilla Apache CXF defaults apply when the system properties above are not set.

35.3.11 Addressing

JBoss Web Services inherits full WS-Addressing capabilities from the underlying Apache CXF implementation. Apache CXF provides support for 2004-08 and [1.0](#) versions of WS-Addressing.



Enabling WS-Addressing

WS-Addressing can be turned on in multiple standard ways:

- consuming a WSDL contract that specifies a WS-Addressing assertion / policy
- using the `@javax.xml.ws.soap.Addressing` annotation
- using the `javax.xml.ws.soap.AddressingFeature` feature

i The supported addressing policy elements are:

```
[http://www.w3.org/2005/02/addressing/wsdl]UsingAddressing
[http://schemas.xmlsoap.org/ws/2004/08/addressing/policy]UsingAddressing
[http://www.w3.org/2006/05/addressing/wsdl]UsingAddressing
[http://www.w3.org/2007/05/addressing/metadata]Addressing
```

Alternatively, Apache CXF proprietary ways are also available:

- specifying the `[http://cxf.apache.org/ws/addressing]addressing` feature for a given client/endpoint
- using the `org.apache.cxf.ws.addressing.WSAddressingFeature` feature through the API
- manually configuring the Apache CXF addressing interceptors (`org.apache.cxf.ws.addressing.MAPAggregator` and `org.apache.cxf.ws.addressing.soap.MAPCodec`)
- setting the `org.apache.cxf.ws.addressing.using` property in the message context

Please refer to the the Apache CXF documentation for further information on the proprietary [WS-Addressing setup](#) and [configuration details](#).

Addressing Policy

The WS-Addressing support is also perfectly integrated with the Apache CXF WS-Policy engine.

This basically means that the WSDL contract generation for code-first endpoint deployment is policy-aware: users can annotate endpoints with the `@javax.xml.ws.soap.Addressing` annotation and expect the published WSDL contract to contain proper WS-Addressing policy (assuming no `wSDLLocation` is specified in the endpoint's `@WebService` annotation).

Similarly, on client side users do not need to manually specify the `javax.xml.ws.soap.AddressingFeature` feature, as the policy engine is able to properly process the WS-Addressing policy in the consumed WSDL and turn on addressing as requested.

Example

Here is an example showing how to simply enable WS-Addressing through WS-Policy.



Endpoint

A simple JAX-WS endpoint is prepared using a java-first approach; WS-Addressing is enforced through `@Addressing` annotation and no `wsdlLocation` is provided in `@WebService`:

```
package org.jboss.test.ws.jaxws.samples.wsa;

import javax.jws.WebService;
import javax.xml.ws.soap.Addressing;
import org.jboss.logging.Logger;

@WebService
(
    portName = "AddressingServicePort",
    serviceName = "AddressingService",
    targetNamespace = "http://www.jboss.org/jbossws/ws-extensions/wsaddressing",
    endpointInterface = "org.jboss.test.ws.jaxws.samples.wsa.ServiceIface"
)
@Addressing(enabled=true, required=true)
public class ServiceImpl implements ServiceIface
{
    private Logger log = Logger.getLogger(this.getClass());

    public String sayHello(String name)
    {
        return "Hello " + name + "!";
    }
}
```

The WSDL contract that's generated at deploy time and published looks like this:



```
<wsdl:definitions ....>
...
<wsdl:binding name="AddressingServiceSoapBinding" type="tns:ServiceIface">
  <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
  <wsaw:UsingAddressing wsdl:required="true"/>
  <wsp:PolicyReference URI="#AddressingServiceSoapBinding_WSAM_Addressig_Policy"/>

  <wsdl:operation name="sayHello">
    <soap:operation soapAction="" style="document"/>
    <wsdl:input name="sayHello">
      <soap:body use="literal"/>
    </wsdl:input>
    <wsdl:output name="sayHelloResponse">
      <soap:body use="literal"/>
    </wsdl:output>
  </wsdl:operation>

</wsdl:binding>
<wsdl:service name="AddressingService">
  <wsdl:port binding="tns:AddressingServiceSoapBinding" name="AddressingServicePort">
    <soap:address location="http://localhost:8080/jaxws-samples-wsa"/>
  </wsdl:port>
</wsdl:service>
  <wsp:Policy wsu:Id="AddressingServiceSoapBinding_WSAM_Addressig_Policy"

xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd">
  <wsam:Addressing xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata">
    <wsp:Policy/>
  </wsam:Addressing>
</wsp:Policy>
</wsdl:definitions>
```

Client

Since the WS-Policy engine is on by default, the client side code is basically a pure JAX-WS client app:

```
QName serviceName = new QName("http://www.jboss.org/jbossws/ws-extensions/wsaddressing",
    "AddressingService");
URL wsdlURL = new URL("http://localhost:8080/jaxws-samples-wsa?wsdl");
Service service = Service.create(wsdlURL, serviceName);
ServiceIface proxy = (ServiceIface)service.getPort(ServiceIface.class);
proxy.sayHello("World");
```



35.3.12 Security

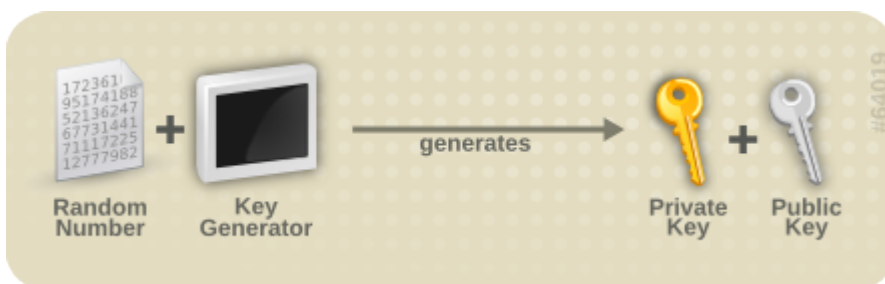
- [WS-Security overview](#)
- [JBoss WS-Security support](#)
 - [Apache CXF WS-Security implementation](#)
 - [WS-Security Policy support](#)
 - [JBossWS configuration additions](#)
 - [Apache CXF annotations](#)
- [Examples](#)
 - [Signature and encryption](#)
 - [Endpoint](#)
 - [Client](#)
 - [Endpoint serving multiple clients](#)
 - [Authentication and authorization](#)
 - [Endpoint](#)
 - [Client](#)
 - [Secure transport](#)
 - [Secure conversation](#)

Security overview

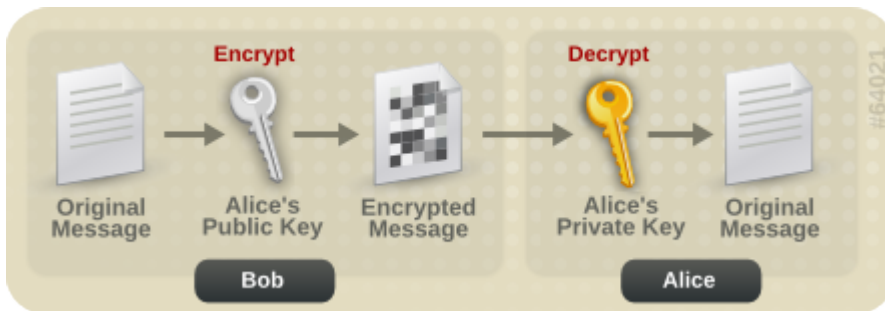
WS-Security provides the means to secure your services beyond transport level protocols such as *HTTPS*. Through a number of standards such as [XML-Encryption](#), and headers defined in the [WS-Security](#) standard, it allows you to:

- Pass authentication tokens between services.
- Encrypt messages or parts of messages.
- Sign messages.
- Timestamp messages.

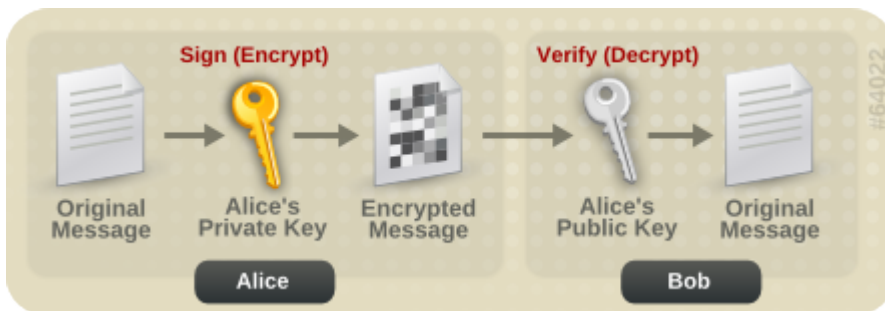
WS-Security makes heavy use of public and private key cryptography. It is helpful to understand these basics to really understand how to configure WS-Security. With public key cryptography, a user has a pair of public and private keys. These are generated using a large prime number and a key function.



The keys are related mathematically, but cannot be derived from one another. With these keys we can encrypt messages. For example, if Bob wants to send a message to Alice, he can encrypt a message using her public key. Alice can then decrypt this message using her private key. Only Alice can decrypt this message as she is the only one with the private key.



Messages can also be signed. This allows you to ensure the authenticity of the message. If Alice wants to send a message to Bob, and Bob wants to be sure that it is from Alice, Alice can sign the message using her private key. Bob can then verify that the message is from Alice by using her public key.



JBoss WS-Security support

JBoss Web Services supports many real world scenarios requiring WS-Security functionalities. This includes signature and encryption support through X509 certificates, authentication and authorization through username tokens as well as all ws-security configurations covered by WS-[SecurityPolicy](#) specification.

[As well as for other WS-* features](#), the core of WS-Security functionalities is provided through the Apache CXF engine. On top of that the JBossWS integration adds few configuration enhancements to simplify the setup of WS-Security enabled endpoints.

Apache CXF WS-Security implementation

Apache CXF features a top class WS-Security module supporting multiple configurations and easily extendible.

The system is based on *interceptors* that delegate to [Apache WSS4J](#) for the low level security operations. Interceptors can be configured in different ways, either through Spring configuration files or directly using Apache CXF client API. Please refer to the [Apache CXF documentation](#) if you're looking for more details.



Recent versions of Apache CXF, however, introduced support for WS-Security Policy, which aims at moving most of the security configuration into the service contract (through policies), so that clients can easily be configured almost completely automatically from that. This way users do not need to manually deal with configuring / installing the required interceptors; the Apache CXF WS-Policy engine internally takes care of that instead.

Security Policy support

WS-SecurityPolicy describes the actions that are required to securely communicate with a service advertised in a given WSDL contract. The WSDL bindings / operations reference WS-Policy fragments with the security requirements to interact with the service. The [WS-SecurityPolicy specification](#) allows for specifying things like asymmetric/symmetric keys, using transports (https) for encryption, which parts/headers to encrypt or sign, whether to sign then encrypt or encrypt then sign, whether to include timestamps, whether to use derived keys, etc.

However some mandatory configuration elements are not covered by WS-SecurityPolicy, basically because they're not meant to be public / part of the published endpoint contract; those include things such as keystore locations, usernames and passwords, etc. Apache CXF allows configuring these elements either through Spring xml descriptors or using the client API / annotations. Below is the list of supported configuration properties:



<code>ws-security.username</code>	The username used for UsernameToken policy assertions
<code>ws-security.password</code>	The password used for UsernameToken policy assertions. If not specified, the callback handler will be called.
<code>ws-security.callback-handler</code>	The WSS4J security CallbackHandler that will be used to retrieve passwords for keystores and UsernameTokens.
<code>ws-security.signature.properties</code>	The properties file/object that contains the WSS4J properties for configuring the signature keystore and crypto objects
<code>ws-security.encryption.properties</code>	The properties file/object that contains the WSS4J properties for configuring the encryption keystore and crypto objects
<code>ws-security.signature.username</code>	The username or alias for the key in the signature keystore that will be used. If not specified, it uses the the default alias set in the properties file. If that's also not set, and the keystore only contains a single key, that key will be used.
<code>ws-security.encryption.username</code>	The username or alias for the key in the encryption keystore that will be used. If not specified, it uses the the default alias set in the properties file. If that's also not set, and the keystore only contains a single key, that key will be used. For the web service provider, the <code>useReqSigCert</code> keyword can be used to accept (encrypt to) any client whose public key is in the service's truststore (defined in <code>ws-security.encryption.properties</code> .)
<code>ws-security.signature.crypto</code>	Instead of specifying the signature properties, this can point to the full WSS4J Crypto object. This can allow easier "programmatic" configuration of the Crypto information."
<code>ws-security.encryption.crypto</code>	Instead of specifying the encryption properties, this can point to the full WSS4J Crypto object. This can allow easier "programmatic" configuration of the Crypto information."
<code>ws-security.enable.streaming</code>	Enable streaming (StAX based) processing of WS-Security messages

Here is an example of configuration using the client API:

```
Map<String, Object> ctx = ((BindingProvider)port).getRequestContext();
ctx.put("ws-security.encryption.properties", properties);
port.echoString("hello");
```

Please refer to the [Apache CXF documentation](#) for additional configuration details.



JBossWS configuration additions

In order for removing the need of Spring on server side for setting up WS-Security configuration properties not covered by policies, the JBossWS integration allows for getting those pieces of information from a defined *endpoint configuration*. [Endpoint configurations](#) can include property declarations and endpoint implementations can be associated with a given endpoint configuration using the `@EndpointConfig` annotation.

```
<?xml version="1.0" encoding="UTF-8"?>
<jaxws-config xmlns="urn:jboss:jbossws-jaxws-config:4.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:javaee="http://java.sun.com/xml/ns/javaee"
  xsi:schemaLocation="urn:jboss:jbossws-jaxws-config:4.0 schema/jbossws-jaxws-config_4_0.xsd">
  <endpoint-config>
    <config-name>Custom WS-Security Endpoint</config-name>
    <property>
      <property-name>ws-security.signature.properties</property-name>
      <property-value>bob.properties</property-value>
    </property>
    <property>
      <property-name>ws-security.encryption.properties</property-name>
      <property-value>bob.properties</property-value>
    </property>
    <property>
      <property-name>ws-security.signature.username</property-name>
      <property-value>bob</property-value>
    </property>
    <property>
      <property-name>ws-security.encryption.username</property-name>
      <property-value>alice</property-value>
    </property>
    <property>
      <property-name>ws-security.callback-handler</property-name>
      <property-value>org.jboss.test.ws.jaxws.samples.wsse.policy.basic.KeystorePasswordCallback</property-value>
    </property>
  </endpoint-config>
</jaxws-config>
```




```
import javax.jws.WebService;
import org.jboss.ws.api.annotation.EndpointConfig;

@WebService
(
    portName = "SecurityServicePort",
    serviceName = "SecurityService",
    wsdlLocation = "WEB-INF/wsdl/SecurityService.wsdl",
    targetNamespace = "http://www.jboss.org/jbossws/ws-extensions/wssecuritypolicy",
    endpointInterface = "org.jboss.test.ws.jaxws.samples.wsse.policy.basic.ServiceIface"
)
@EndpointConfig(configFile = "WEB-INF/jaxws-endpoint-config.xml", configName = "Custom
WS-Security Endpoint")
public class ServiceImpl implements ServiceIface
{
    public String sayHello()
    {
        return "Secure Hello World!";
    }
}
```

Apache CXF annotations

The JBossWS configuration additions allow for a descriptor approach to the WS-Security Policy engine configuration. If you prefer to provide the same information through an annotation approach, you can leverage the Apache CXF `@org.apache.cxf.annotations.EndpointProperties` annotation:

```
@WebService(
    ...
)
@EndpointProperties(value = {
    @EndpointProperty(key = "ws-security.signature.properties", value = "bob.properties"),
    @EndpointProperty(key = "ws-security.encryption.properties", value = "bob.properties"),
    @EndpointProperty(key = "ws-security.signature.username", value = "bob"),
    @EndpointProperty(key = "ws-security.encryption.username", value = "alice"),
    @EndpointProperty(key = "ws-security.callback-handler", value =
"org.jboss.test.ws.jaxws.samples.wsse.policy.basic.KeystorePasswordCallback")
})
public class ServiceImpl implements ServiceIface {
    ...
}
```

Examples

In this section some sample of WS-Security service endpoints and clients are provided. Please note they're only meant as tutorials; you should really careful isolate the ws-security policies / assertion that best suite your security needs before going to production environment.



⊖ The following sections provide directions and examples on understanding some of the configuration options for WS-Security engine. Please note the implementor remains responsible for assessing the application requirements and choosing the most suitable security policy for them.

Signature and encryption

Endpoint

First of all you need to create the web service endpoint using JAX-WS. While this can generally be achieved in different ways, it's required to use a contract-first approach when using WS-Security, as the policies declared in the wsdl are parsed by the Apache CXF engine on both server and client sides. So, here is an example of WSDL contract enforcing signature and encryption using X 509 certificates (the referenced schema is omitted):

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<definitions targetNamespace="http://www.jboss.org/jboss/ws-extensions/wssecuritypolicy"
name="SecurityService"
  xmlns:tns="http://www.jboss.org/jboss/ws-extensions/wssecuritypolicy"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:wsp="http://www.w3.org/ns/ws-policy"

  xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
  xmlns:wsaws="http://www.w3.org/2005/08/addressing"
  xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy">
  <types>
    <xsd:schema>
      <xsd:import namespace="http://www.jboss.org/jboss/ws-extensions/wssecuritypolicy"
schemaLocation="SecurityService_schema1.xsd"/>
    </xsd:schema>
  </types>
  <message name="sayHello">
    <part name="parameters" element="tns:sayHello"/>
  </message>
  <message name="sayHelloResponse">
    <part name="parameters" element="tns:sayHelloResponse"/>
  </message>
  <portType name="ServiceIface">
    <operation name="sayHello">
      <input message="tns:sayHello"/>
      <output message="tns:sayHelloResponse"/>
    </operation>
  </portType>
  <binding name="SecurityServicePortBinding" type="tns:ServiceIface">
    <wsp:PolicyReference URI="#SecurityServiceSignThenEncryptPolicy"/>
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
    <operation name="sayHello">
      <soap:operation soapAction="" />
      <input>
        <soap:body use="literal"/>
      </input>
    </operation>
  </binding>
</definitions>
```



```
<output>
  <soap:body use="literal"/>
</output>
</operation>
</binding>
<service name="SecurityService">
  <port name="SecurityServicePort" binding="tns:SecurityServicePortBinding">
    <soap:address location="http://localhost:8080/jaxws-samples-wssePolicy-sign-encrypt"/>
  </port>
</service>

<wsp:Policy wsu:Id="SecurityServiceSignThenEncryptPolicy"
xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy">
  <wsp:ExactlyOne>
    <wsp>All>
      <sp:AsymmetricBinding xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy">
        <wsp:Policy>
          <sp:InitiatorToken>
            <wsp:Policy>
              <sp:X509Token
sp:IncludeToken="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy/IncludeToken/AlwaysToRecipient"
<wsp:Policy>
                <sp:WssX509V1Token11/>
              </wsp:Policy>
            </sp:X509Token>
          </wsp:Policy>
          <sp:InitiatorToken>
            <sp:RecipientToken>
              <wsp:Policy>
                <sp:X509Token
sp:IncludeToken="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy/IncludeToken/Never"
                <wsp:Policy>
                  <sp:WssX509V1Token11/>
                </wsp:Policy>
              </sp:X509Token>
            </wsp:Policy>
            <sp:RecipientToken>
              <sp:AlgorithmSuite>
                <wsp:Policy>
                  <sp:TripleDesRsa15/>
                </wsp:Policy>
              </sp:AlgorithmSuite>
            <sp:Layout>
              <wsp:Policy>
                <sp:Lax/>
              </wsp:Policy>
            </sp:Layout>
            <sp:IncludeTimestamp/>
            <sp:EncryptSignature/>
            <sp:OnlySignEntireHeadersAndBody/>
            <sp:SignBeforeEncrypting/>
          </wsp:Policy>
        </sp:AsymmetricBinding>
      <sp:SignedParts xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy">
        <sp:Body/>
      </sp:SignedParts>
      <sp:EncryptedParts xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy">
        <sp:Body/>
      </sp:EncryptedParts>
    </wsp>All>
  </wsp:ExactlyOne>
</wsp:Policy>
```



```
</sp:EncryptedParts>
<sp:Wss10 xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy">
  <wsp:Policy>
    <sp:MustSupportRefIssuerSerial/>
  </wsp:Policy>
</sp:Wss10>
</wsp:All>
</wsp:ExactlyOne>
</wsp:Policy>
</definitions>
```

The service endpoint can be generated using the `wsconsume` tool and then enriched with a `@EndpointConfig` annotation:

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.basic;

import javax.jws.WebService;
import org.jboss.ws.api.annotation.EndpointConfig;

@WebService
(
  portName = "SecurityServicePort",
  serviceName = "SecurityService",
  wsdlLocation = "WEB-INF/wsdl/SecurityService.wsdl",
  targetNamespace = "http://www.jboss.org/jbossws/ws-extensions/wssecuritypolicy",
  endpointInterface = "org.jboss.test.ws.jaxws.samples.wsse.policy.basic.ServiceIface"
)
@EndpointConfig(configFile = "WEB-INF/jaxws-endpoint-config.xml", configName = "Custom
WS-Security Endpoint")
public class ServiceImpl implements ServiceIface
{
  public String sayHello()
  {
    return "Secure Hello World!";
  }
}
```

The referenced `jaxws-endpoint-config.xml` descriptor is used to provide a custom endpoint configuration with the required server side configuration properties; this tells the engine which certificate / key to use for signature / signature verification and for encryption / decryption:



```
<?xml version="1.0" encoding="UTF-8"?>
<jaxws-config xmlns="urn:jboss:jbossws-jaxws-config:4.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:javaee="http://java.sun.com/xml/ns/javaee"
  xsi:schemaLocation="urn:jboss:jbossws-jaxws-config:4.0 schema/jbossws-jaxws-config_4_0.xsd">
  <endpoint-config>
    <config-name>Custom WS-Security Endpoint</config-name>
    <property>
      <property-name>ws-security.signature.properties</property-name>
      <property-value>bob.properties</property-value>
    </property>
    <property>
      <property-name>ws-security.encryption.properties</property-name>
      <property-value>bob.properties</property-value>
    </property>
    <property>
      <property-name>ws-security.signature.username</property-name>
      <property-value>bob</property-value>
    </property>
    <property>
      <property-name>ws-security.encryption.username</property-name>
      <property-value>alice</property-value>
    </property>
    <property>
      <property-name>ws-security.callback-handler</property-name>
      <property-value>org.jboss.test.ws.jaxws.samples.wsse.policy.basic.KeystorePasswordCallback</property-value>
    </property>
  </endpoint-config>
</jaxws-config>
```

... the *bob.properties* configuration file is also referenced above; it includes the WSS4J Crypto properties which in turn link to the keystore file, type and the alias/password to use for accessing it:

```
org.apache.ws.security.crypto.provider=org.apache.ws.security.components.crypto.Merlin
org.apache.ws.security.crypto.merlin.keystore.type=jks
org.apache.ws.security.crypto.merlin.keystore.password=password
org.apache.ws.security.crypto.merlin.keystore.alias=bob
org.apache.ws.security.crypto.merlin.keystore.file=bob.jks
```

A callback handler for the letting Apache CXF access the keystore is also provided:



```
package org.jboss.test.ws.jaxws.samples.wsse.policy.basic;

import java.io.IOException;
import java.util.HashMap;
import java.util.Map;
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.UnsupportedCallbackException;
import org.apache.ws.security.WSPasswordCallback;

public class KeystorePasswordCallback implements CallbackHandler {
    private Map<String, String> passwords = new HashMap<String, String>();

    public KeystorePasswordCallback() {
        passwords.put("alice", "password");
        passwords.put("bob", "password");
    }

    /**
     * It attempts to get the password from the private
     * alias/passwords map.
     */
    public void handle(Callback[] callbacks) throws IOException, UnsupportedCallbackException {
        for (int i = 0; i < callbacks.length; i++) {
            WSPasswordCallback pc = (WSPasswordCallback)callbacks[i];

            String pass = passwords.get(pc.getIdentifier());
            if (pass != null) {
                pc.setPassword(pass);
                return;
            }
        }
    }

    /**
     * Add an alias/password pair to the callback mechanism.
     */
    public void setAliasPassword(String alias, String password) {
        passwords.put(alias, password);
    }
}
```

Assuming the *bob.jks* keystore has been properly generated and contains Bob's (server) full key (private/certificate + public key) as well as Alice's (client) public key, we can proceed to packaging the endpoint. Here is the expected content (the endpoint is a *POJO* one in a *war* archive, but *EJB3* endpoints in *jar* archives are of course also supported):



```
alessio@inuyasha /dati/jboss/ws/stack/cxf/trunk $ jar -tvf
./modules/testsuite/cxf-tests/target/test-libs/jaxws-samples-wsse-policy-sign-encrypt.war
 0 Thu Jun 16 18:50:48 CEST 2011 META-INF/
140 Thu Jun 16 18:50:46 CEST 2011 META-INF/MANIFEST.MF
 0 Thu Jun 16 18:50:48 CEST 2011 WEB-INF/
586 Thu Jun 16 18:50:44 CEST 2011 WEB-INF/web.xml
 0 Thu Jun 16 18:50:48 CEST 2011 WEB-INF/classes/
 0 Thu Jun 16 18:50:48 CEST 2011 WEB-INF/classes/org/
 0 Thu Jun 16 18:50:48 CEST 2011 WEB-INF/classes/org/jboss/
 0 Thu Jun 16 18:50:48 CEST 2011 WEB-INF/classes/org/jboss/test/
 0 Thu Jun 16 18:50:48 CEST 2011 WEB-INF/classes/org/jboss/test/ws/
 0 Thu Jun 16 18:50:48 CEST 2011 WEB-INF/classes/org/jboss/test/ws/jaxws/
 0 Thu Jun 16 18:50:48 CEST 2011 WEB-INF/classes/org/jboss/test/ws/jaxws/samples/
 0 Thu Jun 16 18:50:48 CEST 2011 WEB-INF/classes/org/jboss/test/ws/jaxws/samples/wsse/
 0 Thu Jun 16 18:50:48 CEST 2011
WEB-INF/classes/org/jboss/test/ws/jaxws/samples/wsse/policy/
 0 Thu Jun 16 18:50:48 CEST 2011
WEB-INF/classes/org/jboss/test/ws/jaxws/samples/wsse/policy/basic/
1687 Thu Jun 16 18:50:48 CEST 2011
WEB-INF/classes/org/jboss/test/ws/jaxws/samples/wsse/policy/basic/KeystorePasswordCallback.class
383 Thu Jun 16 18:50:48 CEST 2011
WEB-INF/classes/org/jboss/test/ws/jaxws/samples/wsse/policy/basic/ServiceIface.class
1070 Thu Jun 16 18:50:48 CEST 2011
WEB-INF/classes/org/jboss/test/ws/jaxws/samples/wsse/policy/basic/ServiceImpl.class
 0 Thu Jun 16 18:50:48 CEST 2011
WEB-INF/classes/org/jboss/test/ws/jaxws/samples/wsse/policy/jaxws/
705 Thu Jun 16 18:50:48 CEST 2011
WEB-INF/classes/org/jboss/test/ws/jaxws/samples/wsse/policy/jaxws/SayHello.class
1069 Thu Jun 16 18:50:48 CEST 2011
WEB-INF/classes/org/jboss/test/ws/jaxws/samples/wsse/policy/jaxws/SayHelloResponse.class
1225 Thu Jun 16 18:50:44 CEST 2011 WEB-INF/jaxws-endpoint-config.xml
 0 Thu Jun 16 18:50:44 CEST 2011 WEB-INF/wsd/
4086 Thu Jun 16 18:50:44 CEST 2011 WEB-INF/wsd/SecurityService.wsdl
653 Thu Jun 16 18:50:44 CEST 2011 WEB-INF/wsd/SecurityService_schema1.xsd
1820 Thu Jun 16 18:50:44 CEST 2011 WEB-INF/classes/bob.jks
311 Thu Jun 16 18:50:44 CEST 2011 WEB-INF/classes/bob.properties
```

As you can see, the jaxws classes generated by the tools are of course also included, as well as a basic *web.xml* referencing the endpoint bean:



```
<?xml version="1.0" encoding="UTF-8"?>
<web-app
  version="2.5" 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
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
  <servlet>
    <servlet-name>TestService</servlet-name>

<servlet-class>org.jboss.test.ws.jaxws.samples.wsse.policy.basic.ServiceImpl</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>TestService</servlet-name>
    <url-pattern>/*</url-pattern>
  </servlet-mapping>
</web-app>
```



If you're deploying the endpoint archive on WildFly, remember to add a dependency to *org.apache.ws.security* module in the MANIFEST.MF file.

```
Manifest-Version: 1.0
Ant-Version: Apache Ant 1.7.1
Created-By: 17.0-b16 (Sun Microsystems Inc.)
Dependencies: org.apache.ws.security
```




Client

You start by consuming the published WSDL contract using the *wconsume* tool on client side too. Then you simply invoke the the endpoint as a standard JAX-WS one:

```
QName serviceName = new QName("http://www.jboss.org/jbossws/ws-extensions/wssecuritypolicy",
    "SecurityService");
URL wsdlURL = new URL(serviceURL + "?wsdl");
Service service = Service.create(wsdlURL, serviceName);
ServiceIface proxy = (ServiceIface)service.getPort(ServiceIface.class);

((BindingProvider)proxy).getRequestContext().put(SecurityConstants.CALLBACK_HANDLER, new
    KeystorePasswordCallback());
((BindingProvider)proxy).getRequestContext().put(SecurityConstants.SIGNATURE_PROPERTIES,
    Thread.currentThread().getContextClassLoader().getResource("META-INF/alice.properties"));
((BindingProvider)proxy).getRequestContext().put(SecurityConstants.ENCRYPT_PROPERTIES,
    Thread.currentThread().getContextClassLoader().getResource("META-INF/alice.properties"));
((BindingProvider)proxy).getRequestContext().put(SecurityConstants.SIGNATURE_USERNAME, "alice");
((BindingProvider)proxy).getRequestContext().put(SecurityConstants.ENCRYPT_USERNAME, "bob");

proxy.sayHello();
```

As you can see, the WS-Security properties are set in the request context. Here the *KeystorePasswordCallback* is the same as on server side above, you might want/need different implementation in real world scenarios, of course.

The *alice.properties* file is the client side equivalent of the server side *bob.properties* and references the *alice.jks* keystore file, which has been populated with Alice's (client) full key (private/certificate + public key) as well as Bob's (server) public key.

```
org.apache.ws.security.crypto.provider=org.apache.ws.security.components.crypto.Merlin
org.apache.ws.security.crypto.merlin.keystore.type=jks
org.apache.ws.security.crypto.merlin.keystore.password=password
org.apache.ws.security.crypto.merlin.keystore.alias=alice
org.apache.ws.security.crypto.merlin.keystore.file=META-INF/alice.jks
```

The Apache CXF WS-Policy engine will digest the security requirements in the contract and ensure a valid secure communication is in place for interacting with the server endpoint.

Endpoint serving multiple clients

The server side configuration described above implies the endpoint is configured for serving a given client which a service agreement has been established for. In some real world scenarios though, the same server might be expected to be able to deal with (including decrypting and encrypting) messages coming from and being sent to multiple clients. Apache CXF supports that through the *useReqSigCert* value for the *ws-security.encryption.username* configuration parameter.

Of course the referenced server side keystore then needs to contain the public key of all the clients that are expected to be served.



Authentication and authorization

The Username Token Profile can be used to provide client's credentials to a WS-Security enabled target endpoint.

Apache CXF provides means for setting basic *password callback handlers* on both client and server sides to set/check passwords; the `ws-security.username` and `ws-security.callback-handler` properties can be used similarly as shown in the signature and encryption example. Things become more interesting when requiring a given user to be authenticated (and authorized) against a security domain on the target application server.

On server side, you need to install two additional interceptors that act as bridges towards the application server authentication layer:

- an interceptor for performing authentication and populating a valid SecurityContext; the provided interceptor should extend `org.apache.cxf.ws.interceptor.security.AbstractUsernameTokenInInterceptor`, in particular JBossWS integration comes with `org.jboss.wsf.stack.cxf.security.authentication.SubjectCreatingInterceptor` for this;
- an interceptor for performing authorization; CXF requires that to extend `org.apache.cxf.interceptor.security.AbstractAuthorizingInInterceptor`, for instance the `SimpleAuthorizingInterceptor` can be used for simply mapping endpoint operations to allowed roles.

So, here follows an example of WS-SecurityPolicy endpoint using Username Token Profile for authenticating through the application server security domain system.

Endpoint

As in the other example, we start with a wsdl contract containing the proper WS-Security Policy:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<definitions targetNamespace="http://www.jboss.org/jbossws/ws-extensions/wssecuritypolicy"
name="SecurityService"
  xmlns:tns="http://www.jboss.org/jbossws/ws-extensions/wssecuritypolicy"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"

  xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
  xmlns:wsaws="http://www.w3.org/2005/08/addressing">
  <types>
    <xsd:schema>
      <xsd:import namespace="http://www.jboss.org/jbossws/ws-extensions/wssecuritypolicy"
schemaLocation="SecurityService_schemal.xsd"/>
    </xsd:schema>
  </types>
  <message name="sayHello">
    <part name="parameters" element="tns:sayHello"/>
  </message>
  <message name="sayHelloResponse">
    <part name="parameters" element="tns:sayHelloResponse"/>
  </message>
  <message name="greetMe">
```



```
<part name="parameters" element="tns:greetMe" />
</message>
<message name="greetMeResponse">
  <part name="parameters" element="tns:greetMeResponse" />
</message>
<portType name="ServiceIface">
  <operation name="sayHello">
    <input message="tns:sayHello" />
    <output message="tns:sayHelloResponse" />
  </operation>
  <operation name="greetMe">
    <input message="tns:greetMe" />
    <output message="tns:greetMeResponse" />
  </operation>
</portType>
<binding name="SecurityServicePortBinding" type="tns:ServiceIface">
  <wsp:PolicyReference URI="#SecurityServiceUsernameUnsecureTransportPolicy" />
  <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document" />
  <operation name="sayHello">
    <soap:operation soapAction="" />
    <input>
      <soap:body use="literal" />
    </input>
    <output>
      <soap:body use="literal" />
    </output>
  </operation>
  <operation name="greetMe">
    <soap:operation soapAction="" />
    <input>
      <soap:body use="literal" />
    </input>
    <output>
      <soap:body use="literal" />
    </output>
  </operation>
</binding>
<service name="SecurityService">
  <port name="SecurityServicePort" binding="tns:SecurityServicePortBinding">
    <soap:address location="http://localhost:8080/jaxws-samples-wsse-username-jaas" />
  </port>
</service>

<wsp:Policy wsu:Id="SecurityServiceUsernameUnsecureTransportPolicy">
  <wsp:ExactlyOne>
    <wsp:All>
      <sp:SupportingTokens
xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
        <wsp:Policy>
          <sp:UsernameToken
sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/IncludeToken/AlwaysToRe
<wsp:Policy>
            <sp:WssUsernameToken10/>
          </sp:Policy>
        </sp:UsernameToken>
      </wsp:Policy>
    </sp:SupportingTokens>
  </wsp:All>
```



```
</wsp:ExactlyOne>
</wsp:Policy>

</definitions>
```

i If you want to send hash / digest passwords, you can use a policy such as what follows:

```
<wsp:Policy wsu:Id="SecurityServiceUsernameHashPasswordPolicy">
  <wsp:ExactlyOne>
    <wsp:All>
      <sp:SupportingTokens
xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
        <wsp:Policy>
          <sp:UsernameToken
sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/IncludeToken/Alwa
<wsp:Policy>
              <sp:HashPassword/>
            </wsp:Policy>
          </sp:UsernameToken>
        </wsp:Policy>
      </sp:SupportingTokens>
    </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>
```

Please note the specified JBoss security domain needs to be properly configured for computing digests.

The service endpoint can be generated using the `wsconsume` tool and then enriched with a `@EndpointConfig` annotation and `@InInterceptors` annotation to add the two interceptors mentioned above for JAAS integration:



```
package org.jboss.test.ws.jaxws.samples.wsse.policy.jaas;

import javax.jws.WebService;
import org.apache.cxf.interceptor.InInterceptors;
import org.jboss.ws.api.annotation.EndpointConfig;

@WebService
(
    portName = "SecurityServicePort",
    serviceName = "SecurityService",
    wsdlLocation = "WEB-INF/wsdl/SecurityService.wsdl",
    targetNamespace = "http://www.jboss.org/jbossws/ws-extensions/wssecuritypolicy",
    endpointInterface = "org.jboss.test.ws.jaxws.samples.wsse.policy.jaas.ServiceIface"
)
@EndpointConfig(configFile = "WEB-INF/jaxws-endpoint-config.xml", configName = "Custom
WS-Security Endpoint")
@InInterceptors(interceptors = {
    "org.jboss.wsf.stack.cxf.security.authentication.SubjectCreatingPolicyInterceptor",
    "org.jboss.test.ws.jaxws.samples.wsse.policy.jaas.POJOEndpointAuthorizationInterceptor"}
)
public class ServiceImpl implements ServiceIface
{
    public String sayHello()
    {
        return "Secure Hello World!";
    }

    public String greetMe()
    {
        return "Greetings!";
    }
}
```

The `POJOEndpointAuthorizationInterceptor` is included into the deployment and deals with the roles checks:



```
package org.jboss.test.ws.jaxws.samples.wsse.policy.jaas;

import java.util.HashMap;
import java.util.Map;
import org.apache.cxf.interceptor.security.SimpleAuthorizingInterceptor;

public class POJOEndpointAuthorizationInterceptor extends SimpleAuthorizingInterceptor
{

    public POJOEndpointAuthorizationInterceptor()
    {
        super();
        readRoles();
    }

    private void readRoles()
    {
        //just an example, this might read from a configuration file or such
        Map<String, String> roles = new HashMap<String, String>();
        roles.put("sayHello", "friend");
        roles.put("greetMe", "snoopies");
        setMethodRolesMap(roles);
    }
}
```

The *jaxws-endpoint-config.xml* descriptor is used to provide a custom endpoint configuration with the required server side configuration properties; in particular for this Username Token case that's just a CXF configuration option for leaving the username token validation to the configured interceptors:

```
<?xml version="1.0" encoding="UTF-8"?>
<jaxws-config xmlns="urn:jboss:jbossws-jaxws-config:4.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:javaee="http://java.sun.com/xml/ns/javaee"
xsi:schemaLocation="urn:jboss:jbossws-jaxws-config:4.0 schema/jbossws-jaxws-config_4_0.xsd">
  <endpoint-config>
    <config-name>Custom WS-Security Endpoint</config-name>
    <property>
      <property-name>ws-security.validate.token</property-name>
      <property-value>>false</property-value>
    </property>
  </endpoint-config>
</jaxws-config>
```

In order for requiring a given JBoss security domain to be used to protect access to the endpoint (a POJO one in this case), we declare that in a *jboss-web.xml* descriptor (the *JBossWS* security domain is used):



```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jboss-web PUBLIC "-//JBoss//DTD Web Application 2.4//EN"
"http://www.jboss.org/j2ee/dtd/jboss-web_4_0.dtd">
<jboss-web>
  <security-domain>java:/jaas/JBossWS</security-domain>
</jboss-web>
```

Finally, the *web.xml* is as simple as usual:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app
  version="2.5" 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
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
  <servlet>
    <servlet-name>TestService</servlet-name>

<servlet-class>org.jboss.test.ws.jaxws.samples.wsse.policy.jaas.ServiceImpl</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>TestService</servlet-name>
    <url-pattern>*</url-pattern>
  </servlet-mapping>
</web-app>
```

The endpoint is packaged into a war archive, including the JAXWS classes generated by wsconsume:



```
alessio@inuyasha /dati/jboss/ws/stack/cxf/trunk $ jar -tvf
./modules/testsuite/cxf-tests/target/test-libs/jaxws-samples-wsse-policy-username-jaas.war
 0 Thu Jun 16 18:50:48 CEST 2011 META-INF/
155 Thu Jun 16 18:50:46 CEST 2011 META-INF/MANIFEST.MF
 0 Thu Jun 16 18:50:48 CEST 2011 WEB-INF/
585 Thu Jun 16 18:50:44 CEST 2011 WEB-INF/web.xml
 0 Thu Jun 16 18:50:48 CEST 2011 WEB-INF/classes/
 0 Thu Jun 16 18:50:48 CEST 2011 WEB-INF/classes/org/
 0 Thu Jun 16 18:50:48 CEST 2011 WEB-INF/classes/org/jboss/
 0 Thu Jun 16 18:50:48 CEST 2011 WEB-INF/classes/org/jboss/test/
 0 Thu Jun 16 18:50:48 CEST 2011 WEB-INF/classes/org/jboss/test/ws/
 0 Thu Jun 16 18:50:48 CEST 2011 WEB-INF/classes/org/jboss/test/ws/jaxws/
 0 Thu Jun 16 18:50:48 CEST 2011 WEB-INF/classes/org/jboss/test/ws/jaxws/samples/
 0 Thu Jun 16 18:50:48 CEST 2011 WEB-INF/classes/org/jboss/test/ws/jaxws/samples/wsse/
 0 Thu Jun 16 18:50:48 CEST 2011
WEB-INF/classes/org/jboss/test/ws/jaxws/samples/wsse/policy/
 0 Thu Jun 16 18:50:48 CEST 2011
WEB-INF/classes/org/jboss/test/ws/jaxws/samples/wsse/policy/jaas/
 982 Thu Jun 16 18:50:48 CEST 2011
WEB-INF/classes/org/jboss/test/ws/jaxws/samples/wsse/policy/jaas/POJOEndpointAuthorizationIntercep
412 Thu Jun 16 18:50:48 CEST 2011
WEB-INF/classes/org/jboss/test/ws/jaxws/samples/wsse/policy/jaas/ServiceIface.class
1398 Thu Jun 16 18:50:48 CEST 2011
WEB-INF/classes/org/jboss/test/ws/jaxws/samples/wsse/policy/jaas/ServiceImpl.class
 0 Thu Jun 16 18:50:48 CEST 2011
WEB-INF/classes/org/jboss/test/ws/jaxws/samples/wsse/policy/jaxws/
 701 Thu Jun 16 18:50:48 CEST 2011
WEB-INF/classes/org/jboss/test/ws/jaxws/samples/wsse/policy/jaxws/GreetMe.class
1065 Thu Jun 16 18:50:48 CEST 2011
WEB-INF/classes/org/jboss/test/ws/jaxws/samples/wsse/policy/jaxws/GreetMeResponse.class
 705 Thu Jun 16 18:50:48 CEST 2011
WEB-INF/classes/org/jboss/test/ws/jaxws/samples/wsse/policy/jaxws/SayHello.class
1069 Thu Jun 16 18:50:48 CEST 2011
WEB-INF/classes/org/jboss/test/ws/jaxws/samples/wsse/policy/jaxws/SayHelloResponse.class
 556 Thu Jun 16 18:50:44 CEST 2011 WEB-INF/jaxws-endpoint-config.xml
 241 Thu Jun 16 18:50:44 CEST 2011 WEB-INF/jboss-web.xml
 0 Thu Jun 16 18:50:44 CEST 2011 WEB-INF/wsdl/
3183 Thu Jun 16 18:50:44 CEST 2011 WEB-INF/wsdl/SecurityService.wsdl
1012 Thu Jun 16 18:50:44 CEST 2011 WEB-INF/wsdl/SecurityService_schema1.xsd
```



If you're deploying the endpoint archive on WildFly, remember to add a dependency to *org.apache.ws.security* and *org.apache.cxf* module (due to the `@InInterceptor` annotation) in the MANIFEST.MF file.

```
Manifest-Version: 1.0
Ant-Version: Apache Ant 1.7.1
Created-By: 17.0-b16 (Sun Microsystems Inc.)
Dependencies: org.apache.ws.security,org.apache.cxf
```




Client

Here too you start by consuming the published WSDL contract using the *wconsume* tool. Then you simply invoke the the endpoint as a standard JAX-WS one:

```
QName serviceName = new QName("http://www.jboss.org/jbossws/ws-extensions/wssecuritypolicy",
    "SecurityService");
URL wsdlURL = new URL(serviceURL + "?wsdl");
Service service = Service.create(wsdlURL, serviceName);
ServiceIface proxy = (ServiceIface)service.getPort(ServiceIface.class);

((BindingProvider)proxy).getRequestContext().put(SecurityConstants.USERNAME, "kermit");
((BindingProvider)proxy).getRequestContext().put(SecurityConstants.CALLBACK_HANDLER,
    "org.jboss.test.ws.jaxws.samples.wsse.policy.jaas.UsernamePasswordCallback");

proxy.sayHello();
```

The `UsernamePasswordCallback` class is shown below and is responsible for setting the passwords on client side just before performing the invocations:

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.jaas;

import java.io.IOException;
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.UnsupportedCallbackException;
import org.apache.ws.security.WSPasswordCallback;

public class UsernamePasswordCallback implements CallbackHandler
{
    public void handle(Callback[] callbacks) throws IOException, UnsupportedCallbackException
    {
        WSPasswordCallback pc = (WSPasswordCallback)callbacks[0];
        if ("kermit".equals(pc.getIdentifier()))
            pc.setPassword("thefrog");
    }
}
```

If everything has been done properly, you should expect to calls to `sayHello()` fail when done with user "snoopy" and pass with user "kermit" (and credential "thefrog"); moreover, you should get an authorization error when trying to call `greetMe()` with user "kermit", as that does not have the "snoopies" role.

Secure transport

Another quite common use case is using WS-Security Username Token Profile over a secure transport (HTTPS). A scenario like this is implemented similarly to what's described in the previous example, except for few differences explained below.

First of all, here is an excerpt of a wsdl with a sample security policy for Username Token over HTTPS:

```
...
```



```
<binding name="SecurityServicePortBinding" type="tns:ServiceIface">
  <wsp:PolicyReference URI="#SecurityServiceBindingPolicy"/>
  <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
  <operation name="sayHello">
    <soap:operation soapAction=""/>
    <input>
      <soap:body use="literal"/>
    </input>
    <output>
      <soap:body use="literal"/>
    </output>
  </operation>
</binding>
<service name="SecurityService">
  <port name="SecurityServicePort" binding="tns:SecurityServicePortBinding">
    <soap:address location="https://localhost:8443/jaxws-samples-wsse-policy-username"/>
  </port>
</service>

<wsp:Policy wsu:Id="SecurityServiceBindingPolicy">
  <wsp:ExactlyOne>
    <wsp:All>
      <foo:unknownPolicy xmlns:foo="http://cxf.apache.org/not/a/policy"/>
    </wsp:All>
    <wsp:All>
      <wsaws:UsingAddressing xmlns:wsaws="http://www.w3.org/2006/05/addressing/wsdl"/>
      <sp:TransportBinding>
        <wsp:Policy>
          <sp:TransportToken>
            <wsp:Policy>
              <sp:HttpsToken RequireClientCertificate="false"/>
            </wsp:Policy>
          </sp:TransportToken>
          <sp:Layout>
            <wsp:Policy>
              <sp:Lax/>
            </wsp:Policy>
          </sp:Layout>
          <sp:IncludeTimestamp/>
          <sp:AlgorithmSuite>
            <wsp:Policy>
              <sp:Basic128/>
            </wsp:Policy>
          </sp:AlgorithmSuite>
        </wsp:Policy>
      </sp:TransportBinding>
      <sp:Wss10>
        <wsp:Policy>
          <sp:MustSupportRefKeyIdentifier/>
        </wsp:Policy>
      </sp:Wss10>
      <sp:SignedSupportingTokens>
        <wsp:Policy>
          <sp:UsernameToken
            sp:IncludeToken="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy/IncludeToken/AlwaysToRecipient"
          >
        </wsp:Policy>
          <sp:WssUsernameToken10/>
        </sp:Policy>
      </sp:SignedSupportingTokens>
    </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>
```



```
        </wsp:Policy>
    </sp:UsernameToken>
    </wsp:Policy>
</sp:SignedSupportingTokens>
</wsp:All>
</wsp:ExactlyOne>
</wsp:Policy>
```

The endpoint then needs of course to be actually available on HTTPS only, so we have a *web.xml* setting the *transport-guarantee* such as below:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app
  version="2.5" 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
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
  <servlet>
    <servlet-name>TestService</servlet-name>

<servlet-class>org.jboss.test.ws.jaxws.samples.wsse.policy.basic.ServiceImpl</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>TestService</servlet-name>
    <url-pattern>/*</url-pattern>
  </servlet-mapping>

  <security-constraint>
    <web-resource-collection>
      <web-resource-name>TestService</web-resource-name>
      <url-pattern>/*</url-pattern>
    </web-resource-collection>
    <user-data-constraint>
      <transport-guarantee>CONFIDENTIAL</transport-guarantee>
    </user-data-constraint>
  </security-constraint>
</web-app>
```



Secure conversation

Apache CXF supports [WS-SecureConversation](#) specification, which is about improving performance by allowing client and server to negotiate initial security keys to be used for later communication encryption/signature. This is done by having two policies in the wsdl contract, an outer one setting the security requirements to actually communicate with the endpoint and a bootstrap one, related to the communication for establishing the secure conversation keys. The client will be automatically sending an initial message to the server for negotiating the keys, then the actual communication to the endpoint takes place. As a consequence, Apache CXF needs a way to specify which WS-Security configuration properties are intended for the bootstrap policy and which are intended for the actual service policy. To accomplish this, properties intended for the bootstrap policy are appended with `.sct`.

```
...
((BindingProvider)proxy).getRequestContext().put("ws-security.signature.username.sct", "alice");
((BindingProvider)proxy).getRequestContext().put("ws-security.encryption.username.sct", "bob");
...
```

```
@WebService(
    ...
)
@EndpointProperties(value = {
    @EndpointProperty(key = "ws-security.encryption.properties.sct", value =
"bob.properties"),
    @EndpointProperty(key = "ws-security.signature.properties.sct", value = "bob.properties"),
    ...
})
}
)
public class ServiceImpl implements ServiceIface {
    ...
}
```



35.3.13 Trust and STS

- [WS-Trust overview](#)
- [Security Token Service](#)
- [Apache CXF support](#)
- [A Basic WS-Trust Scenario](#)
 - [Web service provider](#)
 - [Web service provider WSDL](#)
 - [Web service provider Interface](#)
 - [Web service provider Implementation](#)
 - [ServerCallbackHandler](#)
 - [Crypto properties and keystore files](#)
 - [MANIFEST.MF](#)
 - [Security Token Service \(STS\)](#)
 - [STS WSDL](#)
 - [STS Implementation](#)
 - [STSCallbackHandler](#)
 - [Crypto properties and keystore files](#)
 - [MANIFEST.MF](#)
 - [Security Domain](#)
 - [Web service requester](#)
 - [Web service requester Implementation](#)
 - [ClientCallbackHandler](#)
 - [Requester Crypto properties and keystore files](#)
 - [PicketLink STS](#)



Trust overview

[WS-Trust](#) is a Web service specification that defines extensions to WS-Security. It is a general framework for implementing security in a distributed system. The standard is based on a centralized Security Token Service, STS, which is capable of authenticating clients and issuing tokens containing various kinds of authentication and authorization data. The specification describes a protocol used for issuance, exchange, and validation of security tokens, however the following specifications play an important role in the WS-Trust architecture: [WS-SecurityPolicy 1.2](#), [SAML 2.0](#), [Username Token Profile](#), [X.509 Token Profile](#), [SAML Token Profile](#), and [Kerberos Token Profile](#).

The WS-Trust extensions address the needs of applications that span multiple domains and requires the sharing of security keys by providing a standards based trusted third party web service (STS) to broker trust relationships between a Web service requester and a Web service provider. This architecture also alleviates the pain of service updates that require credential changes by providing a common location for this information. The STS is the common access point from which both the requester and provider retrieves and verifies security tokens.

There are three main components of the WS-Trust specification.

- The Security Token Service (STS), a web service that issues, renews, and validates security tokens.
- The message formats for security token requests and responses.
- The mechanisms for key exchange

Security Token Service

The Security Token Service, STS, is the core of the WS-Trust specification. It is a standards based mechanism for authentication and authorization. The STS is an implementation of the WS-Trust specification's protocol for issuing, exchanging, and validating security tokens, based on token format, namespace, or trust boundaries. The STS is a web service that acts as a trusted third party to broker trust relationships between a Web service requester and a Web service provider. It is a common access point trusted by both requester and provider to provide interoperable security tokens. It removes the need for a direct relationship between the two. Because the STS is a standards based mechanism for authentication, it helps ensure interoperability across realms and between different platforms.

The STS's WSDL contract defines how other applications and processes interact with it. In particular the WSDL defines the WS-Trust and WS-Security policies that a requester must fulfill in order to successfully communicate with the STS's endpoints. A web service requester consumes the STS's WSDL and with the aid of an STSClient utility, generates a message request compliant with the stated security policies and submits it to the STS endpoint. The STS validates the request and returns an appropriate response.



Apache CXF support

Apache CXF is an open-source, fully featured Web services framework. The JBossWS open source project integrates the JBoss Web Services (JBossWS) stack with the Apache CXF project modules thus providing WS-Trust and other JAX-WS functionality in WildFly. This integration makes it easy to deploy CXF STS implementations, however WildFly can run any WS-Trust compliant STS. In addition the Apache CXF API provides a STSClient utility to facilitate web service requester communication with its STS.

Detailed information about the Apache CXF's WS-Trust implementation can be found [here](#).

A Basic WS-Trust Scenario

Here is an example of a basic WS-Trust scenario. It is comprised of a Web service requester (ws-requester), a Web service provider (ws-provider), and a Security Token Service (STS). The ws-provider requires a SAML 2.0 token issued from a designed STS to be presented by the ws-requester using asymmetric binding. These communication requirements are declared in the ws-provider's WSDL. The STS requires ws-requester credentials be provided in a WSS UsernameToken format request using symmetric binding. The STS's response is provided containing a SAML 2.0 token. These communication requirements are declared in the STS's WSDL.

1. A ws-requester contacts the ws-provider and consumes its WSDL. Upon finding the security token issuer requirement, it creates and configures a STSClient with the information it requires to generate a proper request.
2. The STSClient contacts the STS and consumes its WSDL. The security policies are discovered. The STSClient creates and sends an authentication request, with appropriate credentials.
3. The STS verifies the credentials.
4. In response, the STS issues a security token that provides proof that the ws-requester has authenticated with the STS.
5. The STClient presents a message with the security token to the ws-provider.
6. The ws-provider verifies the token was issued by the STS, thus proving the ws-requester has successfully authenticated with the STS.
7. The ws-provider executes the requested service and returns the results to the the ws-requester.

Web service provider

This section examines the crucial elements in providing endpoint security in the web service provider described in the basic WS-Trust scenario. The components that will be discussed are.

- web service provider's WSDL
- web service provider's Interface and Implementation classes.
- ServerCallbackHandler class
- Crypto properties and keystore files
- MANIFEST.MF

Web service provider WSDL



The web service provider is a contract-first endpoint. All the WS-trust and security policies for it are declared in the WSDL, `SecurityService.wsdl`. For this scenario a ws-requester is required to present a SAML 2.0 token issued from a designed STS. The address of the STS is provided in the WSDL. An asymmetric binding policy is used to encrypt and sign the SOAP body of messages that pass back and forth between ws-requester and ws-provider. X.509 certificates are use for the asymmetric binding. The rules for sharing the public and private keys in the SOAP request and response messages are declared. A detailed explanation of the security settings are provided in the comments in the listing below.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<definitions targetNamespace="http://www.jboss.org/jbossws/ws-extensions/wssecuritypolicy"
name="SecurityService"
  xmlns:tns="http://www.jboss.org/jbossws/ws-extensions/wssecuritypolicy"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:wsp="http://www.w3.org/ns/ws-policy"
  xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata"

  xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
  xmlns:wsaws="http://www.w3.org/2005/08/addressing"
  xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702"
  xmlns:t="http://docs.oasis-open.org/ws-sx/ws-trust/200512">
  <types>
    <xsd:schema>
      <xsd:import namespace="http://www.jboss.org/jbossws/ws-extensions/wssecuritypolicy"
schemaLocation="SecurityService_schemal.xsd"/>
    </xsd:schema>
  </types>
  <message name="sayHello">
    <part name="parameters" element="tns:sayHello"/>
  </message>
  <message name="sayHelloResponse">
    <part name="parameters" element="tns:sayHelloResponse"/>
  </message>
  <portType name="ServiceIface">
    <operation name="sayHello">
      <input message="tns:sayHello"/>
      <output message="tns:sayHelloResponse"/>
    </operation>
  </portType>
  <!--
    The wsp:PolicyReference binds the security requirments on all the STS endpoints.
    The wsp:Policy wsu:Id="#AsymmetricSAML2Policy" element is defined later in this file.
  -->
  <binding name="SecurityServicePortBinding" type="tns:ServiceIface">
    <wsp:PolicyReference URI="#AsymmetricSAML2Policy" />
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
    <operation name="sayHello">
      <soap:operation soapAction="" />
      <input>
        <soap:body use="literal"/>
        <wsp:PolicyReference URI="#Input_Policy" />
      </input>
      <output>
        <soap:body use="literal"/>
      </output>
    </operation>
  </binding>
</definitions>
```




```
        <wsp:PolicyReference URI="#Output_Policy" />
    </output>
</operation>
</binding>
<service name="SecurityService">
    <port name="SecurityServicePort" binding="tns:SecurityServicePortBinding">
        <soap:address
location="http://@jboss.bind.address@:8080/jaxws-samples-wsse-policy-trust/SecurityService"/>
    </port>
</service>

<wsp:Policy wsu:Id="AsymmetricSAML2Policy">
    <wsp:ExactlyOne>
        <wsp:All>
<!--
    The wsam:Addressing element, indicates that the endpoints of this
    web service MUST conform to the WS-Addressing specification.  The
    attribute wsp:Optional="false" enforces this assertion.
-->
        <wsam:Addressing wsp:Optional="false">
            <wsp:Policy />
        </wsam:Addressing>
<!--
    The sp:AsymmetricBinding element indicates that security is provided
    at the SOAP layer.  A public/private key combinations is required to
    protect the message.  The initiator will use it's private key to sign
    the message and the recipient's public key is used to encrypt the message.
    The recipient of the message will use it's private key to decrypt it and
    initiator's public key to verify the signature.
-->
        <sp:AsymmetricBinding>
            <wsp:Policy>
<!--
    The sp:InitiatorToken element specifies the elements required in
    generating the initiator request to the ws-provider's service.
-->
            <sp:InitiatorToken>
                <wsp:Policy>
<!--
    The sp:IssuedToken element asserts that a SAML 2.0 security token is
    expected from the STS using a public key type.  The
    sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/IncludeToken/AlwaysToRe
attribute instructs the runtime to include the initiator's public key
with every message sent to the recipient.

    The sp:RequestSecurityTokenTemplate element directs that all of the
    children of this element will be copied directly into the body of the
    RequestSecurityToken (RST) message that is sent to the STS when the
    initiator asks the STS to issue a token.
-->
            <sp:IssuedToken

sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/IncludeToken/AlwaysToRe
<sp:RequestSecurityTokenTemplate>

<t:TokenType>http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.1#SAMLV2.0</t:TokenType>
<t:KeyType>http://docs.oasis-open.org/ws-sx/ws-trust/200512/PublicKey</t:KeyType>
```



```
        </sp:RequestSecurityTokenTemplate>
        <wsp:Policy>
            <sp:RequireInternalReference />
        </wsp:Policy>
<!--
    The sp:Issuer element defines the STS's address and endpoint information
    This information is used by the STSClient.
-->
        <sp:Issuer>
<wsaws:Address>http://@jboss.bind.address@:8080/jaxws-samples-wsse-policy-trust-sts/SecurityTokenS
<wsaws:Metadata xmlns:wsdli="http://www.w3.org/2006/01/wsdli-instance"

wsdli:wsdliLocation="http://@jboss.bind.address@:8080/jaxws-samples-wsse-policy-trust-sts/SecurityT
<wsaw:ServiceName xmlns:wsaw="http://www.w3.org/2006/05/addressing/wsdli"

xmlns:stsns="http://docs.oasis-open.org/ws-sx/ws-trust/200512/"

EndpointName="UT_Port">stsns:SecurityTokenService</wsaw:ServiceName>
        </wsaws:Metadata>
        </sp:Issuer>
        </sp:IssuedToken>
        </wsp:Policy>
    </sp:InitiatorToken>
<!--
    The sp:RecipientToken element asserts the type of public/private key-pair
    expected from the recipient.  The

sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/IncludeToken/Never">
    attribute indicates that the initiator's public key will never be included
    in the reply messages.

    The sp:WssX509V3Token10 element indicates that an X509 Version 3 token
    should be used in the message.
-->
        <sp:RecipientToken>
            <wsp:Policy>
                <sp:X509Token

sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/IncludeToken/Never">
                <wsp:Policy>
                    <sp:WssX509V3Token10 />
                    <sp:RequireIssuerSerialReference />
                </wsp:Policy>
            </sp:X509Token>
        </wsp:Policy>
    </sp:RecipientToken>
<!--
    The sp:Layout element, indicates the layout rules to apply when adding
    items to the security header.  The sp:Lax sub-element indicates items
    are added to the security header in any order that conforms to
    WSS: SOAP Message Security.
-->
        <sp:Layout>
            <wsp:Policy>
                <sp:Lax />
            </wsp:Policy>
        </sp:Layout>
```



```
        <sp:IncludeTimestamp />
        <sp:OnlySignEntireHeadersAndBody />
<!--
    The sp:AlgorithmSuite element, requires the Basic256 algorithm suite
    be used in performing cryptographic operations.
-->
        <sp:AlgorithmSuite>
            <wsp:Policy>
                <sp:Basic256 />
            </wsp:Policy>
        </sp:AlgorithmSuite>
    </wsp:Policy>
</sp:AsymmetricBinding>
<!--
    The sp:Wss11 element declares WSS: SOAP Message Security 1.1 options
    to be supported by the STS.  These particular elements generally refer
    to how keys are referenced within the SOAP envelope.  These are normally
    handled by CXF.
-->
        <sp:Wss11>
            <wsp:Policy>
                <sp:MustSupportRefIssuerSerial />
                <sp:MustSupportRefThumbprint />
                <sp:MustSupportRefEncryptedKey />
            </wsp:Policy>
        </sp:Wss11>
<!--
    The sp:Trust13 element declares controls for WS-Trust 1.3 options.
    They are policy assertions related to exchanges specifically with
    client and server challenges and entropy behaviors.  Again these are
    normally handled by CXF.
-->
        <sp:Trust13>
            <wsp:Policy>
                <sp:MustSupportIssuedTokens />
                <sp:RequireClientEntropy />
                <sp:RequireServerEntropy />
            </wsp:Policy>
        </sp:Trust13>
    </wsp:All>
</wsp:ExactlyOne>
</wsp:Policy>

<wsp:Policy wsu:Id="Input_Policy">
    <wsp:ExactlyOne>
        <wsp:All>
            <sp:EncryptedParts>
                <sp:Body />
            </sp:EncryptedParts>
            <sp:SignedParts>
                <sp:Body />
                <sp:Header Name="To" Namespace="http://www.w3.org/2005/08/addressing" />
                <sp:Header Name="From" Namespace="http://www.w3.org/2005/08/addressing" />
                <sp:Header Name="FaultTo" Namespace="http://www.w3.org/2005/08/addressing"
/>
                <sp:Header Name="ReplyTo" Namespace="http://www.w3.org/2005/08/addressing"
/>
                <sp:Header Name="MessageID" Namespace="http://www.w3.org/2005/08/addressing"
```



```
</>
    <sp:Header Name="RelatesTo" Namespace="http://www.w3.org/2005/08/addressing"
/>
    <sp:Header Name="Action" Namespace="http://www.w3.org/2005/08/addressing" />
    </sp:SignedParts>
  </wsp:All>
</wsp:ExactlyOne>
</wsp:Policy>

<wsp:Policy wsu:Id="Output_Policy">
  <wsp:ExactlyOne>
    <wsp:All>
      <sp:EncryptedParts>
        <sp:Body />
      </sp:EncryptedParts>
      <sp:SignedParts>
        <sp:Body />
        <sp:Header Name="To" Namespace="http://www.w3.org/2005/08/addressing" />
        <sp:Header Name="From" Namespace="http://www.w3.org/2005/08/addressing" />
        <sp:Header Name="FaultTo" Namespace="http://www.w3.org/2005/08/addressing"
/>
        <sp:Header Name="ReplyTo" Namespace="http://www.w3.org/2005/08/addressing"
/>
        <sp:Header Name="MessageID" Namespace="http://www.w3.org/2005/08/addressing"
/>
        <sp:Header Name="RelatesTo" Namespace="http://www.w3.org/2005/08/addressing"
/>
        <sp:Header Name="Action" Namespace="http://www.w3.org/2005/08/addressing" />
      </sp:SignedParts>
    </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>
</definitions>
```

Web service provider Interface

The web service provider interface class, `ServiceIface`, is a simple straight forward web service definition.

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.service;

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

@WebService
(
    targetNamespace = "http://www.jboss.org/jbossws/ws-extensions/wssecuritypolicy"
)
public interface ServiceIface
{
    @WebMethod
    String sayHello();
}
```

Web service provider Implementation



The web service provider implementation class, `ServiceImpl`, is a simple POJO. It uses the standard `WebService` annotation to define the service endpoint. In addition there are two Apache CXF annotations, `EndpointProperties` and `EndpointProperty` used for configuring the endpoint for the CXF runtime. These annotations come from the [Apache WSS4J project](#), which provides a Java implementation of the primary WS-Security standards for Web Services. These annotations are programmatically adding properties to the endpoint. With plain Apache CXF, these properties are often set via the `<jaxws:properties>` element on the `<jaxws:endpoint>` element in the Spring config; these annotations allow the properties to be configured in the code.

WSS4J uses the `Crypto` interface to get keys and certificates for encryption/decryption and for signature creation/verification. As is asserted by the WSDL, X509 keys and certificates are required for this service. The WSS4J configuration information being provided by `ServiceImpl` is for `Crypto`'s Merlin implementation. More information will be provided about this in the keystore section.

The first `EndpointProperty` statement in the listing is declaring the user's name to use for the message signature. It is used as the alias name in the keystore to get the user's cert and private key for signature. The next two `EndpointProperty` statements declares the Java properties file that contains the (Merlin) `crypto` configuration information. In this case both for signing and encrypting the messages. WSS4J reads this file and extra required information for message handling. The last `EndpointProperty` statement declares the `ServerCallbackHandler` implementation class. It is used to obtain the user's password for the certificates in the keystore file.



```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.service;

import javax.xml.ws.WebService;

import org.apache.cxf.annotations.EndpointProperties;
import org.apache.cxf.annotations.EndpointProperty;

@WebService
(
    portName = "SecurityServicePort",
    serviceName = "SecurityService",
    wsdlLocation = "WEB-INF/wsdl/SecurityService.wsdl",
    targetNamespace = "http://www.jboss.org/jbossws/ws-extensions/wssecuritypolicy",
    endpointInterface = "org.jboss.test.ws.jaxws.samples.wsse.policy.trust.service.ServiceIface"
)
@EndpointProperties(value = {
    @EndpointProperty(key = "ws-security.signature.username", value = "myservicekey"),
    @EndpointProperty(key = "ws-security.signature.properties", value =
"serviceKeystore.properties"),
    @EndpointProperty(key = "ws-security.encryption.properties", value =
"serviceKeystore.properties"),
    @EndpointProperty(key = "ws-security.callback-handler", value =
"org.jboss.test.ws.jaxws.samples.wsse.policy.trust.service.ServerCallbackHandler")
})
public class ServiceImpl implements ServiceIface
{
    public String sayHello()
    {
        return "WS-Trust Hello World!";
    }
}
```



ServerCallbackHandler

ServerCallbackHandler is a callback handler for the WSS4J Crypto API. It is used to obtain the password for the private key in the keystore. This class enables CXF to retrieve the password of the user name to use for the message signature. A certificates' password is not discoverable. The creator of the certificate must record the password he assigns and provide it when requested through the CallbackHandler. In this scenario skpass is the password for user myservicekey.

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.service;

import java.util.HashMap;
import java.util.Map;

import org.jboss.wsf.stack.cxf.extensions.security.PasswordCallbackHandler;

public class ServerCallbackHandler extends PasswordCallbackHandler
{

    public ServerCallbackHandler()
    {
        super(getInitMap());
    }

    private static Map<String, String> getInitMap()
    {
        Map<String, String> passwords = new HashMap<String, String>();
        passwords.put("myservicekey", "skpass");
        return passwords;
    }
}
```

Crypto properties and keystore files

WSS4J's Crypto implementation is loaded and configured via a Java properties file that contains Crypto configuration data. The file contains implementation-specific properties such as a keystore location, password, default alias and the like. This application is using the Merlin implementation. File serviceKeystore.properties contains this information.

File servicestore.jks, is a Java KeyStore (JKS) repository. It contains self signed certificates for myservicekey and mystskey. *Self signed certificates are not appropriate for production use.*

```
org.apache.ws.security.crypto.provider=org.apache.ws.security.components.crypto.Merlin
org.apache.ws.security.crypto.merlin.keystore.type=jks
org.apache.ws.security.crypto.merlin.keystore.password=sspass
org.apache.ws.security.crypto.merlin.keystore.alias=myservicekey
org.apache.ws.security.crypto.merlin.keystore.file=servicestore.jks
```



MANIFEST.MF

When deployed on WildFly this application requires access to the JBossWs and CXF APIs provided in module `org.jboss.ws.cxf.jbossws-cxf-client`. The dependency statement directs the server to provide them at deployment.

```
Manifest-Version: 1.0
Ant-Version: Apache Ant 1.8.2
Created-By: 1.7.0_25-b15 (Oracle Corporation)
Dependencies: org.jboss.ws.cxf.jbossws-cxf-client
```

Security Token Service (STS)

This section examines the crucial elements in providing the Security Token Service functionality described in the basic WS-Trust scenario. The components that will be discussed are.

- STS's WSDL
- STS's implementation class.
- STSCallbackHandler class
- Crypto properties and keystore files
- MANIFEST.MF
- Server configuration files

STS WSDL

The STS is a contract-first endpoint. All the WS-trust and security policies for it are declared in the WSDL, `ws-trust-1.4-service.wsdl`. A symmetric binding policy is used to encrypt and sign the SOAP body of messages that pass back and forth between `ws-requester` and the STS. The `ws-requester` is required to authenticate itself by providing WSS UsernameToken credentials. The rules for sharing the public and private keys in the SOAP request and response messages are declared. A detailed explanation of the security settings are provided in the comments in the listing below.

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions
  targetNamespace="http://docs.oasis-open.org/ws-sx/ws-trust/200512/"
  xmlns:tns="http://docs.oasis-open.org/ws-sx/ws-trust/200512/"
  xmlns:wstrust="http://docs.oasis-open.org/ws-sx/ws-trust/200512/"
  xmlns:wSDL="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:wsap10="http://www.w3.org/2006/05/addressing/wsdl"

  xmlns:wssu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
  xmlns:wsp="http://www.w3.org/ns/ws-policy"
  xmlns:wst="http://docs.oasis-open.org/ws-sx/ws-trust/200512"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata">

  <wsdl:types>
    <xs:schema elementFormDefault="qualified"
      targetNamespace='http://docs.oasis-open.org/ws-sx/ws-trust/200512'>
```




```
<xs:element name='RequestSecurityToken' type='wst:AbstractRequestSecurityTokenType' />
<xs:element name='RequestSecurityTokenResponse'
type='wst:AbstractRequestSecurityTokenType' />

<xs:complexType name='AbstractRequestSecurityTokenType' >
  <xs:sequence>
    <xs:any namespace='##any' processContents='lax' minOccurs='0' maxOccurs='unbounded' />
  </xs:sequence>
  <xs:attribute name='Context' type='xs:anyURI' use='optional' />
  <xs:anyAttribute namespace='##other' processContents='lax' />
</xs:complexType>
<xs:element name='RequestSecurityTokenCollection'
type='wst:RequestSecurityTokenCollectionType' />
<xs:complexType name='RequestSecurityTokenCollectionType' >
  <xs:sequence>
    <xs:element name='RequestSecurityToken' type='wst:AbstractRequestSecurityTokenType'
minOccurs='2' maxOccurs='unbounded' />
  </xs:sequence>
</xs:complexType>

<xs:element name='RequestSecurityTokenResponseCollection'
type='wst:RequestSecurityTokenResponseCollectionType' />
<xs:complexType name='RequestSecurityTokenResponseCollectionType' >
  <xs:sequence>
    <xs:element ref='wst:RequestSecurityTokenResponse' minOccurs='1' maxOccurs='unbounded'
/>
  </xs:sequence>
  <xs:anyAttribute namespace='##other' processContents='lax' />
</xs:complexType>

</xs:schema>
</wsdl:types>

<!-- WS-Trust defines the following GEDs -->
<wsdl:message name="RequestSecurityTokenMsg">
  <wsdl:part name="request" element="wst:RequestSecurityToken" />
</wsdl:message>
<wsdl:message name="RequestSecurityTokenResponseMsg">
  <wsdl:part name="response"
    element="wst:RequestSecurityTokenResponse" />
</wsdl:message>
<wsdl:message name="RequestSecurityTokenCollectionMsg">
  <wsdl:part name="requestCollection"
    element="wst:RequestSecurityTokenCollection" />
</wsdl:message>
<wsdl:message name="RequestSecurityTokenResponseCollectionMsg">
  <wsdl:part name="responseCollection"
    element="wst:RequestSecurityTokenResponseCollection" />
</wsdl:message>

<!-- This portType an example of a Requestor (or other) endpoint that
  Accepts SOAP-based challenges from a Security Token Service -->
<wsdl:portType name="WSSecurityRequestor">
  <wsdl:operation name="Challenge">
    <wsdl:input message="tns:RequestSecurityTokenResponseMsg" />
    <wsdl:output message="tns:RequestSecurityTokenResponseMsg" />
  </wsdl:operation>
</wsdl:portType>
```



```
<!-- This portType is an example of an STS supporting full protocol -->
<!--
  The wsdl:portType and data types are XML elements defined by the
  WS_Trust specification.  The wsdl:portType defines the endpoints
  supported in the STS implementation.  This WSDL defines all operations
  that an STS implementation can support.
-->
<wsdl:portType name="STS">
  <wsdl:operation name="Cancel">
    <wsdl:input wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Cancel"
message="tns:RequestSecurityTokenMsg"/>
    <wsdl:output
wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/CancelFinal"
message="tns:RequestSecurityTokenResponseMsg"/>
  </wsdl:operation>
  <wsdl:operation name="Issue">
    <wsdl:input wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Issue"
message="tns:RequestSecurityTokenMsg"/>
    <wsdl:output
wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTRC/IssueFinal"
message="tns:RequestSecurityTokenResponseCollectionMsg"/>
  </wsdl:operation>
  <wsdl:operation name="Renew">
    <wsdl:input wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Renew"
message="tns:RequestSecurityTokenMsg"/>
    <wsdl:output
wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/RenewFinal"
message="tns:RequestSecurityTokenResponseMsg"/>
  </wsdl:operation>
  <wsdl:operation name="Validate">
    <wsdl:input wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Validate"
message="tns:RequestSecurityTokenMsg"/>
    <wsdl:output
wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/ValidateFinal"
message="tns:RequestSecurityTokenResponseMsg"/>
  </wsdl:operation>
  <wsdl:operation name="KeyExchangeToken">
    <wsdl:input wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/KET"
message="tns:RequestSecurityTokenMsg"/>
    <wsdl:output wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/KETFinal"
message="tns:RequestSecurityTokenResponseMsg"/>
  </wsdl:operation>
  <wsdl:operation name="RequestCollection">
    <wsdl:input message="tns:RequestSecurityTokenCollectionMsg"/>
    <wsdl:output message="tns:RequestSecurityTokenResponseCollectionMsg"/>
  </wsdl:operation>
</wsdl:portType>

<!-- This portType is an example of an endpoint that accepts
  Unsolicited RequestSecurityTokenResponse messages -->
<wsdl:portType name="SecurityTokenResponseService">
  <wsdl:operation name="RequestSecurityTokenResponse">
    <wsdl:input message="tns:RequestSecurityTokenResponseMsg"/>
  </wsdl:operation>
</wsdl:portType>
```



```
<!--
  The wsp:PolicyReference binds the security requirements on all the STS endpoints.
  The wsp:Policy wsu:Id="UT_policy" element is later in this file.
-->
<wsdl:binding name="UT_Binding" type="wstrust:STS">
  <wsp:PolicyReference URI="#UT_policy" />
  <soap:binding style="document"
    transport="http://schemas.xmlsoap.org/soap/http" />
  <wsdl:operation name="Issue">
    <soap:operation
      soapAction="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Issue" />
    <wsdl:input>
      <wsp:PolicyReference
        URI="#Input_policy" />
      <soap:body use="literal" />
    </wsdl:input>
    <wsdl:output>
      <wsp:PolicyReference
        URI="#Output_policy" />
      <soap:body use="literal" />
    </wsdl:output>
  </wsdl:operation>
  <wsdl:operation name="Validate">
    <soap:operation
      soapAction="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Validate" />
    <wsdl:input>
      <wsp:PolicyReference
        URI="#Input_policy" />
      <soap:body use="literal" />
    </wsdl:input>
    <wsdl:output>
      <wsp:PolicyReference
        URI="#Output_policy" />
      <soap:body use="literal" />
    </wsdl:output>
  </wsdl:operation>
  <wsdl:operation name="Cancel">
    <soap:operation
      soapAction="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Cancel" />
    <wsdl:input>
      <soap:body use="literal" />
    </wsdl:input>
    <wsdl:output>
      <soap:body use="literal" />
    </wsdl:output>
  </wsdl:operation>
  <wsdl:operation name="Renew">
    <soap:operation
      soapAction="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Renew" />
    <wsdl:input>
      <soap:body use="literal" />
    </wsdl:input>
    <wsdl:output>
      <soap:body use="literal" />
    </wsdl:output>
  </wsdl:operation>
  <wsdl:operation name="KeyExchangeToken">
    <soap:operation
```



```
        soapAction="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/KeyExchangeToken"
    />
    <wsdl:input>
        <soap:body use="literal" />
    </wsdl:input>
    <wsdl:output>
        <soap:body use="literal" />
    </wsdl:output>
</wsdl:operation>
<wsdl:operation name="RequestCollection">
    <soap:operation

soapAction="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/RequestCollection" />
    <wsdl:input>
        <soap:body use="literal" />
    </wsdl:input>
    <wsdl:output>
        <soap:body use="literal" />
    </wsdl:output>
</wsdl:operation>
</wsdl:binding>

<wsdl:service name="SecurityTokenService">
    <wsdl:port name="UT_Port" binding="tns:UT_Binding">
        <soap:address location="http://localhost:8080/SecurityTokenService/UT" />
    </wsdl:port>
</wsdl:service>

<wsp:Policy wsu:Id="UT_policy">
    <wsp:ExactlyOne>
        <wsp:All>
<!--
    The sp:UsingAddressing element, indicates that the endpoints of this
    web service conforms to the WS-Addressing specification. More detail
    can be found here: [http://www.w3.org/TR/2006/CR-ws-addr-wsdl-20060529]
-->
        <wsap10:UsingAddressing/>
<!--
    The sp:SymmetricBinding element indicates that security is provided
    at the SOAP layer and any initiator must authenticate itself by providing
    WSS UsernameToken credentials.
-->
        <sp:SymmetricBinding
            xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
            <wsp:Policy>
<!--
    In a symmetric binding, the keys used for encrypting and signing in both
    directions are derived from a single key, the one specified by the
    sp:ProtectionToken element. The sp:X509Token sub-element declares this
    key to be a X.509 certificate and the
    IncludeToken="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/IncludeToken/Never"
    attribute adds the requirement that the token MUST NOT be included in
    any messages sent between the initiator and the recipient; rather, an
    external reference to the token should be used. Lastly the WssX509V3Token10
    sub-element declares that the Username token presented by the initiator
    should be compliant with Web Services Security UsernameToken Profile
    1.0 specification. [
    http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0.pdf ]
```



```
-->
    <sp:ProtectionToken>
        <wsp:Policy>
            <sp:X509Token

sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/IncludeToken/Never">
            <wsp:Policy>
                <sp:RequireDerivedKeys />
                <sp:RequireThumbprintReference />
                <sp:WssX509V3Token10 />
            </wsp:Policy>
        </sp:X509Token>
    </wsp:Policy>
</sp:ProtectionToken>
<!--
    The sp:AlgorithmSuite element, requires the Basic256 algorithm suite
    be used in performing cryptographic operations.
-->
    <sp:AlgorithmSuite>
        <wsp:Policy>
            <sp:Basic256 />
        </wsp:Policy>
    </sp:AlgorithmSuite>
<!--
    The sp:Layout element, indicates the layout rules to apply when adding
    items to the security header. The sp:Lax sub-element indicates items
    are added to the security header in any order that conforms to
    WSS: SOAP Message Security.
-->
    <sp:Layout>
        <wsp:Policy>
            <sp:Lax />
        </wsp:Policy>
    </sp:Layout>
    <sp:IncludeTimestamp />
    <sp:EncryptSignature />
    <sp:OnlySignEntireHeadersAndBody />
</wsp:Policy>
</sp:SymmetricBinding>
<!--
    The sp:SignedSupportingTokens element declares that the security header
    of messages must contain a sp:UsernameToken and the token must be signed.
    The attribute
    IncludeToken="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/IncludeToken/AlwaysToRecip
on sp:UsernameToken indicates that the token MUST be included in all
    messages sent from initiator to the recipient and that the token MUST
    NOT be included in messages sent from the recipient to the initiator.
    And finally the element sp:WssUsernameToken10 is a policy assertion
    indicating the Username token should be as defined in Web Services
    Security UsernameToken Profile 1.0
-->
    <sp:SignedSupportingTokens
        xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
        <wsp:Policy>
            <sp:UsernameToken

sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/IncludeToken/AlwaysToRe
<wsp:Policy>
```



```
        <sp:WssUsernameToken10 />
        </wsp:Policy>
    </sp:UsernameToken>
</wsp:Policy>
</sp:SignedSupportingTokens>
<!--
The sp:Wss11 element declares WSS: SOAP Message Security 1.1 options
to be supported by the STS.  These particular elements generally refer
to how keys are referenced within the SOAP envelope.  These are normally
handled by CXF.
-->
    <sp:Wss11
        xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
        <wsp:Policy>
            <sp:MustSupportRefKeyIdentifier />
            <sp:MustSupportRefIssuerSerial />
            <sp:MustSupportRefThumbprint />
            <sp:MustSupportRefEncryptedKey />
        </wsp:Policy>
    </sp:Wss11>
<!--
The sp:Trust13 element declares controls for WS-Trust 1.3 options.
They are policy assertions related to exchanges specifically with
client and server challenges and entropy behaviors.  Again these are
normally handled by CXF.
-->
    <sp:Trust13
        xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
        <wsp:Policy>
            <sp:MustSupportIssuedTokens />
            <sp:RequireClientEntropy />
            <sp:RequireServerEntropy />
        </wsp:Policy>
    </sp:Trust13>
</wsp:All>
</wsp:ExactlyOne>
</wsp:Policy>

<wsp:Policy wsu:Id="Input_policy">
    <wsp:ExactlyOne>
        <wsp:All>
            <sp:SignedParts
                xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
                <sp:Body />
                <sp:Header Name="To"
                    Namespace="http://www.w3.org/2005/08/addressing" />
                <sp:Header Name="From"
                    Namespace="http://www.w3.org/2005/08/addressing" />
                <sp:Header Name="FaultTo"
                    Namespace="http://www.w3.org/2005/08/addressing" />
                <sp:Header Name="ReplyTo"
                    Namespace="http://www.w3.org/2005/08/addressing" />
                <sp:Header Name="MessageID"
                    Namespace="http://www.w3.org/2005/08/addressing" />
                <sp:Header Name="RelatesTo"
                    Namespace="http://www.w3.org/2005/08/addressing" />
                <sp:Header Name="Action"
                    Namespace="http://www.w3.org/2005/08/addressing" />
```



```
</sp:SignedParts>
<sp:EncryptedParts
  xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
  <sp:Body />
</sp:EncryptedParts>
</wsp:All>
</wsp:ExactlyOne>
</wsp:Policy>

<wsp:Policy wsu:Id="Output_policy">
  <wsp:ExactlyOne>
    <wsp:All>
      <sp:SignedParts
        xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
          <sp:Body />
          <sp:Header Name="To"
            Namespace="http://www.w3.org/2005/08/addressing" />
          <sp:Header Name="From"
            Namespace="http://www.w3.org/2005/08/addressing" />
          <sp:Header Name="FaultTo"
            Namespace="http://www.w3.org/2005/08/addressing" />
          <sp:Header Name="ReplyTo"
            Namespace="http://www.w3.org/2005/08/addressing" />
          <sp:Header Name="MessageID"
            Namespace="http://www.w3.org/2005/08/addressing" />
          <sp:Header Name="RelatesTo"
            Namespace="http://www.w3.org/2005/08/addressing" />
          <sp:Header Name="Action"
            Namespace="http://www.w3.org/2005/08/addressing" />
        </sp:SignedParts>
        <sp:EncryptedParts
          xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
          <sp:Body />
        </sp:EncryptedParts>
      </wsp:All>
    </wsp:ExactlyOne>
  </wsp:Policy>

</wsdl:definitions>
```

STS Implementation

The Apache CXF's STS, `SecurityTokenServiceProvider`, is a web service provider that is compliant with the protocols and functionality defined by the WS-Trust specification. It has a modular architecture. Many of its components are configurable or replaceable and there are many optional features that are enabled by implementing and configuring plug-ins. Users can customize their own STS by extending from `SecurityTokenServiceProvider` and overriding the default settings. Extensive information about the CXF's STS configurable and pluggable components can be found [here](#).



This STS implementation class, `SimpleSTS`, is a POJO that extends from `SecurityTokenServiceProvider`. Note that the class is defined with a `WebServiceProvider` annotation and not a `WebService` annotation. This annotation defines the service as a Provider-based endpoint, meaning it supports a more messaging-oriented approach to Web services. In particular, it signals that the exchanged messages will be XML documents of some type. `SecurityTokenServiceProvider` is an implementation of the `javax.xml.ws.Provider` interface. In comparison the `WebService` annotation defines a (service endpoint interface) SEI-based endpoint which supports message exchange via SOAP envelopes.

As was done in the `ServiceImpl` class, the WSS4J annotations `EndpointProperties` and `EndpointProperty` are providing endpoint configuration for the CXF runtime. This was previous described [here](#).

The `InInterceptors` annotation is used to specify a JBossWS integration interceptor to be used for authenticating incoming requests; JAAS integration is used here for authentication, the username/password coming from the `UsernameToken` in the `ws-requester` message are used for authenticating the requester against a security domain on the application server hosting the STS deployment.

In this implementation we are customizing the operations of token issuance, token validation and their static properties.

`StaticSTSProperties` is used to set select properties for configuring resources in the STS. You may think this is a duplication of the settings made with the WSS4J annotations. The values are the same but the underlying structures being set are different, thus this information must be declared in both places.

The `setIssuer` setting is important because it uniquely identifies the issuing STS. The issuer string is embedded in issued tokens and, when validating tokens, the STS checks the issuer string value. Consequently, it is important to use the issuer string in a consistent way, so that the STS can recognize the tokens that it has issued.

The `setEndpoints` call allows the declaration of a set of allowed token recipients by address. The addresses are specified as reg-ex patterns.

`TokenIssueOperation` and `TokenValidateOperation` have a modular structure. This allows custom behaviors to be injected into the processing of messages. In this case we are overriding the `SecurityTokenServiceProvider`'s default behavior and performing SAML token processing and validation. CXF provides an implementation of a `SAMLTokenProvider` and `SAMLTokenValidator` which we are using rather than writing our own.

Learn more about the `SAMLTokenProvider` [here](#).

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust;

import java.util.Arrays;
import java.util.LinkedList;
import java.util.List;

import javax.xml.ws.WebServiceProvider;

import org.apache.cxf.annotations.EndpointProperties;
import org.apache.cxf.annotations.EndpointProperty;
import org.apache.cxf.interceptor.InInterceptors;
import org.apache.cxf.sts.StaticSTSProperties;
```




```
import org.apache.cxf.sts.operation.TokenIssueOperation;
import org.apache.cxf.sts.operation.TokenValidateOperation;
import org.apache.cxf.sts.service.ServiceMBean;
import org.apache.cxf.sts.service.StaticService;
import org.apache.cxf.sts.token.provider.SAMLTokenProvider;
import org.apache.cxf.sts.token.validator.SAMLTokenValidator;
import org.apache.cxf.ws.security.sts.provider.SecurityTokenServiceProvider;

@WebServiceProvider(serviceName = "SecurityTokenService",
    portName = "UT_Port",
    targetNamespace = "http://docs.oasis-open.org/ws-sx/ws-trust/200512/",
    wsdlLocation = "WEB-INF/wsdl/ws-trust-1.4-service.wsdl")
@EndpointProperties(value = {
    @EndpointProperty(key = "ws-security.signature.username", value = "mystskey"),
    @EndpointProperty(key = "ws-security.signature.properties", value =
"stsKeystore.properties"),
    @EndpointProperty(key = "ws-security.callback-handler", value =
"org.jboss.test.ws.jaxws.samples.wsse.policy.trust.STSCallbackHandler"),
    //to let the JAAS integration deal with validation through the interceptor below
    @EndpointProperty(key = "ws-security.validate.token", value = "false")
})
@InInterceptors(interceptors =
{"org.jboss.wsf.stack.cxf.security.authentication.SubjectCreatingPolicyInterceptor"})
public class SampleSTS extends SecurityTokenServiceProvider
{
    public SampleSTS() throws Exception
    {
        super();

        StaticSTSProperties props = new StaticSTSProperties();
        props.setSignaturePropertiesFile("stsKeystore.properties");
        props.setSignatureUsername("mystskey");
        props.setCallbackHandlerClass(STSCallbackHandler.class.getName());
        props.setIssuer("DoubleItSTSIssuer");

        List<ServiceMBean> services = new LinkedList<ServiceMBean>();
        StaticService service = new StaticService();
        service.setEndpoints(Arrays.asList(
            "http://localhost:(\\d)*/jaxws-samples-wsse-policy-trust/SecurityService",
            "http://\\[::1\\]:(\\d)*/jaxws-samples-wsse-policy-trust/SecurityService",
            "http://\\[[0:0:0:0:0:0:1\\]:(\\d)*/jaxws-samples-wsse-policy-trust/SecurityService"
        ));
        services.add(service);

        TokenIssueOperation issueOperation = new TokenIssueOperation();
        issueOperation.setServices(services);
        issueOperation.getTokenProviders().add(new SAMLTokenProvider());
        issueOperation.setStsProperties(props);

        TokenValidateOperation validateOperation = new TokenValidateOperation();
        validateOperation.getTokenValidators().add(new SAMLTokenValidator());
        validateOperation.setStsProperties(props);

        this.setIssueOperation(issueOperation);
        this.setValidateOperation(validateOperation);
    }
}
```



```
}
```

STSCallbackHandler

STSCallbackHandler is a callback handler for the WSS4J Crypto API. It is used to obtain the password for the private key in the keystore. This class enables CXF to retrieve the password of the user name to use for the message signature.

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.sts;

import java.util.HashMap;
import java.util.Map;

import org.jboss.wsf.stack.cxf.extensions.security.PasswordCallbackHandler;

public class STSCallbackHandler extends PasswordCallbackHandler
{
    public STSCallbackHandler()
    {
        super(getInitMap());
    }

    private static Map<String, String> getInitMap()
    {
        Map<String, String> passwords = new HashMap<String, String>();
        passwords.put("mystskey", "stskpass");
        return passwords;
    }
}
```

Crypto properties and keystore files

WSS4J's Crypto implementation is loaded and configured via a Java properties file that contains Crypto configuration data. The file contains implementation-specific properties such as a keystore location, password, default alias and the like. This application is using the Merlin implementation. File `stsKeystore.properties` contains this information.

File `servicestore.jks`, is a Java KeyStore (JKS) repository. It contains self signed certificates for `myservicekey` and `mystskey`. *Self signed certificates are not appropriate for production use.*

```
org.apache.ws.security.crypto.provider=org.apache.wss4j.common.crypto.Merlin
org.apache.ws.security.crypto.merlin.keystore.type=jks
org.apache.ws.security.crypto.merlin.keystore.password=stsspass
org.apache.ws.security.crypto.merlin.keystore.file=stsstore.jks
```



MANIFEST.MF

When deployed on WildFly, this application requires access to the JBossWs and CXF APIs provided in modules `org.jboss.ws.cxf.jbossws-cxf-client` and `org.apache.cxf`. The Apache CXF internals, `org.apache.cxf.impl`, are needed to build the STS configuration in the `SampleSTS` constructor. The dependency statement directs the server to provide them at deployment.

```
Manifest-Version: 1.0
Ant-Version: Apache Ant 1.8.2
Created-By: 1.7.0_25-b15 (Oracle Corporation)
Dependencies: org.jboss.ws.cxf.jbossws-cxf-client,org.apache.cxf.impl
```

Security Domain

The STS requires a JBoss security domain be configured. The `jboss-web.xml` descriptor declares a named security domain, "JBossWS-trust-sts" to be used by this service for authentication. This security domain requires two properties files and the addition of a security-domain declaration in the JBoss server configuration file.

For this scenario the domain needs to contain user *alice*, password *clarinet*, and role *friend*. See the listings below for `jbossws-users.properties` and `jbossws-roles.properties`. In addition the following XML must be added to the JBoss security subsystem in the server configuration file. Replace "**SOME_PATH**" with appropriate information.

```
<security-domain name="JBossWS-trust-sts">
  <authentication>
    <login-module code="UsersRoles" flag="required">
      <module-option name="usersProperties" value="/SOME_PATH/jbossws-users.properties"/>
      <module-option name="unauthenticatedIdentity" value="anonymous"/>
      <module-option name="rolesProperties" value="/SOME_PATH/jbossws-roles.properties"/>
    </login-module>
  </authentication>
</security-domain>
```

jboss-web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jboss-web PUBLIC "-//JBoss//DTD Web Application 2.4//EN" "">
<jboss-web>
  <security-domain>java:/jaas/JBossWS-trust-sts</security-domain>
</jboss-web>
```

jbossws-users.properties

```
# A sample users.properties file for use with the UsersRolesLoginModule
alice=clarinet
```

jbossws-roles.properties



```
# A sample roles.properties file for use with the UsersRolesLoginModule
alice=friend
```

✔ WS-MetadataExchange and interoperability

To achieve better interoperability, you might consider allowing the STS endpoint to reply to WS-MetadataExchange messages directed to the `/mex` URL sub-path (e.g.

<http://localhost:8080/jaxws-samples-wsse-policy-trust-sts/SecurityTokenService/mex>). This can be done by tweaking the *url-pattern* for the underlying endpoint servlet, for instance by adding a *web.xml* descriptor as follows to the deployment: `<?xml version="1.0" encoding="UTF-8"?>`

```
<web-app
version="2.5" 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
http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd">
<servlet>
<servlet-name>TestSecurityTokenService</servlet-name>
<servlet-class>org.jboss.test.ws.jaxws.samples.wsse.policy.trust.SampleSTS</servlet-class>
</servlet>
<servlet-mapping>
<servlet-name>TestSecurityTokenService</servlet-name>
<url-pattern>/SecurityTokenService/*</url-pattern>
</servlet-mapping>
</web-app>
```

As a matter of fact, at the time of writing some webservices implementations (including *Metro*) assume the `/mex` URL as the default choice for directing WS-MetadataExchange requests to and use that to retrieve STS wsdl contracts.

Web service requester

This section examines the crucial elements in calling a web service that implements endpoint security as described in the basic WS-Trust scenario. The components that will be discussed are.

- web service requester's implementation
- ClientCallbackHandler
- Crypto properties and keystore files

Web service requester Implementation



The ws-requester, the client, uses standard procedures for creating a reference to the web service in the first four lines. To address the endpoint security requirements, the web service's "Request Context" is configured with the information needed in message generation. In addition, the STSClient that communicates with the STS is configured with similar values. Note the key strings ending with a ".it" suffix. This suffix flags these settings as belonging to the STSClient. The internal CXF code assigns this information to the STSClient that is auto-generated for this service call.

There is an alternate method of setting up the STSClient. The user may provide their own instance of the STSClient. The CXF code will use this object and not auto-generate one. This is used in the ActAs and OnBehalfOf examples. When providing the STSClient in this way, the user must provide a org.apache.cxf.Bus for it and the configuration keys must not have the ".it" suffix.

```
QName serviceName = new QName("http://www.jboss.org/jbossws/ws-extensions/wssecuritypolicy",
    "SecurityService");
URL wsdlURL = new URL(serviceURL + "?wsdl");
Service service = Service.create(wsdlURL, serviceName);
ServiceIface proxy = (ServiceIface) service.getPort(ServiceIface.class);

// set the security related configuration information for the service "request"
Map<String, Object> ctx = ((BindingProvider) proxy).getRequestContext();
ctx.put(SecurityConstants.CALLBACK_HANDLER, new ClientCallbackHandler());
ctx.put(SecurityConstants.SIGNATURE_PROPERTIES,
    Thread.currentThread().getContextClassLoader().getResource(
        "META-INF/clientKeystore.properties"));
ctx.put(SecurityConstants.ENCRYPT_PROPERTIES,
    Thread.currentThread().getContextClassLoader().getResource(
        "META-INF/clientKeystore.properties"));
ctx.put(SecurityConstants.SIGNATURE_USERNAME, "myclientkey");
ctx.put(SecurityConstants.ENCRYPT_USERNAME, "myservicekey");

/-- Configuration settings that will be transfered to the STSClient
// "alice" is the name provided for the WSS Username. Her password will
// be retrieved from the ClientCallbackHander by the STSClient.
ctx.put(SecurityConstants.USERNAME + ".it", "alice");
ctx.put(SecurityConstants.CALLBACK_HANDLER + ".it", new ClientCallbackHandler());
ctx.put(SecurityConstants.ENCRYPT_PROPERTIES + ".it",
    Thread.currentThread().getContextClassLoader().getResource(
        "META-INF/clientKeystore.properties"));
ctx.put(SecurityConstants.ENCRYPT_USERNAME + ".it", "mystskey");
// alias name in the keystore to get the user's public key to send to the STS
ctx.put(SecurityConstants.STS_TOKEN_USERNAME + ".it", "myclientkey");
// Crypto property configuration to use for the STS
ctx.put(SecurityConstants.STS_TOKEN_PROPERTIES + ".it",
    Thread.currentThread().getContextClassLoader().getResource(
        "META-INF/clientKeystore.properties"));
// write out an X509Certificate structure in UseKey/KeyInfo
ctx.put(SecurityConstants.STS_TOKEN_USE_CERT_FOR_KEYINFO + ".it", "true");
// Setting indicates the STSclient should not try using the WS-MetadataExchange
// call using STS EPR WSA address when the endpoint contract does not contain
// WS-MetadataExchange info.
ctx.put("ws-security.sts.disable-wsmex-call-using-epr-address", "true");

proxy.sayHello();
```



ClientCallbackHandler

ClientCallbackHandler is a callback handler for the WSS4J Crypto API. It is used to obtain the password for the private key in the keystore. This class enables CXF to retrieve the password of the user name to use for the message signature. Note that "alice" and her password have been provided here. This information is not in the (JKS) keystore but provided in the WildFly security domain. It was declared in file `jbossws-users.properties`.

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.shared;

import java.io.IOException;
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.UnsupportedCallbackException;
import org.apache.ws.security.WSPasswordCallback;

public class ClientCallbackHandler implements CallbackHandler {

    public void handle(Callback[] callbacks) throws IOException,
        UnsupportedCallbackException {
        for (int i = 0; i < callbacks.length; i++) {
            if (callbacks[i] instanceof WSPasswordCallback) {
                WSPasswordCallback pc = (WSPasswordCallback) callbacks[i];
                if ("myclientkey".equals(pc.getIdentifier())) {
                    pc.setPassword("ckpass");
                    break;
                } else if ("alice".equals(pc.getIdentifier())) {
                    pc.setPassword("clarinet");
                    break;
                }
            }
        }
    }
}
```

Requester Crypto properties and keystore files

WSS4J's Crypto implementation is loaded and configured via a Java properties file that contains Crypto configuration data. The file contains implementation-specific properties such as a keystore location, password, default alias and the like. This application is using the Merlin implementation. File `clientKeystore.properties` contains this information.

File `clientstore.jks`, is a Java KeyStore (JKS) repository. It contains self signed certificates for `myservicekey` and `mystskey`. *Self signed certificates are not appropriate for production use.*

```
org.apache.ws.security.crypto.provider=org.apache.ws.security.components.crypto.Merlin
org.apache.ws.security.crypto.merlin.keystore.type=jks
org.apache.ws.security.crypto.merlin.keystore.password=cspass
org.apache.ws.security.crypto.merlin.keystore.alias=myclientkey
org.apache.ws.security.crypto.merlin.keystore.file=META-INF/clientstore.jks
```



PicketLink STS

[PicketLink](#) provides facilities for building up an alternative to the Apache CXF Security Token Service implementation.

Similarly to the previous implementation, the STS is served through a `WebServiceProvider` annotated POJO:

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust;

import javax.annotation.Resource;
import javax.xml.ws.Service;
import javax.xml.ws.ServiceMode;
import javax.xml.ws.WebServiceContext;
import javax.xml.ws.WebServiceProvider;

import org.apache.cxf.annotations.EndpointProperties;
import org.apache.cxf.annotations.EndpointProperty;
import org.apache.cxf.interceptor.InInterceptors;
import org.picketlink.identity.federation.core.wstrust.PicketLinkSTS;

@WebServiceProvider(serviceName = "PicketLinkSTS", portName = "PicketLinkSTSPort",
targetNamespace = "urn:picketlink:identity-federation:sts", wsdlLocation =
"WEB-INF/wsdl/PicketLinkSTS.wsdl")
@ServiceMode(value = Service.Mode.MESSAGE)
//be sure to have dependency on org.apache.cxf module when on AS7, otherwise Apache CXF
annotations are ignored
@EndpointProperties(value = {
@EndpointProperty(key = "ws-security.signature.username", value = "mystskey"),
@EndpointProperty(key = "ws-security.signature.properties", value = "stsKeystore.properties"),
@EndpointProperty(key = "ws-security.callback-handler", value =
"org.jboss.test.ws.jaxws.samples.wsse.policy.trust.STSCallbackHandler"),
@EndpointProperty(key = "ws-security.validate.token", value = "false") //to let the JAAS
integration deal with validation through the interceptor below
})
@InInterceptors(interceptors =

)
public class PicketLinkSTService extends PicketLinkSTS {
@Resource
public void setWSC(WebServiceContext wctx)
Unknown macro: { this.context = wctx; }

}
```

The `@WebServiceProvider` annotation references the following WS-Policy enabled wsdl contract; please note the wsdl operations, messages and such must match the `PicketLinkSTS` implementation:

```
<?xml version="1.0"?>
<wsdl:definitions name="PicketLinkSTS" targetNamespace="urn:picketlink:identity-federation:sts"
xmlns:tns="urn:picketlink:identity-federation:sts"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:wsap10="http://www.w3.org/2006/05/addressing/wsdl"
xmlns:wsp="http://www.w3.org/ns/ws-policy">
```



```
xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
  xmlns:wst="http://docs.oasis-open.org/ws-sx/ws-trust/200512"
  xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/">
  <wsdl:types>
    <xs:schema elementFormDefault="qualified"
targetNamespace='http://docs.oasis-open.org/ws-sx/ws-trust/200512'
xmlns:xs="http://www.w3.org/2001/XMLSchema">
      <xs:element name='RequestSecurityToken' type='wst:AbstractRequestSecurityTokenType' />
      <xs:element name='RequestSecurityTokenResponse'
type='wst:AbstractRequestSecurityTokenType' />
      <xs:complexType name='AbstractRequestSecurityTokenType' >
        <xs:sequence>
          <xs:any namespace='##any' processContents='lax' minOccurs='0' maxOccurs='unbounded' />
        </xs:sequence>
        <xs:attribute name='Context' type='xs:anyURI' use='optional' />
        <xs:anyAttribute namespace='##other' processContents='lax' />
      </xs:complexType>
      <xs:element name='RequestSecurityTokenCollection'
type='wst:RequestSecurityTokenCollectionType' />
      <xs:complexType name='RequestSecurityTokenCollectionType' >
        <xs:sequence>
          <xs:element name='RequestSecurityToken' type='wst:AbstractRequestSecurityTokenType'
minOccurs='2' maxOccurs='unbounded' />
        </xs:sequence>
      </xs:complexType>
      <xs:element name='RequestSecurityTokenResponseCollection'
type='wst:RequestSecurityTokenResponseCollectionType' />
      <xs:complexType name='RequestSecurityTokenResponseCollectionType' >
        <xs:sequence>
          <xs:element ref='wst:RequestSecurityTokenResponse' minOccurs='1' maxOccurs='unbounded'
/ >
        </xs:sequence>
      </xs:complexType>
    </xs:schema>
  </wsdl:types>

  <wsdl:message name="RequestSecurityTokenMsg">
    <wsdl:part name="request" element="wst:RequestSecurityToken" />
  </wsdl:message>
  <wsdl:message name="RequestSecurityTokenResponseCollectionMsg">
    <wsdl:part name="responseCollection"
      element="wst:RequestSecurityTokenResponseCollection"/>
  </wsdl:message>

  <wsdl:portType name="SecureTokenService">
    <wsdl:operation name="IssueToken">
      <wsdl:input wsap10:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Issue"
message="tns:RequestSecurityTokenMsg" />
      <wsdl:output
wsap10:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTRC/IssueFinal"
message="tns:RequestSecurityTokenResponseCollectionMsg" />
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="STSBinding" type="tns:SecureTokenService">
    <wsp:PolicyReference URI="#UT_policy" />
    <soap12:binding transport="http://schemas.xmlsoap.org/soap/http"/>
```




```
<wsdl:operation name="IssueToken">
  <soap12:operation soapAction="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Issue"
style="document"/>
  <wsdl:input>
    <wsp:PolicyReference URI="#Input_policy" />
    <soap12:body use="literal"/>
  </wsdl:input>
  <wsdl:output>
    <wsp:PolicyReference URI="#Output_policy" />
    <soap12:body use="literal"/>
  </wsdl:output>
</wsdl:operation>
</wsdl:binding>
<wsdl:service name="PicketLinkSTS">
  <wsdl:port name="PicketLinkSTSPort" binding="tns:STSBinding">
    <soap12:address location="http://localhost:8080/picketlink-sts/PicketLinkSTS"/>
  </wsdl:port>
</wsdl:service>

<wsp:Policy wsu:Id="UT_policy">
  <wsp:ExactlyOne>
    <wsp:All>
      <wsap10:UsingAddressing/>
      <sp:SymmetricBinding
        xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
        <wsp:Policy>
          <sp:ProtectionToken>
            <wsp:Policy>
              <sp:X509Token
sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/IncludeToken/Never">
                <wsp:Policy>
                  <sp:RequireDerivedKeys />
                  <sp:RequireThumbprintReference />
                  <sp:WssX509V3Token10 />
                </wsp:Policy>
              </sp:X509Token>
            </wsp:Policy>
          </sp:ProtectionToken>
          <sp:AlgorithmSuite>
            <wsp:Policy>
              <sp:Basic256 />
            </wsp:Policy>
          </sp:AlgorithmSuite>
          <sp:Layout>
            <wsp:Policy>
              <sp:Lax />
            </wsp:Policy>
          </sp:Layout>
          <sp:IncludeTimestamp />
          <sp:EncryptSignature />
          <sp:OnlySignEntireHeadersAndBody />
        </wsp:Policy>
      </sp:SymmetricBinding>
      <sp:SignedSupportingTokens
        xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
        <wsp:Policy>
          <sp:UsernameToken
```



```
sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/IncludeToken/AlwaysToRe
<wsp:Policy>
    <sp:WssUsernameToken10 />
    </wsp:Policy>
    </sp:UsernameToken>
</wsp:Policy>
</sp:SignedSupportingTokens>
<sp:Wss11
  xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
  <wsp:Policy>
    <sp:MustSupportRefKeyIdentifier />
    <sp:MustSupportRefIssuerSerial />
    <sp:MustSupportRefThumbprint />
    <sp:MustSupportRefEncryptedKey />
  </wsp:Policy>
</sp:Wss11>
<sp:Trust13
  xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
  <wsp:Policy>
    <sp:MustSupportIssuedTokens />
    <sp:RequireClientEntropy />
    <sp:RequireServerEntropy />
  </wsp:Policy>
</sp:Trust13>
</wsp:All>
</wsp:ExactlyOne>
</wsp:Policy>

<wsp:Policy wsu:Id="Input_policy">
  <wsp:ExactlyOne>
    <wsp:All>
      <sp:SignedParts
        xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
        <sp:Body />
        <sp:Header Name="To"
          Namespace="http://www.w3.org/2005/08/addressing" />
        <sp:Header Name="From"
          Namespace="http://www.w3.org/2005/08/addressing" />
        <sp:Header Name="FaultTo"
          Namespace="http://www.w3.org/2005/08/addressing" />
        <sp:Header Name="ReplyTo"
          Namespace="http://www.w3.org/2005/08/addressing" />
        <sp:Header Name="MessageID"
          Namespace="http://www.w3.org/2005/08/addressing" />
        <sp:Header Name="RelatesTo"
          Namespace="http://www.w3.org/2005/08/addressing" />
        <sp:Header Name="Action"
          Namespace="http://www.w3.org/2005/08/addressing" />
      </sp:SignedParts>
      <sp:EncryptedParts
        xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
        <sp:Body />
      </sp:EncryptedParts>
    </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>
```



```
<wsp:Policy wsu:Id="Output_policy">
  <wsp:ExactlyOne>
    <wsp:All>
      <sp:SignedParts
        xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
        <sp:Body />
        <sp:Header Name="To"
          Namespace="http://www.w3.org/2005/08/addressing" />
        <sp:Header Name="From"
          Namespace="http://www.w3.org/2005/08/addressing" />
        <sp:Header Name="FaultTo"
          Namespace="http://www.w3.org/2005/08/addressing" />
        <sp:Header Name="ReplyTo"
          Namespace="http://www.w3.org/2005/08/addressing" />
        <sp:Header Name="MessageID"
          Namespace="http://www.w3.org/2005/08/addressing" />
        <sp:Header Name="RelatesTo"
          Namespace="http://www.w3.org/2005/08/addressing" />
        <sp:Header Name="Action"
          Namespace="http://www.w3.org/2005/08/addressing" />
      </sp:SignedParts>
      <sp:EncryptedParts
        xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
        <sp:Body />
      </sp:EncryptedParts>
    </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>

</wsdl:definitions>
```

Differently from the Apache CXF STS example described above, the PicketLink based STS gets its configuration from a picketlink-sts.xml descriptor which must be added in WEB-INF into the deployment; please refer to the PicketLink documentation for further information:



```
<PicketLinkSTS xmlns="urn:picketlink:identity-federation:config:1.0"
  STSName="PicketLinkSTS" TokenTimeout="7200" EncryptToken="false">
  <KeyProvider ClassName="org.picketlink.identity.federation.core.impl.KeyStoreKeyManager">
    <Auth Key="KeyStoreURL" Value="stsstore.jks"/>
    <Auth Key="KeyStorePass" Value="stsspass"/>
    <Auth Key="SigningKeyAlias" Value="mystskey"/>
    <Auth Key="SigningKeyPass" Value="stskpass"/>
    <ValidatingAlias
Key="http://localhost:8080/jaxws-samples-wsse-policy-trust/SecurityService"
Value="myservicekey"/>
  </KeyProvider>
  <TokenProviders>
    <TokenProvider
ProviderClass="org.picketlink.identity.federation.core.wstrust.plugins.saml.SAML11TokenProvider"

TokenType="http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.1#SAMLV1.1"
  TokenElement="Assertion"
  TokenElementNS="urn:oasis:names:tc:SAML:1.0:assertion"/>
    </TokenProvider>
    <TokenProvider
ProviderClass="org.picketlink.identity.federation.core.wstrust.plugins.saml.SAML20TokenProvider"

TokenType="http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.1#SAMLV2.0"
  TokenElement="Assertion"
  TokenElementNS="urn:oasis:names:tc:SAML:2.0:assertion"/>
  </TokenProviders>
</PicketLinkSTS>
```

Finally, the PicketLink alternative approach of course requires different WildFly module dependencies to be declared in the MANIFEST.MF:

```
Manifest-Version: 1.0
Ant-Version: Apache Ant 1.8.2
Created-By: 1.6.0_26-b03 (Sun Microsystems Inc.)
Dependencies: org.apache.ws.security,org.apache.cxf,org.picketlink
```

Here is how the PicketLink STS endpoint is packaged:



```
alessio@inuyasha /dati/jboss/ws/stack/cxf/trunk $ jar -tvf
./modules/testsuite/cxf-tests/target/test-libs/jaxws-samples-wsse-policy-trustPicketLink-sts.war
 0 Mon Sep 03 17:38:38 CEST 2012 META-INF/
174 Mon Sep 03 17:38:36 CEST 2012 META-INF/MANIFEST.MF
 0 Mon Sep 03 17:38:38 CEST 2012 WEB-INF/
 0 Mon Sep 03 17:38:38 CEST 2012 WEB-INF/classes/
 0 Mon Sep 03 16:35:50 CEST 2012 WEB-INF/classes/org/
 0 Mon Sep 03 16:35:50 CEST 2012 WEB-INF/classes/org/jboss/
 0 Mon Sep 03 16:35:50 CEST 2012 WEB-INF/classes/org/jboss/test/
 0 Mon Sep 03 16:35:52 CEST 2012 WEB-INF/classes/org/jboss/test/ws/
 0 Mon Sep 03 16:35:50 CEST 2012 WEB-INF/classes/org/jboss/test/ws/jaxws/
 0 Mon Sep 03 16:35:52 CEST 2012 WEB-INF/classes/org/jboss/test/ws/jaxws/samples/
 0 Mon Sep 03 16:35:50 CEST 2012 WEB-INF/classes/org/jboss/test/ws/jaxws/samples/wsse/
 0 Mon Sep 03 16:35:50 CEST 2012
WEB-INF/classes/org/jboss/test/ws/jaxws/samples/wsse/policy/
 0 Mon Sep 03 16:35:52 CEST 2012
WEB-INF/classes/org/jboss/test/ws/jaxws/samples/wsse/policy/trust/
1686 Mon Sep 03 16:35:50 CEST 2012
WEB-INF/classes/org/jboss/test/ws/jaxws/samples/wsse/policy/trust/PicketLinkSTService.class
1148 Mon Sep 03 16:35:52 CEST 2012
WEB-INF/classes/org/jboss/test/ws/jaxws/samples/wsse/policy/trust/STSCallbackHandler.class
251 Mon Sep 03 17:38:34 CEST 2012 WEB-INF/jboss-web.xml
 0 Mon Sep 03 16:35:50 CEST 2012 WEB-INF/wsd/
9070 Mon Sep 03 17:38:34 CEST 2012 WEB-INF/wsd/PicketLinkSTS.wsdl
1267 Mon Sep 03 17:38:34 CEST 2012 WEB-INF/classes/picketlink-sts.xml
1054 Mon Sep 03 16:35:50 CEST 2012 WEB-INF/classes/stsKeystore.properties
3978 Mon Sep 03 16:35:50 CEST 2012 WEB-INF/classes/stsstore.jks
```

ActAs WS-Trust Scenario

- [ActAs WS-Trust Scenario](#)
 - [Web service provider](#)
 - [Web service provider WSDL](#)
 - [Web Service Interface](#)
 - [Web Service Implementation](#)
 - [ActAsCallbackHandler](#)
 - [UsernameTokenCallbackHandler](#)
 - [Crypto properties and keystore files](#)
 - [MANIFEST.MF](#)
 - [Security Token Service](#)
 - [STS Implementation class](#)
 - [STSCallbackHandler](#)
 - [Web service requester](#)
 - [Web service requester Implementation](#)



ActAs WS-Trust Scenario

The ActAs feature is used in scenarios that require composite delegation. It is commonly used in multi-tiered systems where an application calls a service on behalf of a logged in user or a service calls another service on behalf of the original caller.

ActAs is nothing more than a new sub-element in the RequestSecurityToken (RST). It provides additional information about the original caller when a token is negotiated with the STS. The ActAs element usually takes the form of a token with identity claims such as name, role, and authorization code, for the client to access the service.

The ActAs scenario is an extension of [the basic WS-Trust scenario](#). In this example the ActAs service calls the ws-service on behalf of a user. There are only a couple of additions to the basic scenario's code. An ActAs web service provider and callback handler have been added. The ActAs web services' WSDL imposes the same security policies as the ws-provider. UsernameTokenCallbackHandler is new. It is a utility that generates the content for the ActAs element. And lastly there are a couple of code additions in the STS to support the ActAs request.

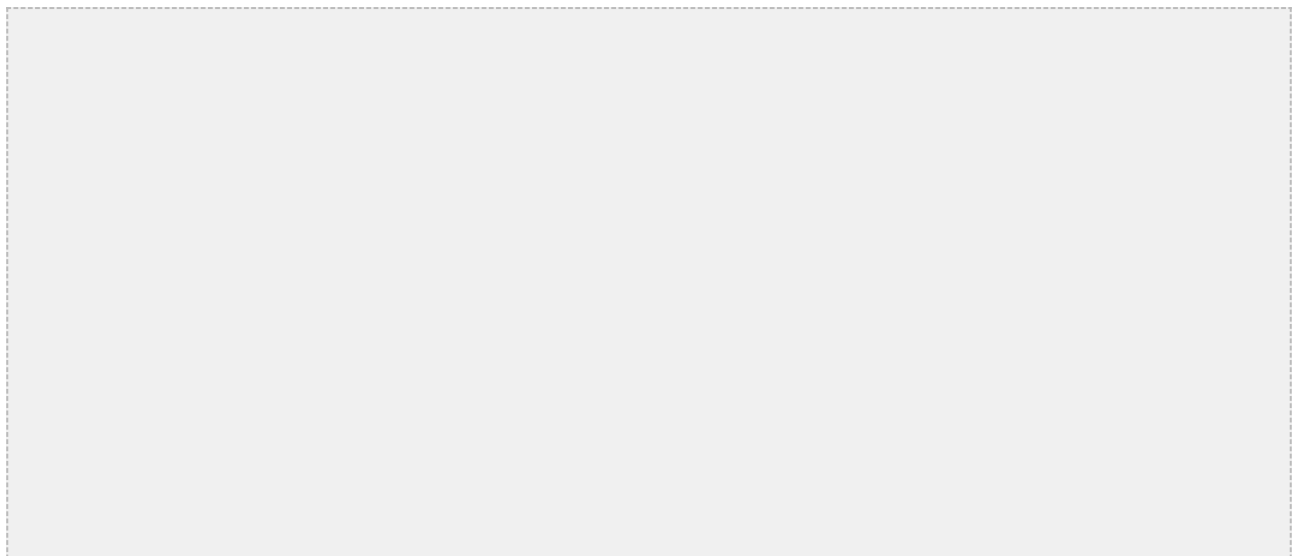
Web service provider

This section examines the web service elements from the basic WS-Trust scenario that have been changed to address the needs of the ActAs example. The components are

- ActAs web service provider's WSDL
- ActAs web service provider's Interface and Implementation classes.
- ActAsCallbackHandler class
- UsernameTokenCallbackHandler
- Crypto properties and keystore files
- MANIFEST.MF

Web service provider WSDL

The ActAs web service provider's WSDL is a clone of the ws-provider's WSDL. The `wsp:Policy` section is the same. There are changes to the service endpoint, `targetNamespace`, `portType`, binding name, and service.





```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<definitions targetNamespace="http://www.jboss.org/jbossws/ws-extensions/actaswssecuritypolicy"
name="ActAsService"
    xmlns:tns="http://www.jboss.org/jbossws/ws-extensions/actaswssecuritypolicy"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:wsp="http://www.w3.org/ns/ws-policy"
    xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata"

xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
    xmlns:wsaws="http://www.w3.org/2005/08/addressing"
    xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702"
    xmlns:t="http://docs.oasis-open.org/ws-sx/ws-trust/200512">
    <types>
        <xsd:schema>
            <xsd:import
namespace="http://www.jboss.org/jbossws/ws-extensions/actaswssecuritypolicy"
                schemaLocation="ActAsService_schema1.xsd" />
            </xsd:schema>
        </types>
        <message name="sayHello">
            <part name="parameters" element="tns:sayHello" />
        </message>
        <message name="sayHelloResponse">
            <part name="parameters" element="tns:sayHelloResponse" />
        </message>
        <portType name="ActAsServiceIface">
            <operation name="sayHello">
                <input message="tns:sayHello" />
                <output message="tns:sayHelloResponse" />
            </operation>
        </portType>
        <binding name="ActAsServicePortBinding" type="tns:ActAsServiceIface">
            <wsp:PolicyReference URI="#AsymmetricSAML2Policy" />
            <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document" />
            <operation name="sayHello">
                <soap:operation soapAction="" />
                <input>
                    <soap:body use="literal" />
                    <wsp:PolicyReference URI="#Input_Policy" />
                </input>
                <output>
                    <soap:body use="literal" />
                    <wsp:PolicyReference URI="#Output_Policy" />
                </output>
            </operation>
        </binding>
        <service name="ActAsService">
            <port name="ActAsServicePort" binding="tns:ActAsServicePortBinding">
                <soap:address
location="http://@jboss.bind.address@:8080/jaxws-samples-wsse-policy-trust-actas/ActAsService" />
                </port>
            </service>
    </definitions>
```



Web Service Interface

The web service provider interface class, `ActAsServiceIface`, is a simple web service definition.

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.actas;

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

@WebService
(
    targetNamespace = "http://www.jboss.org/jbossws/ws-extensions/actaswssecuritypolicy"
)
public interface ActAsServiceIface
{
    @WebMethod
    String sayHello();
}
```

Web Service Implementation

The web service provider implementation class, `ActAsServiceImpl`, is a simple POJO. It uses the standard `WebService` annotation to define the service endpoint and two Apache WSS4J annotations, `EndpointProperties` and `EndpointProperty` used for configuring the endpoint for the CXF runtime. The WSS4J configuration information provided is for WSS4J's Crypto Merlin implementation.

`ActAsServiceImpl` is calling `ServiceImpl` acting on behalf of the user. Method `setupService` performs the requisite configuration setup.

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.actas;

import org.apache.cxf.Bus;
import org.apache.cxf.BusFactory;
import org.apache.cxf.annotations.EndpointProperties;
import org.apache.cxf.annotations.EndpointProperty;
import org.apache.cxf.ws.security.SecurityConstants;
import org.apache.cxf.ws.security.trust.STSClient;
import org.jboss.test.ws.jaxws.samples.wsse.policy.trust.service.ServiceIface;
import org.jboss.test.ws.jaxws.samples.wsse.policy.trust.shared.WSTrustAppUtils;

import javax.jws.WebService;
import javax.xml.namespace.QName;
import javax.xml.ws.BindingProvider;
import javax.xml.ws.Service;
import java.net.MalformedURLException;
import java.net.URL;
import java.util.Map;

@WebService
(
    portName = "ActAsServicePort",
    serviceName = "ActAsService",
    wsdlLocation = "WEB-INF/wsdl/ActAsService.wsdl",
    targetNamespace = "http://www.jboss.org/jbossws/ws-extensions/actaswssecuritypolicy",
```




```
    endpointInterface =
"org.jboss.test.ws.jaxws.samples.wsse.policy.trust.actas.ActAsServiceIface"
)

@EndpointProperties(value = {
    @EndpointProperty(key = "ws-security.signature.username", value = "myactaskey"),
    @EndpointProperty(key = "ws-security.signature.properties", value =
"actasKeystore.properties"),
    @EndpointProperty(key = "ws-security.encryption.properties", value =
"actasKeystore.properties"),
    @EndpointProperty(key = "ws-security.callback-handler", value =
"org.jboss.test.ws.jaxws.samples.wsse.policy.trust.actas.ActAsCallbackHandler")
})

public class ActAsServiceImpl implements ActAsServiceIface
{
    public String sayHello() {
        try {
            ServiceIface proxy = setupService();
            return "ActAs " + proxy.sayHello();
        } catch (MalformedURLException e) {
            e.printStackTrace();
        }
        return null;
    }

    private ServiceIface setupService()throws MalformedURLException {
        ServiceIface proxy = null;
        Bus bus = BusFactory.newInstance().createBus();

        try {
            BusFactory.setThreadDefaultBus(bus);

            final String serviceURL = "http://" + WSTrustAppUtils.getServerHost() +
":8080/jaxws-samples-wsse-policy-trust/SecurityService";
            final QName serviceName = new
QName("http://www.jboss.org/jbossws/ws-extensions/wssecuritypolicy", "SecurityService");
            final URL wsdlURL = new URL(serviceURL + "?wsdl");
            Service service = Service.create(wsdlURL, serviceName);
            proxy = (ServiceIface) service.getPort(ServiceIface.class);

            Map<String, Object> ctx = ((BindingProvider) proxy).getRequestContext();
            ctx.put(SecurityConstants.CALLBACK_HANDLER, new ActAsCallbackHandler());

            ctx.put(SecurityConstants.SIGNATURE_PROPERTIES,

Thread.currentThread().getContextClassLoader().getResource("actasKeystore.properties" ));
            ctx.put(SecurityConstants.SIGNATURE_USERNAME, "myactaskey" );
            ctx.put(SecurityConstants.ENCRYPT_PROPERTIES,

Thread.currentThread().getContextClassLoader().getResource("../META-INF/clientKeystore.properti
));
            ctx.put(SecurityConstants.ENCRYPT_USERNAME, "myservicekey");

            STSClient stsClient = new STSClient(bus);
            Map<String, Object> props = stsClient.getProperties();
            props.put(SecurityConstants.USERNAME, "alice");
            props.put(SecurityConstants.ENCRYPT_USERNAME, "mystskey");
```



```
        props.put(SecurityConstants.STS_TOKEN_USERNAME, "myactaskey" );
        props.put(SecurityConstants.STS_TOKEN_PROPERTIES,

Thread.currentThread().getContextClassLoader().getResource("actasKeystore.properties" ));
        props.put(SecurityConstants.STS_TOKEN_USE_CERT_FOR_KEYINFO, "true");

        ctx.put(SecurityConstants.STS_CLIENT, stsClient);

    } finally {
        bus.shutdown(true);
    }

    return proxy;
}
}
```

ActAsCallbackHandler

ActAsCallbackHandler is a callback handler for the WSS4J Crypto API. It is used to obtain the password for the private key in the keystore. This class enables CXF to retrieve the password of the user name to use for the message signature. This class has been revised to return the passwords for this service, myactaskey and the "actas" user, alice.

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.actas;

import org.jboss.wsf.stack.cxf.extensions.security.PasswordCallbackHandler;
import java.util.HashMap;
import java.util.Map;

public class ActAsCallbackHandler extends PasswordCallbackHandler {

    public ActAsCallbackHandler()
    {
        super(getInitMap());
    }

    private static Map<String, String> getInitMap()
    {
        Map<String, String> passwords = new HashMap<String, String>();
        passwords.put("myactaskey", "aspass");
        passwords.put("alice", "clarinet");
        return passwords;
    }
}
```

UsernameTokenCallbackHandler

The ActAs and OnBeholdOf sub-elements of the RequestSecurityToken are required to be defined as WSSE Username Tokens. This utility generates the properly formatted element.

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.shared;

import org.apache.cxf.helpers.DOMUtils;
```



```
import org.apache.cxf.message.Message;
import org.apache.cxf.ws.security.SecurityConstants;
import org.apache.cxf.ws.security.trust.delegation.DelegationCallback;
import org.apache.ws.security.WSConstants;
import org.apache.ws.security.message.token.UsernameToken;
import org.w3c.dom.Document;
import org.w3c.dom.Node;
import org.w3c.dom.Element;
import org.w3c.dom.ls.DOMImplementationLS;
import org.w3c.dom.ls.LSSerializer;

import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.UnsupportedCallbackException;
import java.io.IOException;
import java.util.Map;

/**
 * A utility to provide the 3 different input parameter types for jaxws property
 * "ws-security.sts.token.act-as" and "ws-security.sts.token.on-behalf-of".
 * This implementation obtains a username and password via the jaxws property
 * "ws-security.username" and "ws-security.password" respectively, as defined
 * in SecurityConstants. It creates a wss UsernameToken to be used as the
 * delegation token.
 */

public class UsernameTokenCallbackHandler implements CallbackHandler {

    public void handle(Callback[] callbacks)
        throws IOException, UnsupportedCallbackException {
        for (int i = 0; i < callbacks.length; i++) {
            if (callbacks[i] instanceof DelegationCallback) {
                DelegationCallback callback = (DelegationCallback) callbacks[i];
                Message message = callback.getCurrentMessage();

                String username =
                    (String)message.getContextualProperty(SecurityConstants.USERNAME);
                String password =
                    (String)message.getContextualProperty(SecurityConstants.PASSWORD);
                if (username != null) {
                    Node contentNode = message.getContent(Node.class);
                    Document doc = null;
                    if (contentNode != null) {
                        doc = contentNode.getOwnerDocument();
                    } else {
                        doc = DOMUtils.createDocument();
                    }
                    UsernameToken usernameToken = createWSSEUsernameToken(username,password, doc);
                    callback.setToken(usernameToken.getElement());
                }
            } else {
                throw new UnsupportedCallbackException(callbacks[i], "Unrecognized Callback");
            }
        }
    }

    /**
     * Provide UsernameToken as a string.
     */
}
```



```
* @param ctx
* @return
*/
public String getUsernameTokenString(Map<String, Object> ctx){
    Document doc = DOMUtils.createDocument();
    String result = null;
    String username = (String)ctx.get(SecurityConstants.USERNAME);
    String password = (String)ctx.get(SecurityConstants.PASSWORD);
    if (username != null) {
        UsernameToken usernameToken = createWSSEUsernameToken(username,password, doc);
        result = toString(usernameToken.getElement().getFirstChild().getParentNode());
    }
    return result;
}

/**
 *
 * @param username
 * @param password
 * @return
 */
public String getUsernameTokenString(String username, String password){
    Document doc = DOMUtils.createDocument();
    String result = null;
    if (username != null) {
        UsernameToken usernameToken = createWSSEUsernameToken(username,password, doc);
        result = toString(usernameToken.getElement().getFirstChild().getParentNode());
    }
    return result;
}

/**
 * Provide UsernameToken as a DOM Element.
 * @param ctx
 * @return
 */
public Element getUsernameTokenElement(Map<String, Object> ctx){
    Document doc = DOMUtils.createDocument();
    Element result = null;
    UsernameToken usernameToken = null;
    String username = (String)ctx.get(SecurityConstants.USERNAME);
    String password = (String)ctx.get(SecurityConstants.PASSWORD);
    if (username != null) {
        usernameToken = createWSSEUsernameToken(username,password, doc);
        result = usernameToken.getElement();
    }
    return result;
}

/**
 *
 * @param username
 * @param password
 * @return
 */
public Element getUsernameTokenElement(String username, String password){
    Document doc = DOMUtils.createDocument();
    Element result = null;
```



```
UsernameToken usernameToken = null;
if (username != null) {
    usernameToken = createWSSEUsernameToken(username,password, doc);
    result = usernameToken.getElement();
}
return result;
}

private UsernameToken createWSSEUsernameToken(String username, String password, Document doc)
{

    UsernameToken usernameToken = new UsernameToken(true, doc,
        (password == null)? null: WSConstants.PASSWORD_TEXT);
    usernameToken.setName(username);
    usernameToken.addWSUNamespace();
    usernameToken.addWSSNamespace();
    usernameToken.setID("id-" + username);

    if (password != null){
        usernameToken.setPassword(password);
    }

    return usernameToken;
}

private String toString(Node node) {
    String str = null;

    if (node != null) {
        DOMImplementationLS lsImpl = (DOMImplementationLS)
            node.getOwnerDocument().getImplementation().getFeature("LS", "3.0");
        LSSerializer serializer = lsImpl.createLSSerializer();
        serializer.getDomConfig().setParameter("xml-declaration", false); //by default its
true, so set it to false to get String without xml-declaration
        str = serializer.writeToString(node);
    }
    return str;
}
}
```

Crypto properties and keystore files

The ActAs service must provide its own credentials. The requisite properties file, `actasKeystore.properties`, and keystore, `actasstore.jks`, were created.

```
org.apache.ws.security.crypto.provider=org.apache.ws.security.components.crypto.Merlin
org.apache.ws.security.crypto.merlin.keystore.type=jks
org.apache.ws.security.crypto.merlin.keystore.password=aapass
org.apache.ws.security.crypto.merlin.keystore.alias=myactaskey
org.apache.ws.security.crypto.merlin.keystore.file=actasstore.jks
```



MANIFEST.MF

When deployed on WildFly this application requires access to the JBossWS and CXF APIs provided in modules `org.jboss.ws.cxf.jbossws-cxf-client` and `org.apache.cxf`. The Apache CXF internals, `org.apache.cxf.impl`, are needed in handling the ActAs and OnBehalfOf extensions. The dependency statement directs the server to provide them at deployment.

```
Manifest-Version: 1.0
Ant-Version: Apache Ant 1.8.2
Created-By: 1.7.0_25-b15 (Oracle Corporation)
Dependencies: org.jboss.ws.cxf.jbossws-cxf-client, org.apache.cxf.impl
```

Security Token Service

This section examines the STS elements from the basic WS-Trust scenario that have been changed to address the needs of the ActAs example. The components are.

- STS's implementation class.
- STSCallbackHandler class

STS Implementation class

The initial description of SampleSTS can be found [here](#).

The declaration of the set of allowed token recipients by address has been extended to accept ActAs addresses and OnBehalfOf addresses. The addresses are specified as reg-ex patterns.

The TokenIssueOperation requires class, UsernameTokenValidator be provided in order to validate the contents of the OnBehalfOf claims and class, UsernameTokenDelegationHandler to be provided in order to process the token delegation request of the ActAs on OnBehalfOf user.

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.sts;

import java.util.Arrays;
import java.util.LinkedList;
import java.util.List;

import javax.xml.ws.WebServiceProvider;

import org.apache.cxf.annotations.EndpointProperties;
import org.apache.cxf.annotations.EndpointProperty;
import org.apache.cxf.interceptor.InInterceptors;
import org.apache.cxf.sts.StaticSTSProperties;
import org.apache.cxf.sts.operation.TokenIssueOperation;
import org.apache.cxf.sts.operation.TokenValidateOperation;
import org.apache.cxf.sts.service.ServiceMBean;
import org.apache.cxf.sts.service.StaticService;
import org.apache.cxf.sts.token.delegation.UsernameTokenDelegationHandler;
import org.apache.cxf.sts.token.provider.SAMLTokenProvider;
import org.apache.cxf.sts.token.validator.SAMLTokenValidator;
import org.apache.cxf.sts.token.validator.UsernameTokenValidator;
import org.apache.cxf.ws.security.sts.provider.SecurityTokenServiceProvider;
```



```
@WebServiceProvider(serviceName = "SecurityTokenService",
    portName = "UT_Port",
    targetNamespace = "http://docs.oasis-open.org/ws-sx/ws-trust/200512/",
    wsdlLocation = "WEB-INF/wsdl/ws-trust-1.4-service.wsdl")
//be sure to have dependency on org.apache.cxf module when on AS7, otherwise Apache CXF
annotations are ignored
@EndpointProperties(value = {
    @EndpointProperty(key = "ws-security.signature.username", value = "mystskey"),
    @EndpointProperty(key = "ws-security.signature.properties", value =
"stsKeystore.properties"),
    @EndpointProperty(key = "ws-security.callback-handler", value =
"org.jboss.test.ws.jaxws.samples.wsse.policy.trust.sts.STSCallbackHandler"),
    @EndpointProperty(key = "ws-security.validate.token", value = "false") //to let the JAAS
integration deal with validation through the interceptor below
})
@InInterceptors(interceptors =
{"org.jboss.wsf.stack.cxf.security.authentication.SubjectCreatingPolicyInterceptor"})
public class SampleSTS extends SecurityTokenServiceProvider
{
    public SampleSTS() throws Exception
    {
        super();

        StaticSTSProperties props = new StaticSTSProperties();
        props.setSignatureCryptoProperties("stsKeystore.properties");
        props.setSignatureUsername("mystskey");
        props.setCallbackHandlerClass(STSCallbackHandler.class.getName());
        props.setIssuer("DoubleItSTSIssuer");

        List<ServiceMBean> services = new LinkedList<ServiceMBean>();
        StaticService service = new StaticService();
        service.setEndpoints(Arrays.asList(
            "http://localhost:(\\d)*/jaxws-samples-wsse-policy-trust/SecurityService",
            "http://\\[::1\\]:(\\d)*/jaxws-samples-wsse-policy-trust/SecurityService",
            "http://\\[0:0:0:0:0:0:1\\]:(\\d)*/jaxws-samples-wsse-policy-trust/SecurityService",

            "http://localhost:(\\d)*/jaxws-samples-wsse-policy-trust-actas/ActAsService",
            "http://\\[::1\\]:(\\d)*/jaxws-samples-wsse-policy-trust-actas/ActAsService",

"http://\\[0:0:0:0:0:0:1\\]:(\\d)*/jaxws-samples-wsse-policy-trust-actas/ActAsService",

            "http://localhost:(\\d)*/jaxws-samples-wsse-policy-trust-onbehalfOf/OnBehalfOfService",
            "http://\\[::1\\]:(\\d)*/jaxws-samples-wsse-policy-trust-onbehalfOf/OnBehalfOfService",

"http://\\[0:0:0:0:0:0:1\\]:(\\d)*/jaxws-samples-wsse-policy-trust-onbehalfOf/OnBehalfOfService"
));
        services.add(service);

        TokenIssueOperation issueOperation = new TokenIssueOperation();
        issueOperation.setServices(services);
        issueOperation.getTokenProviders().add(new SAMLTokenProvider());
        // required for OnBehalfOf
        issueOperation.getTokenValidators().add(new UsernameTokenValidator());
        // added for OnBehalfOf and ActAs
        issueOperation.getDelegationHandlers().add(new UsernameTokenDelegationHandler());
        issueOperation.setStsProperties(props);

        TokenValidateOperation validateOperation = new TokenValidateOperation();
```



```
validateOperation.getTokenValidators().add(new SAMLTokenValidator());
validateOperation.setStsProperties(props);

this.setIssueOperation(issueOperation);
this.setValidateOperation(validateOperation);
}
}
```

STSCallbackHandler

The user, alice, and corresponding password was required to be added for the ActAs example.

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.sts;

import java.util.HashMap;
import java.util.Map;

import org.jboss.wsf.stack.cxf.extensions.security.PasswordCallbackHandler;

public class STSCallbackHandler extends PasswordCallbackHandler
{
    public STSCallbackHandler()
    {
        super(getInitMap());
    }

    private static Map<String, String> getInitMap()
    {
        Map<String, String> passwords = new HashMap<String, String>();
        passwords.put("mystskey", "stskpass");
        passwords.put("alice", "clarinet");
        return passwords;
    }
}
```

Web service requester

This section examines the ws-requester elements from the basic WS-Trust scenario that have been changed to address the needs of the ActAs example. The component is

- ActAs web service requester implementation class

Web service requester Implementation

The ActAs ws-requester, the client, uses standard procedures for creating a reference to the web service in the first four lines. To address the endpoint security requirements, the web service's "Request Context" is configured via the BindingProvider. Information needed in the message generation is provided through it. The ActAs user, myactaskey, is declared in this section and UsernameTokenCallbackHandler is used to provide the contents of the ActAs element to the STSClient. In this example a STSClient object is created and provided to the proxy's request context. The alternative is to provide keys tagged with the ".it" suffix as was done in [the Basic Scenario client](#). The use of ActAs is configured through the props map using the SecurityConstants.STS_TOKEN_ACT_AS key. The alternative is to use the STSClient.setActAs method.



```
final QName serviceName = new
QName("http://www.jboss.org/jbossws/ws-extensions/actaswssecuritypolicy", "ActAsService");
final URL wsdlURL = new URL(serviceURL + "?wsdl");
Service service = Service.create(wsdlURL, serviceName);
ActAsServiceIface proxy = (ActAsServiceIface) service.getPort(ActAsServiceIface.class);

Bus bus = BusFactory.newInstance().createBus();
try {
    BusFactory.setThreadDefaultBus(bus);

    Map<String, Object> ctx = proxy.getRequestContext();

    ctx.put(SecurityConstants.CALLBACK_HANDLER, new ClientCallbackHandler());
    ctx.put(SecurityConstants.ENCRYPT_PROPERTIES,
        Thread.currentThread().getContextClassLoader().getResource(
            "META-INF/clientKeystore.properties"));
    ctx.put(SecurityConstants.ENCRYPT_USERNAME, "myactaskey");
    ctx.put(SecurityConstants.SIGNATURE_PROPERTIES,
        Thread.currentThread().getContextClassLoader().getResource(
            "META-INF/clientKeystore.properties"));
    ctx.put(SecurityConstants.SIGNATURE_USERNAME, "myclientkey");

    // Generate the ActAs element contents and pass to the STSClient as a string
    UsernameTokenCallbackHandler ch = new UsernameTokenCallbackHandler();
    String str = ch.getUsernameTokenString("alice","clarinet");
    ctx.put(SecurityConstants.STS_TOKEN_ACT_AS, str);

    STSClient stsClient = new STSClient(bus);
    Map<String, Object> props = stsClient.getProperties();
    props.put(SecurityConstants.USERNAME, "bob");
    props.put(SecurityConstants.CALLBACK_HANDLER, new ClientCallbackHandler());
    props.put(SecurityConstants.ENCRYPT_PROPERTIES,
        Thread.currentThread().getContextClassLoader().getResource(
            "META-INF/clientKeystore.properties"));
    props.put(SecurityConstants.ENCRYPT_USERNAME, "mystskey");
    props.put(SecurityConstants.STS_TOKEN_USERNAME, "myclientkey");
    props.put(SecurityConstants.STS_TOKEN_PROPERTIES,
        Thread.currentThread().getContextClassLoader().getResource(
            "META-INF/clientKeystore.properties"));
    props.put(SecurityConstants.STS_TOKEN_USE_CERT_FOR_KEYINFO, "true");

    ctx.put(SecurityConstants.STS_CLIENT, stsClient);
} finally {
    bus.shutdown(true);
}
proxy.sayHello();
```



OnBehalfOf WS-Trust Scenario

- [OnBehalfOf WS-Trust Scenario](#)
 - [Web service provider](#)
 - [Web service provider WSDL](#)
 - [Web Service Interface](#)
 - [Web Service Implementation](#)
 - [OnBehalfOfCallbackHandler](#)
 - [Web service requester](#)
 - [Web service requester Implementation](#)

OnBehalfOf WS-Trust Scenario

The OnBehalfOf feature is used in scenarios that use the proxy pattern. In such scenarios, the client cannot access the STS directly, instead it communicates through a proxy gateway. The proxy gateway authenticates the caller and puts information about the caller into the OnBehalfOf element of the RequestSecurityToken (RST) sent to the real STS for processing. The resulting token contains only claims related to the client of the proxy, making the proxy completely transparent to the receiver of the issued token.

OnBehalfOf is nothing more than a new sub-element in the RST. It provides additional information about the original caller when a token is negotiated with the STS. The OnBehalfOf element usually takes the form of a token with identity claims such as name, role, and authorization code, for the client to access the service.

The OnBehalfOf scenario is an extension of [the basic WS-Trust scenario](#). In this example the OnBehalfOf service calls the ws-service on behalf of a user. There are only a couple of additions to the basic scenario's code. An OnBehalfOf web service provider and callback handler have been added. The OnBehalfOf web services' WSDL imposes the same security policies as the ws-provider. UsernameTokenCallbackHandler is a utility shared with ActAs. It generates the content for the OnBehalfOf element. And lastly there are code additions in the STS that both OnBehalfOf and ActAs share in common.

For here [[Open Source Security: Apache CXF 2.5.1 STS updates](#)]

Web service provider

This section examines the web service elements from the basic WS-Trust scenario that have been changed to address the needs of the OnBehalfOf example. The components are.

- web service provider's WSDL
- web service provider's Interface and Implementation classes.
- OnBehalfOfCallbackHandler class

Web service provider WSDL

The OnBehalfOf web service provider's WSDL is a clone of the ws-provider's WSDL. The `wsp:Policy` section is the same. There are changes to the service endpoint, `targetNamespace`, `portType`, binding name, and service.





```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<definitions
targetNamespace="http://www.jboss.org/jbossws/ws-extensions/onbehalfofwssecuritypolicy"
name="OnBehalfOfService"
    xmlns:tns="http://www.jboss.org/jbossws/ws-extensions/onbehalfofwssecuritypolicy"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:wsp="http://www.w3.org/ns/ws-policy"
    xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata"

xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
    xmlns:wsaws="http://www.w3.org/2005/08/addressing"
    xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702"
    xmlns:t="http://docs.oasis-open.org/ws-sx/ws-trust/200512">
    <types>
        <xsd:schema>
            <xsd:import
namespace="http://www.jboss.org/jbossws/ws-extensions/onbehalfofwssecuritypolicy"
                schemaLocation="OnBehalfOfService_schema1.xsd" />
            </xsd:schema>
        </types>
        <message name="sayHello">
            <part name="parameters" element="tns:sayHello" />
        </message>
        <message name="sayHelloResponse">
            <part name="parameters" element="tns:sayHelloResponse" />
        </message>
        <portType name="OnBehalfOfServiceIface">
            <operation name="sayHello">
                <input message="tns:sayHello" />
                <output message="tns:sayHelloResponse" />
            </operation>
        </portType>
        <binding name="OnBehalfOfServicePortBinding" type="tns:OnBehalfOfServiceIface">
            <wsp:PolicyReference URI="#AsymmetricSAML2Policy" />
            <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document" />
            <operation name="sayHello">
                <soap:operation soapAction="" />
                <input>
                    <soap:body use="literal" />
                    <wsp:PolicyReference URI="#Input_Policy" />
                </input>
                <output>
                    <soap:body use="literal" />
                    <wsp:PolicyReference URI="#Output_Policy" />
                </output>
            </operation>
        </binding>
        <service name="OnBehalfOfService">
            <port name="OnBehalfOfServicePort" binding="tns:OnBehalfOfServicePortBinding">
                <soap:address
location="http://@jboss.bind.address@:8080/jaxws-samples-wsse-policy-trust-onbehalfof/OnBehalfOfSe" />
            </port>
        </service>
    </definitions>
```



Web Service Interface

The web service provider interface class, `OnBehalfOfServiceIface`, is a simple web service definition.

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.onbehalfof;

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

@WebService
(
    targetNamespace = "http://www.jboss.org/jbossws/ws-extensions/onbehalfofwssecuritypolicy"
)
public interface OnBehalfOfServiceIface
{
    @WebMethod
    String sayHello();
}
```

Web Service Implementation

The web service provider implementation class, `OnBehalfOfServiceImpl`, is a simple POJO. It uses the standard `WebService` annotation to define the service endpoint and two Apache WSS4J annotations, `EndpointProperties` and `EndpointProperty` used for configuring the endpoint for the CXF runtime. The WSS4J configuration information provided is for WSS4J's Crypto Merlin implementation.

`OnBehalfOfServiceImpl` is calling the `ServiceImpl` acting on behalf of the user. Method `setupService` performs the requisite configuration setup.

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.onbehalfof;

import org.apache.cxf.Bus;
import org.apache.cxf.BusFactory;
import org.apache.cxf.annotations.EndpointProperties;
import org.apache.cxf.annotations.EndpointProperty;
import org.apache.cxf.ws.security.SecurityConstants;
import org.apache.cxf.ws.security.trust.STSClient;
import org.jboss.test.ws.jaxws.samples.wsse.policy.trust.service.ServiceIface;
import org.jboss.test.ws.jaxws.samples.wsse.policy.trust.shared.WSTrustAppUtils;

import javax.jws.WebService;
import javax.xml.namespace.QName;
import javax.xml.ws.BindingProvider;
import javax.xml.ws.Service;
import java.net.*;
import java.util.Map;

@WebService
(
    portName = "OnBehalfOfServicePort",
    serviceName = "OnBehalfOfService",
    wsdlLocation = "WEB-INF/wsdl/OnBehalfOfService.wsdl",
    targetNamespace = "http://www.jboss.org/jbossws/ws-extensions/onbehalfofwssecuritypolicy",
    endpointInterface =
```



```
"org.jboss.test.ws.jaxws.samples.wsse.policy.trust.onbehalfof.OnBehalfOfServiceIface"
)

@EndpointProperties(value = {
    @EndpointProperty(key = "ws-security.signature.username", value = "myactaskey"),
    @EndpointProperty(key = "ws-security.signature.properties", value =
"actasKeystore.properties"),
    @EndpointProperty(key = "ws-security.encryption.properties", value =
"actasKeystore.properties"),
    @EndpointProperty(key = "ws-security.callback-handler", value =
"org.jboss.test.ws.jaxws.samples.wsse.policy.trust.onbehalfof.OnBehalfOfCallbackHandler")
})

public class OnBehalfOfServiceImpl implements OnBehalfOfServiceIface
{
    public String sayHello() {
        try {

            ServiceIface proxy = setupService();
            return "OnBehalfOf " + proxy.sayHello();

        } catch (MalformedURLException e) {
            e.printStackTrace();
        }
        return null;
    }

    /**
     *
     * @return
     * @throws MalformedURLException
     */
    private ServiceIface setupService()throws MalformedURLException {
        ServiceIface proxy = null;
        Bus bus = BusFactory.newInstance().createBus();

        try {
            BusFactory.setThreadDefaultBus(bus);

            final String serviceURL = "http://" + WSTrustAppUtils.getServerHost() +
":8080/jaxws-samples-wsse-policy-trust/SecurityService";
            final QName serviceName = new
QName("http://www.jboss.org/jbossws/ws-extensions/wssecuritypolicy", "SecurityService");
            final URL wsdlURL = new URL(serviceURL + "?wsdl");
            Service service = Service.create(wsdlURL, serviceName);
            proxy = (ServiceIface) service.getPort(ServiceIface.class);

            Map<String, Object> ctx = ((BindingProvider) proxy).getRequestContext();
            ctx.put(SecurityConstants.CALLBACK_HANDLER, new OnBehalfOfCallbackHandler());

            ctx.put(SecurityConstants.SIGNATURE_PROPERTIES,
                Thread.currentThread().getContextClassLoader().getResource(
                    "actasKeystore.properties" ));
            ctx.put(SecurityConstants.SIGNATURE_USERNAME, "myactaskey" );
            ctx.put(SecurityConstants.ENCRYPT_PROPERTIES,
                Thread.currentThread().getContextClassLoader().getResource(
                    "../../META-INF/clientKeystore.properties" ));
            ctx.put(SecurityConstants.ENCRYPT_USERNAME, "myservicekey");
```



```
    STSClient stsClient = new STSClient(bus);
    Map<String, Object> props = stsClient.getProperties();
    props.put(SecurityConstants.USERNAME, "bob");
    props.put(SecurityConstants.ENCRYPT_USERNAME, "mystskey");
    props.put(SecurityConstants.STS_TOKEN_USERNAME, "myactaskey" );
    props.put(SecurityConstants.STS_TOKEN_PROPERTIES,
        Thread.currentThread().getContextClassLoader().getResource(
            "actasKeystore.properties" ));
    props.put(SecurityConstants.STS_TOKEN_USE_CERT_FOR_KEYINFO, "true");

    ctx.put(SecurityConstants.STS_CLIENT, stsClient);

} finally {
    bus.shutdown(true);
}

return proxy;
}
}
```



OnBehalfOfCallbackHandler

OnBehalfOfCallbackHandler is a callback handler for the WSS4J Crypto API. It is used to obtain the password for the private key in the keystore. This class enables CXF to retrieve the password of the user name to use for the message signature. This class has been revised to return the passwords for this service, myactaskey and the "OnBehalfOf" user, alice.

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.onbehalfof;

import org.jboss.wsf.stack.cxf.extensions.security.PasswordCallbackHandler;
import java.util.HashMap;
import java.util.Map;

public class OnBehalfOfCallbackHandler extends PasswordCallbackHandler {

    public OnBehalfOfCallbackHandler()
    {
        super(getInitMap());
    }

    private static Map<String, String> getInitMap()
    {
        Map<String, String> passwords = new HashMap<String, String>();
        passwords.put("myactaskey", "aspass");
        passwords.put("alice", "clarinet");
        passwords.put("bob", "trombone");
        return passwords;
    }
}
```

Web service requester

This section examines the ws-requester elements from the basic WS-Trust scenario that have been changed to address the needs of the OnBehalfOf example. The component is

- OnBehalfOf web service requester implementation class

Web service requester Implementation

The OnBehalfOf ws-requester, the client, uses standard procedures for creating a reference to the web service in the first four lines. To address the endpoint security requirements, the web service's "Request Context" is configured via the BindingProvider. Information needed in the message generation is provided through it. The OnBehalfOf user, alice, is declared in this section and the callbackHandler, UsernameTokenCallbackHandler is provided to the STSClient for generation of the contents for the OnBehalfOf message element. In this example a STSClient object is created and provided to the proxy's request context. The alternative is to provide keys tagged with the ".it" suffix as was done in [the Basic Scenario client](#). The use of OnBehalfOf is configured by the method call stsClient.setOnBehalfOf. The alternative is to use the key SecurityConstants.STS_TOKEN_ON_BEHALF_OF and a value in the props map.



```
final QName serviceName = new
QName("http://www.jboss.org/jbossws/ws-extensions/onbehalfowsssecuritypolicy",
"OnBehalfOfService");
final URL wsdlURL = new URL(serviceURL + "?wsdl");
Service service = Service.create(wsdlURL, serviceName);
OnBehalfOfServiceIface proxy = (OnBehalfOfServiceIface)
service.getPort(OnBehalfOfServiceIface.class);

Bus bus = BusFactory.newInstance().createBus();
try {

    BusFactory.setThreadDefaultBus(bus);

    Map<String, Object> ctx = proxy.getRequestContext();

    ctx.put(SecurityConstants.CALLBACK_HANDLER, new ClientCallbackHandler());
    ctx.put(SecurityConstants.ENCRYPT_PROPERTIES,
        Thread.currentThread().getContextClassLoader().getResource(
            "META-INF/clientKeystore.properties"));
    ctx.put(SecurityConstants.ENCRYPT_USERNAME, "myactaskey");
    ctx.put(SecurityConstants.SIGNATURE_PROPERTIES,
        Thread.currentThread().getContextClassLoader().getResource(
            "META-INF/clientKeystore.properties"));
    ctx.put(SecurityConstants.SIGNATURE_USERNAME, "myclientkey");

    // user and password OnBehalfOf user
    // UsernameTokenCallbackHandler will extract this information when called
    ctx.put(SecurityConstants.USERNAME, "alice");
    ctx.put(SecurityConstants.PASSWORD, "clarinet");

    STSClient stsClient = new STSClient(bus);

    // Providing the STSClient the mechanism to create the claims contents for OnBehalfOf
    stsClient.setOnBehalfOf(new UsernameTokenCallbackHandler());

    Map<String, Object> props = stsClient.getProperties();
    props.put(SecurityConstants.CALLBACK_HANDLER, new ClientCallbackHandler());
    props.put(SecurityConstants.ENCRYPT_PROPERTIES,
        Thread.currentThread().getContextClassLoader().getResource(
            "META-INF/clientKeystore.properties"));
    props.put(SecurityConstants.ENCRYPT_USERNAME, "mystskey");
    props.put(SecurityConstants.STS_TOKEN_USERNAME, "myclientkey");
    props.put(SecurityConstants.STS_TOKEN_PROPERTIES,
        Thread.currentThread().getContextClassLoader().getResource(
            "META-INF/clientKeystore.properties"));
    props.put(SecurityConstants.STS_TOKEN_USE_CERT_FOR_KEYINFO, "true");

    ctx.put(SecurityConstants.STS_CLIENT, stsClient);

} finally {
    bus.shutdown(true);
}
proxy.sayHello();
```




SAML Bearer Assertion Scenario

- [SAML Bearer Assertion Scenario](#)
 - [Web service Provider](#)
 - [Web service provider WSDL](#)
 - [SSL configuration](#)
 - [Web service Interface](#)
 - [Web service Implementation](#)
 - [Crypto properties and keystore files](#)
 - [MANIFEST.MF](#)
 - [Bearer Security Token Service](#)
 - [Security Domain](#)
 - [STS's WSDL](#)
 - [STS's implementation class](#)
 - [STSBearerCallbackHandler](#)
 - [Crypto properties and keystore files](#)
 - [MANIFEST.MF](#)
 - [Web service requester](#)
 - [Web service requester Implementation](#)
 - [ClientCallbackHandler](#)
 - [Crypto properties and keystore files](#)

SAML Bearer Assertion Scenario

WS-Trust deals with managing software security tokens. A SAML assertion is a type of security token. In the SAML Bearer scenario, the service provider automatically trusts that the incoming SOAP request came from the subject defined in the SAML token after the service verifies the tokens signature.

Implementation of this scenario has the following requirements.

- SAML tokens with a Bearer subject confirmation method must be protected so the token can not be snooped. In most cases, a bearer token combined with HTTPS is sufficient to prevent "a man in the middle" getting possession of the token. This means a security policy that uses a `sp:TransportBinding` and `sp:HttpsToken`.
- A bearer token has no encryption or signing keys associated with it, therefore a `sp:IssuedToken` of bearer keyType should be used with a `sp:SupportingToken` or a `sp:SignedSupportingTokens`.

Web service Provider

This section examines the web service elements for the SAML Bearer scenario. The components are

- Bearer web service provider's WSDL
- SSL configuration
- Bearer web service provider's Interface and Implementation classes.
- Crypto properties and keystore files
- MANIFEST.MF



Web service provider WSDL

The web service provider is a contract-first endpoint. All the WS-trust and security policies for it are declared in WSDL, `BearerService.wsdl`. For this scenario a ws-requester is required to present a SAML 2.0 Bearer token issued from a designed STS. The address of the STS is provided in the WSDL. HTTPS, a `TransportBinding` and `HttpsToken` policy are used to protect the SOAP body of messages that pass back and forth between ws-requester and ws-provider. A detailed explanation of the security settings are provided in the comments in the listing below.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<definitions targetNamespace="http://www.jboss.org/jbossws/ws-extensions/bearerwssecuritypolicy"
    name="BearerService"
    xmlns:tns="http://www.jboss.org/jbossws/ws-extensions/bearerwssecuritypolicy"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:wsp="http://www.w3.org/ns/ws-policy"
    xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata"

    xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
    xmlns:wsaws="http://www.w3.org/2005/08/addressing"
    xmlns:wsx="http://schemas.xmlsoap.org/ws/2004/09/mex"
    xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702"
    xmlns:t="http://docs.oasis-open.org/ws-sx/ws-trust/200512">

    <types>
        <xsd:schema>
            <xsd:import namespace="http://www.jboss.org/jbossws/ws-extensions/bearerwssecuritypolicy"
                schemaLocation="BearerService_schema1.xsd" />
        </xsd:schema>
    </types>
    <message name="sayHello">
        <part name="parameters" element="tns:sayHello" />
    </message>
    <message name="sayHelloResponse">
        <part name="parameters" element="tns:sayHelloResponse" />
    </message>
    <portType name="BearerIface">
        <operation name="sayHello">
            <input message="tns:sayHello" />
            <output message="tns:sayHelloResponse" />
        </operation>
    </portType>

    <!--
        The wsp:PolicyReference binds the security requirements on all the endpoints.
        The wsp:Policy wsu:Id="#TransportSAML2BearerPolicy" element is defined later in this
    file.
    -->
    <binding name="BearerServicePortBinding" type="tns:BearerIface">
        <wsp:PolicyReference URI="#TransportSAML2BearerPolicy" />
        <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document" />
        <operation name="sayHello">
            <soap:operation soapAction="" />
            <input>
```



```
        <soap:body use="literal"/>
    </input>
    <output>
        <soap:body use="literal"/>
    </output>
</operation>
</binding>

<!--
    The soap:address has been defined to use JBoss's https port, 8443. This is
    set in conjunction with the sp:TransportBinding policy for https.
-->
    <service name="BearerService">
        <port name="BearerServicePort" binding="tns:BearerServicePortBinding">
            <soap:address
location="https://@jboss.bind.address@:8443/jaxws-samples-wsse-policy-trust-bearer/BearerService"/
</port>
        </service>

    <wsp:Policy wsu:Id="TransportSAML2BearerPolicy">
        <wsp:ExactlyOne>
            <wsp:All>
                <!--
                    The wsam:Addressing element, indicates that the endpoints of this
                    web service MUST conform to the WS-Addressing specification. The
                    attribute wsp:Optional="false" enforces this assertion.
                -->
                <wsam:Addressing wsp:Optional="false">
                    <wsp:Policy />
                </wsam:Addressing>

            <!--
                The sp:TransportBinding element indicates that security is provided by the
                message exchange transport medium, https. WS-Security policy specification
                defines the sp:HttpsToken for use in exchanging messages transmitted over HTTPS.
            -->
            <sp:TransportBinding
                xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
                <wsp:Policy>
                    <sp:TransportToken>
                        <wsp:Policy>
                            <sp:HttpsToken>
                                <wsp:Policy/>
                            </sp:HttpsToken>
                        </wsp:Policy>
                    </sp:TransportToken>

            <!--
                The sp:AlgorithmSuite element, requires the TripleDes algorithm suite
                be used in performing cryptographic operations.
            -->
            <sp:AlgorithmSuite>
                <wsp:Policy>
                    <sp:TripleDes />
                </wsp:Policy>
            </sp:AlgorithmSuite>

            <!--
                The sp:Layout element, indicates the layout rules to apply when adding
```



```
items to the security header. The sp:Lax sub-element indicates items
are added to the security header in any order that conforms to
WSS: SOAP Message Security.
-->
    <sp:Layout>
      <wsp:Policy>
        <sp:Lax />
      </wsp:Policy>
    </sp:Layout>
    <sp:IncludeTimestamp />
  </wsp:Policy>
</sp:TransportBinding>

<!--
The sp:SignedSupportingTokens element causes the supporting tokens
to be signed using the primary token that is used to sign the message.
-->
    <sp:SignedSupportingTokens
      xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
      <wsp:Policy>

<!--
The sp:IssuedToken element asserts that a SAML 2.0 security token of type
Bearer is expected from the STS. The

sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/IncludeToken/AlwaysToRe
attribute instructs the runtime to include the initiator's public key
with every message sent to the recipient.

The sp:RequestSecurityTokenTemplate element directs that all of the
children of this element will be copied directly into the body of the
RequestSecurityToken (RST) message that is sent to the STS when the
initiator asks the STS to issue a token.
-->
    <sp:IssuedToken

sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/IncludeToken/AlwaysToRe
<sp:RequestSecurityTokenTemplate>

<t:TokenType>http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.1#SAMLV2.0</t:TokenType>
<t:KeyType>http://docs.oasis-open.org/ws-sx/ws-trust/200512/Bearer</t:KeyType>
  </sp:RequestSecurityTokenTemplate>
  <wsp:Policy>
    <sp:RequireInternalReference />
  </wsp:Policy>

<!--
The sp:Issuer element defines the STS's address and endpoint information
This information is used by the STSClient.
-->
    <sp:Issuer>

<wsaws:Address>http://@jboss.bind.address@:8080/jaxws-samples-wsse-policy-trust-sts-bearer/Security
<wsaws:Metadata
      xmlns:wsdli="http://www.w3.org/2006/01/wsdli-instance"

wsdli:wsdliLocation="http://@jboss.bind.address@:8080/jaxws-samples-wsse-policy-trust-sts-bearer/Se
<wsaw:ServiceName
      xmlns:wsaw="http://www.w3.org/2006/05/addressing/wsdli"
      xmlns:stsns="http://docs.oasis-open.org/ws-sx/ws-trust/200512/"
```



```
        EndpointName="UT_Port">stsns:SecurityTokenService</wsaw:ServiceName>
    </wsaws:Metadata>
    </sp:Issuer>

    </sp:IssuedToken>
  </wsp:Policy>
</sp:SignedSupportingTokens>
<!--
  The sp:Wss11 element declares WSS: SOAP Message Security 1.1 options
  to be supported by the STS.  These particular elements generally refer
  to how keys are referenced within the SOAP envelope.  These are normally
  handled by CXF.
-->
  <sp:Wss11>
    <wsp:Policy>
      <sp:MustSupportRefIssuerSerial />
      <sp:MustSupportRefThumbprint />
      <sp:MustSupportRefEncryptedKey />
    </wsp:Policy>
  </sp:Wss11>
<!--
  The sp:Trust13 element declares controls for WS-Trust 1.3 options.
  They are policy assertions related to exchanges specifically with
  client and server challenges and entropy behaviors.  Again these are
  normally handled by CXF.
-->
  <sp:Trust13>
    <wsp:Policy>
      <sp:MustSupportIssuedTokens />
      <sp:RequireClientEntropy />
      <sp:RequireServerEntropy />
    </wsp:Policy>
  </sp:Trust13>
</wsp:All>
</wsp:ExactlyOne>
</wsp:Policy>

</definitions>
```



SSL configuration

This web service is using https, therefore the JBoss server must be configured to provide SSL support in the Web subsystem. There are 2 components to SSL configuration.

- create a certificate keystore
- declare an SSL connector in the Web subsystem of the JBoss server configuration file.

Follow the directions in the, "*Using the pure Java implementation supplied by JSSE*" section in the [SSL Setup Guide](#).

Here is an example of an SSL connector declaration.

```
<subsystem xmlns="urn:jboss:domain:web:1.4" default-virtual-server="default-host"
native="false">
  ....
  <connector name="jbws-https-connector" protocol="HTTP/1.1" scheme="https"
socket-binding="https" secure="true" enabled="true">
    <ssl key-alias="tomcat" password="changeit"
certificate-key-file="/myJbossHome/security/test.keystore" verify-client="false"/>
  </connector>
  ...
```

Web service Interface

The web service provider interface class, `BearerIface`, is a simple straight forward web service definition.

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.bearer;

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

@WebService
(
    targetNamespace = "http://www.jboss.org/jbossws/ws-extensions/bearerwssecuritypolicy"
)
public interface BearerIface
{
    @WebMethod
    String sayHello();
}
```



Web service Implementation

The web service provider implementation class, `BearerImpl`, is a simple POJO. It uses the standard `WebService` annotation to define the service endpoint. In addition there are two Apache CXF annotations, `EndpointProperties` and `EndpointProperty` used for configuring the endpoint for the CXF runtime. These annotations come from the [Apache WSS4J project](#), which provides a Java implementation of the primary WS-Security standards for Web Services. These annotations are programmatically adding properties to the endpoint. With plain Apache CXF, these properties are often set via the `<jaxws:properties>` element on the `<jaxws:endpoint>` element in the Spring config; these annotations allow the properties to be configured in the code.

WSS4J uses the `Crypto` interface to get keys and certificates for signature creation/verification, as is asserted by the WSDL for this service. The WSS4J configuration information being provided by `BearerImpl` is for `Crypto`'s Merlin implementation. More information will be provided about this in the keystore section.

Because the web service provider automatically trusts that the incoming SOAP request came from the subject defined in the SAML token there is no need for a `Crypto` callbackHandler class or a signature username, unlike in prior examples, however in order to verify the message signature, the Java properties file that contains the (Merlin) `crypto` configuration information is still required.

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.bearer;

import org.apache.cxf.annotations.EndpointProperties;
import org.apache.cxf.annotations.EndpointProperty;

import javax.jws.WebService;

@WebService
(
    portName = "BearerServicePort",
    serviceName = "BearerService",
    wsdlLocation = "WEB-INF/wsdl/BearerService.wsdl",
    targetNamespace = "http://www.jboss.org/jbossws/ws-extensions/bearerwssecuritypolicy",
    endpointInterface = "org.jboss.test.ws.jaxws.samples.wsse.policy.trust.bearer.BearerIface"
)
@EndpointProperties(value = {
    @EndpointProperty(key = "ws-security.signature.properties", value =
"serviceKeystore.properties")
})
public class BearerImpl implements BearerIface
{
    public String sayHello()
    {
        return "Bearer WS-Trust Hello World!";
    }
}
```



Crypto properties and keystore files

WSS4J's Crypto implementation is loaded and configured via a Java properties file that contains Crypto configuration data. The file contains implementation-specific properties such as a keystore location, password, default alias and the like. This application is using the Merlin implementation. File `serviceKeystore.properties` contains this information.

File `servicestore.jks`, is a Java KeyStore (JKS) repository. It contains self signed certificates for `myservicekey` and `mystskey`. *Self signed certificates are not appropriate for production use.*

```
org.apache.ws.security.crypto.provider=org.apache.ws.security.components.crypto.Merlin
org.apache.ws.security.crypto.merlin.keystore.type=jks
org.apache.ws.security.crypto.merlin.keystore.password=sspass
org.apache.ws.security.crypto.merlin.keystore.alias=myservicekey
org.apache.ws.security.crypto.merlin.keystore.file=servicestore.jks
```

MANIFEST.MF

When deployed on WildFly this application requires access to the JBossWs and CXF APIs provided in module `org.jboss.ws.cxf.jbossws-cxf-client`. The dependency statement directs the server to provide them at deployment.

```
Manifest-Version: 1.0
Ant-Version: Apache Ant 1.8.2
Created-By: 1.7.0_25-b15 (Oracle Corporation)
Dependencies: org.jboss.ws.cxf.jbossws-cxf-client
```

Bearer Security Token Service

This section examines the crucial elements in providing the Security Token Service functionality for providing a SAML Bearer token. The components that will be discussed are.

- Security Domain
- STS's WSDL
- STS's implementation class
- STSBearerCallbackHandler
- Crypto properties and keystore files
- MANIFEST.MF



Security Domain

The STS requires a JBoss security domain be configured. The `jboss-web.xml` descriptor declares a named security domain, "JBossWS-trust-sts" to be used by this service for authentication. This security domain requires two properties files and the addition of a security-domain declaration in the JBoss server configuration file.

For this scenario the domain needs to contain user *alice*, password *clarinet*, and role *friend*. See the listings below for `jbossws-users.properties` and `jbossws-roles.properties`. In addition the following XML must be added to the JBoss security subsystem in the server configuration file. Replace "**SOME_PATH**" with appropriate information.

```
<security-domain name="JBossWS-trust-sts">
  <authentication>
    <login-module code="UsersRoles" flag="required">
      <module-option name="usersProperties" value="/SOME_PATH/jbossws-users.properties"/>
      <module-option name="unauthenticatedIdentity" value="anonymous"/>
      <module-option name="rolesProperties" value="/SOME_PATH/jbossws-roles.properties"/>
    </login-module>
  </authentication>
</security-domain>
```

jboss-web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jboss-web PUBLIC "-//JBoss//DTD Web Application 2.4//EN" "">
<jboss-web>
  <security-domain>java:/jaas/JBossWS-trust-sts</security-domain>
</jboss-web>
```

jbossws-users.properties

```
# A sample users.properties file for use with the UsersRolesLoginModule
alice=clarinet
```

jbossws-roles.properties

```
# A sample roles.properties file for use with the UsersRolesLoginModule
alice=friend
```

STS's WSDL

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions
  targetNamespace="http://docs.oasis-open.org/ws-sx/ws-trust/200512/"
  xmlns:tns="http://docs.oasis-open.org/ws-sx/ws-trust/200512/"
  xmlns:wstrust="http://docs.oasis-open.org/ws-sx/ws-trust/200512/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/">
```



```
xmlns:wsap10="http://www.w3.org/2006/05/addressing/wsd1"
xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
xmlns:wsp="http://www.w3.org/ns/ws-policy"
xmlns:wst="http://docs.oasis-open.org/ws-sx/ws-trust/200512"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata">

<wsdl:types>
  <xs:schema elementFormDefault="qualified"
    targetNamespace='http://docs.oasis-open.org/ws-sx/ws-trust/200512'>

    <xs:element name='RequestSecurityToken'
      type='wst:AbstractRequestSecurityTokenType' />
    <xs:element name='RequestSecurityTokenResponse'
      type='wst:AbstractRequestSecurityTokenType' />

    <xs:complexType name='AbstractRequestSecurityTokenType'>
      <xs:sequence>
        <xs:any namespace='##any' processContents='lax' minOccurs='0'
          maxOccurs='unbounded' />
      </xs:sequence>
      <xs:attribute name='Context' type='xs:anyURI' use='optional' />
      <xs:anyAttribute namespace='##other' processContents='lax' />
    </xs:complexType>
    <xs:element name='RequestSecurityTokenCollection'
      type='wst:RequestSecurityTokenCollectionType' />
    <xs:complexType name='RequestSecurityTokenCollectionType'>
      <xs:sequence>
        <xs:element name='RequestSecurityToken'
          type='wst:AbstractRequestSecurityTokenType' minOccurs='2'
          maxOccurs='unbounded' />
      </xs:sequence>
    </xs:complexType>

    <xs:element name='RequestSecurityTokenResponseCollection'
      type='wst:RequestSecurityTokenResponseCollectionType' />
    <xs:complexType name='RequestSecurityTokenResponseCollectionType'>
      <xs:sequence>
        <xs:element ref='wst:RequestSecurityTokenResponse' minOccurs='1'
          maxOccurs='unbounded' />
      </xs:sequence>
      <xs:anyAttribute namespace='##other' processContents='lax' />
    </xs:complexType>

  </xs:schema>
</wsdl:types>

<!-- WS-Trust defines the following GEDs -->
<wsdl:message name="RequestSecurityTokenMsg">
  <wsdl:part name="request" element="wst:RequestSecurityToken"/>
</wsdl:message>
<wsdl:message name="RequestSecurityTokenResponseMsg">
  <wsdl:part name="response"
    element="wst:RequestSecurityTokenResponse"/>
</wsdl:message>
<wsdl:message name="RequestSecurityTokenCollectionMsg">
  <wsdl:part name="requestCollection"
    element="wst:RequestSecurityTokenCollection"/>
</wsdl:message>
```



```
</wsdl:message>
<wsdl:message name="RequestSecurityTokenResponseCollectionMsg">
  <wsdl:part name="responseCollection"
    element="wst:RequestSecurityTokenResponseCollection"/>
</wsdl:message>

<!-- This portType an example of a Requestor (or other) endpoint that
Accepts SOAP-based challenges from a Security Token Service -->
<wsdl:portType name="WSSecurityRequestor">
  <wsdl:operation name="Challenge">
    <wsdl:input message="tns:RequestSecurityTokenResponseMsg"/>
    <wsdl:output message="tns:RequestSecurityTokenResponseMsg"/>
  </wsdl:operation>
</wsdl:portType>

<!-- This portType is an example of an STS supporting full protocol -->
<!--
  The wsdl:portType and data types are XML elements defined by the
  WS_Trust specification. The wsdl:portType defines the endpoints
  supported in the STS implementation. This WSDL defines all operations
  that an STS implementation can support.
-->
<wsdl:portType name="STS">
  <wsdl:operation name="Cancel">
    <wsdl:input
      wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Cancel"
      message="tns:RequestSecurityTokenMsg"/>
    <wsdl:output
      wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/CancelFinal"
      message="tns:RequestSecurityTokenResponseMsg"/>
  </wsdl:operation>
  <wsdl:operation name="Issue">
    <wsdl:input
      wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Issue"
      message="tns:RequestSecurityTokenMsg"/>
    <wsdl:output
      wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTRC/IssueFinal"
      message="tns:RequestSecurityTokenResponseCollectionMsg"/>
  </wsdl:operation>
  <wsdl:operation name="Renew">
    <wsdl:input
      wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Renew"
      message="tns:RequestSecurityTokenMsg"/>
    <wsdl:output
      wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/RenewFinal"
      message="tns:RequestSecurityTokenResponseMsg"/>
  </wsdl:operation>
  <wsdl:operation name="Validate">
    <wsdl:input
      wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Validate"
      message="tns:RequestSecurityTokenMsg"/>
    <wsdl:output
      wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/ValidateFinal"
      message="tns:RequestSecurityTokenResponseMsg"/>
  </wsdl:operation>
  <wsdl:operation name="KeyExchangeToken">
    <wsdl:input
      wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/KET"
```



```
        message="tns:RequestSecurityTokenMsg" />
    <wsdl:output
        wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/KETFinal"
        message="tns:RequestSecurityTokenResponseMsg" />
    </wsdl:operation>
    <wsdl:operation name="RequestCollection">
        <wsdl:input message="tns:RequestSecurityTokenCollectionMsg" />
        <wsdl:output message="tns:RequestSecurityTokenResponseCollectionMsg" />
    </wsdl:operation>
</wsdl:portType>

<!-- This portType is an example of an endpoint that accepts
Unsolicited RequestSecurityTokenResponse messages -->
<wsdl:portType name="SecurityTokenResponseService">
    <wsdl:operation name="RequestSecurityTokenResponse">
        <wsdl:input message="tns:RequestSecurityTokenResponseMsg" />
    </wsdl:operation>
</wsdl:portType>

<!--
    The wsp:PolicyReference binds the security requirements on all the STS endpoints.
    The wsp:Policy wsu:Id="UT_policy" element is later in this file.
-->
<wsdl:binding name="UT_Binding" type="wstrust:STS">
    <wsp:PolicyReference URI="#UT_policy" />
    <soap:binding style="document"
        transport="http://schemas.xmlsoap.org/soap/http" />
    <wsdl:operation name="Issue">
        <soap:operation
            soapAction="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Issue" />
        <wsdl:input>
            <wsp:PolicyReference
                URI="#Input_policy" />
            <soap:body use="literal" />
        </wsdl:input>
        <wsdl:output>
            <wsp:PolicyReference
                URI="#Output_policy" />
            <soap:body use="literal" />
        </wsdl:output>
    </wsdl:operation>
    <wsdl:operation name="Validate">
        <soap:operation
            soapAction="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Validate" />
        <wsdl:input>
            <wsp:PolicyReference
                URI="#Input_policy" />
            <soap:body use="literal" />
        </wsdl:input>
        <wsdl:output>
            <wsp:PolicyReference
                URI="#Output_policy" />
            <soap:body use="literal" />
        </wsdl:output>
    </wsdl:operation>
    <wsdl:operation name="Cancel">
        <soap:operation
            soapAction="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Cancel" />
```



```
<wsdl:input>
  <soap:body use="literal"/>
</wsdl:input>
<wsdl:output>
  <soap:body use="literal"/>
</wsdl:output>
</wsdl:operation>
<wsdl:operation name="Renew">
  <soap:operation
    soapAction="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Renew"/>
  <wsdl:input>
    <soap:body use="literal"/>
  </wsdl:input>
  <wsdl:output>
    <soap:body use="literal"/>
  </wsdl:output>
</wsdl:operation>
<wsdl:operation name="KeyExchangeToken">
  <soap:operation
    soapAction="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/KeyExchangeToken"/>
  <wsdl:input>
    <soap:body use="literal"/>
  </wsdl:input>
  <wsdl:output>
    <soap:body use="literal"/>
  </wsdl:output>
</wsdl:operation>
<wsdl:operation name="RequestCollection">
  <soap:operation
    soapAction="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/RequestCollection"/>
  <wsdl:input>
    <soap:body use="literal"/>
  </wsdl:input>
  <wsdl:output>
    <soap:body use="literal"/>
  </wsdl:output>
</wsdl:operation>
</wsdl:binding>

<wsdl:service name="SecurityTokenService">
  <wsdl:port name="UT_Port" binding="tns:UT_Binding">
    <soap:address location="http://localhost:8080/SecurityTokenService/UT"/>
  </wsdl:port>
</wsdl:service>

<wsp:Policy wsu:Id="UT_policy">
  <wsp:ExactlyOne>
    <wsp:All>
      <!--
        The sp:UsingAddressing element, indicates that the endpoints of this
        web service conforms to the WS-Addressing specification. More detail
        can be found here: [http://www.w3.org/TR/2006/CR-ws-addr-wsdl-20060529]
      -->
      <wsap10:UsingAddressing/>
      <!--
        The sp:SymmetricBinding element indicates that security is provided
        at the SOAP layer and any initiator must authenticate itself by providing
```



```
WSS UsernameToken credentials.
-->
<sp:SymmetricBinding
  xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
  <wsp:Policy>
    <!--
      In a symmetric binding, the keys used for encrypting and signing in both
      directions are derived from a single key, the one specified by the
      sp:ProtectionToken element. The sp:X509Token sub-element declares this
      key to be a X.509 certificate and the

IncludeToken="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/IncludeToken/Never"
      attribute adds the requirement that the token MUST NOT be included in
      any messages sent between the initiator and the recipient; rather, an
      external reference to the token should be used. Lastly the WssX509V3Token10
      sub-element declares that the Username token presented by the initiator
      should be compliant with Web Services Security UsernameToken Profile
      1.0 specification. [
http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0.pdf ]
      -->
    <sp:ProtectionToken>
      <wsp:Policy>
        <sp:X509Token

sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/IncludeToken/Never">
      <wsp:Policy>
        <sp:RequireDerivedKeys/>
        <sp:RequireThumbprintReference/>
        <sp:WssX509V3Token10/>
      </wsp:Policy>
    </sp:X509Token>
  </wsp:Policy>
</sp:ProtectionToken>
<!--
  The sp:AlgorithmSuite element, requires the Basic256 algorithm suite
  be used in performing cryptographic operations.
-->
<sp:AlgorithmSuite>
  <wsp:Policy>
    <sp:Basic256/>
  </wsp:Policy>
</sp:AlgorithmSuite>
<!--
  The sp:Layout element, indicates the layout rules to apply when adding
  items to the security header. The sp:Lax sub-element indicates items
  are added to the security header in any order that conforms to
  WSS: SOAP Message Security.
-->
<sp:Layout>
  <wsp:Policy>
    <sp:Lax/>
  </wsp:Policy>
</sp:Layout>
<sp:IncludeTimestamp/>
<sp:EncryptSignature/>
<sp:OnlySignEntireHeadersAndBody/>
  </wsp:Policy>
</sp:SymmetricBinding>
```



```
<!--
  The sp:SignedSupportingTokens element declares that the security header
  of messages must contain a sp:UsernameToken and the token must be signed.
  The attribute
  IncludeToken="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/IncludeToken/AlwaysToRecip
  on sp:UsernameToken indicates that the token MUST be included in all
  messages sent from initiator to the recipient and that the token MUST
  NOT be included in messages sent from the recipient to the initiator.
  And finally the element sp:WssUsernameToken10 is a policy assertion
  indicating the Username token should be as defined in Web Services
  Security UsernameToken Profile 1.0
-->
<sp:SignedSupportingTokens
  xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
  <wsp:Policy>
    <sp:UsernameToken

sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/IncludeToken/AlwaysToRe
<wsp:Policy>
    <sp:WssUsernameToken10/>
    </wsp:Policy>
  </sp:UsernameToken>
</wsp:Policy>
</sp:SignedSupportingTokens>
<!--
  The sp:Wss11 element declares WSS: SOAP Message Security 1.1 options
  to be supported by the STS. These particular elements generally refer
  to how keys are referenced within the SOAP envelope. These are normally
  handled by CXF.
-->
<sp:Wss11
  xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
  <wsp:Policy>
    <sp:MustSupportRefKeyIdentifier/>
    <sp:MustSupportRefIssuerSerial/>
    <sp:MustSupportRefThumbprint/>
    <sp:MustSupportRefEncryptedKey/>
  </wsp:Policy>
</sp:Wss11>
<!--
  The sp:Trust13 element declares controls for WS-Trust 1.3 options.
  They are policy assertions related to exchanges specifically with
  client and server challenges and entropy behaviors. Again these are
  normally handled by CXF.
-->
<sp:Trust13
  xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
  <wsp:Policy>
    <sp:MustSupportIssuedTokens/>
    <sp:RequireClientEntropy/>
    <sp:RequireServerEntropy/>
  </wsp:Policy>
</sp:Trust13>
</wsp:All>
</wsp:ExactlyOne>
</wsp:Policy>
```



```
<wsp:Policy wsu:Id="Input_policy">
  <wsp:ExactlyOne>
    <wsp:All>
      <sp:SignedParts
        xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
        <sp:Body/>
        <sp:Header Name="To"
          Namespace="http://www.w3.org/2005/08/addressing"/>
        <sp:Header Name="From"
          Namespace="http://www.w3.org/2005/08/addressing"/>
        <sp:Header Name="FaultTo"
          Namespace="http://www.w3.org/2005/08/addressing"/>
        <sp:Header Name="ReplyTo"
          Namespace="http://www.w3.org/2005/08/addressing"/>
        <sp:Header Name="MessageID"
          Namespace="http://www.w3.org/2005/08/addressing"/>
        <sp:Header Name="RelatesTo"
          Namespace="http://www.w3.org/2005/08/addressing"/>
        <sp:Header Name="Action"
          Namespace="http://www.w3.org/2005/08/addressing"/>
      </sp:SignedParts>
    </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>

<wsp:Policy wsu:Id="Output_policy">
  <wsp:ExactlyOne>
    <wsp:All>
      <sp:SignedParts
        xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
        <sp:Body/>
        <sp:Header Name="To"
          Namespace="http://www.w3.org/2005/08/addressing"/>
        <sp:Header Name="From"
          Namespace="http://www.w3.org/2005/08/addressing"/>
        <sp:Header Name="FaultTo"
          Namespace="http://www.w3.org/2005/08/addressing"/>
        <sp:Header Name="ReplyTo"
          Namespace="http://www.w3.org/2005/08/addressing"/>
        <sp:Header Name="MessageID"
          Namespace="http://www.w3.org/2005/08/addressing"/>
        <sp:Header Name="RelatesTo"
          Namespace="http://www.w3.org/2005/08/addressing"/>
        <sp:Header Name="Action"
          Namespace="http://www.w3.org/2005/08/addressing"/>
      </sp:SignedParts>
    </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>

</wsdl:definitions>
```

STS's implementation class



The Apache CXF's STS, `SecurityTokenServiceProvider`, is a web service provider that is compliant with the protocols and functionality defined by the WS-Trust specification. It has a modular architecture. Many of its components are configurable or replaceable and there are many optional features that are enabled by implementing and configuring plug-ins. Users can customize their own STS by extending from `SecurityTokenServiceProvider` and overriding the default settings. Extensive information about the CXF's STS configurable and pluggable components can be found [here](#).

This STS implementation class, `SampleSTSBearer`, is a POJO that extends from `SecurityTokenServiceProvider`. Note that the class is defined with a `WebServiceProvider` annotation and not a `WebService` annotation. This annotation defines the service as a Provider-based endpoint, meaning it supports a more messaging-oriented approach to Web services. In particular, it signals that the exchanged messages will be XML documents of some type. `SecurityTokenServiceProvider` is an implementation of the `javax.xml.ws.Provider` interface. In comparison the `WebService` annotation defines a (service endpoint interface) SEI-based endpoint which supports message exchange via SOAP envelopes.

As was done in the `BearerImpl` class, the WSS4J annotations `EndpointProperties` and `EndpointProperty` are providing endpoint configuration for the CXF runtime. The first `EndpointProperty` statement in the listing is declaring the user's name to use for the message signature. It is used as the alias name in the keystore to get the user's cert and private key for signature. The next two `EndpointProperty` statements declares the Java properties file that contains the (Merlin) crypto configuration information. In this case both for signing and encrypting the messages. WSS4J reads this file and extra required information for message handling. The last `EndpointProperty` statement declares the `STSBearerCallbackHandler` implementation class. It is used to obtain the user's password for the certificates in the keystore file.

In this implementation we are customizing the operations of token issuance, token validation and their static properties.

`StaticSTSProperties` is used to set select properties for configuring resources in the STS. You may think this is a duplication of the settings made with the WSS4J annotations. The values are the same but the underlying structures being set are different, thus this information must be declared in both places.

The `setIssuer` setting is important because it uniquely identifies the issuing STS. The issuer string is embedded in issued tokens and, when validating tokens, the STS checks the issuer string value. Consequently, it is important to use the issuer string in a consistent way, so that the STS can recognize the tokens that it has issued.

The `setEndpoints` call allows the declaration of a set of allowed token recipients by address. The addresses are specified as reg-ex patterns.

`TokenIssueOperation` has a modular structure. This allows custom behaviors to be injected into the processing of messages. In this case we are overriding the `SecurityTokenServiceProvider`'s default behavior and performing SAML token processing. CXF provides an implementation of a `SAMLTokenProvider` which we are using rather than writing our own.

Learn more about the `SAMLTokenProvider` [here](#).

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.stsbearer;

import org.apache.cxf.annotations.EndpointProperties;
```



```
import org.apache.cxf.annotations.EndpointProperty;
import org.apache.cxf.sts.StaticSTSProperties;
import org.apache.cxf.sts.operation.TokenIssueOperation;
import org.apache.cxf.sts.service.ServiceMBean;
import org.apache.cxf.sts.service.StaticService;
import org.apache.cxf.sts.token.provider.SAMLTokenProvider;
import org.apache.cxf.ws.security.sts.provider.SecurityTokenServiceProvider;

import javax.xml.ws.WebServiceProvider;
import java.util.Arrays;
import java.util.LinkedList;
import java.util.List;

@WebServiceProvider(serviceName = "SecurityTokenService",
    portName = "UT_Port",
    targetNamespace = "http://docs.oasis-open.org/ws-sx/ws-trust/200512/",
    wsdlLocation = "WEB-INF/wsdl/bearer-ws-trust-1.4-service.wsdl")
//be sure to have dependency on org.apache.cxf module when on AS7, otherwise Apache CXF
annotations are ignored
@EndpointProperties(value = {
    @EndpointProperty(key = "ws-security.signature.username", value = "mystskey"),
    @EndpointProperty(key = "ws-security.signature.properties", value =
"stsKeystore.properties"),
    @EndpointProperty(key = "ws-security.callback-handler", value =
"org.jboss.test.ws.jaxws.samples.wsse.policy.trust.stsbearer.STSBearerCallbackHandler")
})
public class SampleSTSBearer extends SecurityTokenServiceProvider
{

    public SampleSTSBearer() throws Exception
    {
        super();

        StaticSTSProperties props = new StaticSTSProperties();
        props.setSignatureCryptoProperties("stsKeystore.properties");
        props.setSignatureUsername("mystskey");
        props.setCallbackHandlerClass(STSBearerCallbackHandler.class.getName());
        props.setEncryptionCryptoProperties("stsKeystore.properties");
        props.setEncryptionUsername("myservicekey");
        props.setIssuer("DoubleItSTSIssuer");

        List<ServiceMBean> services = new LinkedList<ServiceMBean>();
        StaticService service = new StaticService();
        service.setEndpoints(Arrays.asList(
            "https://localhost:(\\d)*/jaxws-samples-wsse-policy-trust-bearer/BearerService",
            "https://\\[[::1\\]:(\\d)*/jaxws-samples-wsse-policy-trust-bearer/BearerService",
"https://\\[[0:0:0:0:0:0:1\\]:(\\d)*/jaxws-samples-wsse-policy-trust-bearer/BearerService"
        ));
        services.add(service);

        TokenIssueOperation issueOperation = new TokenIssueOperation();
        issueOperation.getTokenProviders().add(new SAMLTokenProvider());
        issueOperation.setServices(services);
        issueOperation.setStsProperties(props);
        this.setIssueOperation(issueOperation);
    }
}
```



STSBearerCallbackHandler

STSBearerCallbackHandler is a callback handler for the WSS4J Crypto API. It is used to obtain the password for the private key in the keystore. This class enables CXF to retrieve the password of the user name to use for the message signature.

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.stsbearer;

import org.jboss.wsf.stack.cxf.extensions.security.PasswordCallbackHandler;

import java.util.HashMap;
import java.util.Map;

public class STSBearerCallbackHandler extends PasswordCallbackHandler
{
    public STSBearerCallbackHandler()
    {
        super(getInitMap());
    }

    private static Map<String, String> getInitMap()
    {
        Map<String, String> passwords = new HashMap<String, String>();
        passwords.put("mystskey", "stskpass");
        passwords.put("alice", "clarinet");
        return passwords;
    }
}
```

Crypto properties and keystore files

WSS4J's Crypto implementation is loaded and configured via a Java properties file that contains Crypto configuration data. The file contains implementation-specific properties such as a keystore location, password, default alias and the like. This application is using the Merlin implementation. File `stsKeystore.properties` contains this information.

File `servicestore.jks`, is a Java KeyStore (JKS) repository. It contains self signed certificates for `myservicekey` and `mystskey`. *Self signed certificates are not appropriate for production use.*

```
org.apache.ws.security.crypto.provider=org.apache.ws.security.components.crypto.Merlin
org.apache.ws.security.crypto.merlin.keystore.type=jks
org.apache.ws.security.crypto.merlin.keystore.password=stsspass
org.apache.ws.security.crypto.merlin.keystore.file=stsstore.jks
```



MANIFEST.MF

When deployed on WildFly, this application requires access to the JBossWs and CXF APIs provided in modules `org.jboss.ws.cxf.jbossws-cxf-client` and `org.apache.cxf`. The Apache CXF internals, `org.apache.cxf.impl`, are needed to build the STS configuration in the `SampleSTS` constructor. The dependency statement directs the server to provide them at deployment.

```
Manifest-Version: 1.0
Ant-Version: Apache Ant 1.8.2
Created-By: 1.7.0_25-b15 (Oracle Corporation)
Dependencies: org.jboss.ws.cxf.jbossws-cxf-client,org.apache.cxf.impl
```

Web service requester

This section examines the crucial elements in calling a web service that implements endpoint security as described in the SAML Bearer scenario. The components that will be discussed are.

- Web service requester's implementation
- `ClientCallbackHandler`
- Crypto properties and keystore files



Web service requester Implementation

The ws-requester, the client, uses standard procedures for creating a reference to the web service. To address the endpoint security requirements, the web service's "Request Context" is configured with the information needed in message generation. In addition, the STSClient that communicates with the STS is configured with similar values. Note the key strings ending with a ".it" suffix. This suffix flags these settings as belonging to the STSClient. The internal CXF code assigns this information to the STSClient that is auto-generated for this service call.

There is an alternate method of setting up the STSClient. The user may provide their own instance of the STSClient. The CXF code will use this object and not auto-generate one. When providing the STSClient in this way, the user must provide a `org.apache.cxf.Bus` for it and the configuration keys must not have the ".it" suffix. This is used in the `ActAs` and `OnBehalfOf` examples.

```
String serviceURL = "https://" + getServerHost() +
":8443/jaxws-samples-wsse-policy-trust-bearer/BearerService";

final QName serviceName = new
QName("http://www.jboss.org/jbossws/ws-extensions/bearerwssecuritypolicy", "BearerService");
Service service = Service.create(new URL(serviceURL + "?wsdl"), serviceName);
BearerIface proxy = (BearerIface) service.getPort(BearerIface.class);

Map<String, Object> ctx = ((BindingProvider)proxy).getRequestContext();

// set the security related configuration information for the service "request"
ctx.put(SecurityConstants.CALLBACK_HANDLER, new ClientCallbackHandler());
ctx.put(SecurityConstants.SIGNATURE_PROPERTIES,
Thread.currentThread().getContextClassLoader().getResource(
"META-INF/clientKeystore.properties"));
ctx.put(SecurityConstants.ENCRYPT_PROPERTIES,
Thread.currentThread().getContextClassLoader().getResource(
"META-INF/clientKeystore.properties"));
ctx.put(SecurityConstants.SIGNATURE_USERNAME, "myclientkey");
ctx.put(SecurityConstants.ENCRYPT_USERNAME, "myservicekey");

//-- Configuration settings that will be transfered to the STSClient
// "alice" is the name provided for the WSS Username. Her password will
// be retrieved from the ClientCallbackHander by the STSClient.
ctx.put(SecurityConstants.USERNAME + ".it", "alice");
ctx.put(SecurityConstants.CALLBACK_HANDLER + ".it", new ClientCallbackHandler());
ctx.put(SecurityConstants.ENCRYPT_PROPERTIES + ".it",
Thread.currentThread().getContextClassLoader().getResource(
"META-INF/clientKeystore.properties"));
ctx.put(SecurityConstants.ENCRYPT_USERNAME + ".it", "mystskey");
ctx.put(SecurityConstants.STS_TOKEN_USERNAME + ".it", "myclientkey");
ctx.put(SecurityConstants.STS_TOKEN_PROPERTIES + ".it",
Thread.currentThread().getContextClassLoader().getResource(
"META-INF/clientKeystore.properties"));
ctx.put(SecurityConstants.STS_TOKEN_USE_CERT_FOR_KEYINFO + ".it", "true");

proxy.sayHello();
```



ClientCallbackHandler

<https://docs.jboss.org/author/display/JBWS/WS-Trust+and+STS#WS-TrustandSTS-ClientCallbackHandler>

ClientCallbackHandler is a callback handler for the WSS4J Crypto API. It is used to obtain the password for the private key in the keystore. This class enables CXF to retrieve the password of the user name to use for the message signature. Note that "alice" and her password have been provided here. This information is not in the (JKS) keystore but provided in the WildFly security domain. It was declared in file `jbossws-users.properties`.

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.shared;

import java.io.IOException;
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.UnsupportedCallbackException;
import org.apache.ws.security.WSPasswordCallback;

public class ClientCallbackHandler implements CallbackHandler {

    public void handle(Callback[] callbacks) throws IOException,
        UnsupportedCallbackException {
        for (int i = 0; i < callbacks.length; i++) {
            if (callbacks[i] instanceof WSPasswordCallback) {
                WSPasswordCallback pc = (WSPasswordCallback) callbacks[i];
                if ("myclientkey".equals(pc.getIdentifier())) {
                    pc.setPassword("ckpass");
                    break;
                } else if ("alice".equals(pc.getIdentifier())) {
                    pc.setPassword("clarinet");
                    break;
                } else if ("bob".equals(pc.getIdentifier())) {
                    pc.setPassword("trombone");
                    break;
                } else if ("myservicekey".equals(pc.getIdentifier())) { // rls test added for
bearer test
                    pc.setPassword("skpass");
                    break;
                }
            }
        }
    }
}
```



Crypto properties and keystore files

<https://docs.jboss.org/author/display/JBWS/WS-Trust+and+STS#WS-TrustandSTS-RequesterCryptopropertie>

WSS4J's Crypto implementation is loaded and configured via a Java properties file that contains Crypto configuration data. The file contains implementation-specific properties such as a keystore location, password, default alias and the like. This application is using the Merlin implementation. File `clientKeystore.properties` contains this information.

File `clientstore.jks`, is a Java KeyStore (JKS) repository. It contains self signed certificates for `myservicekey` and `mystskey`. *Self signed certificates are not appropriate for production use.*

```
org.apache.ws.security.crypto.provider=org.apache.ws.security.components.crypto.Merlin
org.apache.ws.security.crypto.merlin.keystore.type=jks
org.apache.ws.security.crypto.merlin.keystore.password=cspass
org.apache.ws.security.crypto.merlin.keystore.alias=myclientkey
org.apache.ws.security.crypto.merlin.keystore.file=META-INF/clientstore.jks
```

SAML Holder-Of-Key Assertion Scenario

- [SAML Holder-Of-Key Assertion Scenario](#)
 - [Web service Provider](#)
 - [Web service provider WSDL](#)
 - [SSL configuration](#)
 - [Web service Interface](#)
 - [Web service Implementation](#)
 - [Crypto properties and keystore files](#)
 - [MANIFEST.MF](#)
 - [Security Token Service](#)
 - [Security Domain](#)
 - [STS's WSDL](#)
 - [STS's implementation class](#)
 - [HolderOfKeyCallbackHandler](#)
 - [Crypto properties and keystore files](#)
 - [MANIFEST.MF](#)
 - [Web service requester](#)
 - [Web service requester Implementation](#)
 - [ClientCallbackHandler](#)
 - [Crypto properties and keystore files](#)

SAML Holder-Of-Key Assertion Scenario

WS-Trust deals with managing software security tokens. A SAML assertion is a type of security token. In the Holder-Of-Key method, the STS creates a SAML token containing the client's public key and signs the SAML token with its private key. The client includes the SAML token and signs the outgoing soap envelope to the web service with its private key. The web service validates the SOAP message and the SAML token.



Implementation of this scenario has the following requirements.

- SAML tokens with a Holder-Of-Key subject confirmation method must be protected so the token can not be snooped. In most cases, a Holder-Of-Key token combined with HTTPS is sufficient to prevent "a man in the middle" getting possession of the token. This means a security policy that uses a `sp:TransportBinding` and `sp:HttpsToken`.
- A Holder-Of-Key token has no encryption or signing keys associated with it, therefore a `sp:IssuedToken` of `SymmetricKey` or `PublicKey` `keyType` should be used with a `sp:SignedEndorsingSupportingTokens`.

Web service Provider

This section examines the web service elements for the SAML Holder-Of-Key scenario. The components are

- Web service provider's WSDL
- SSL configuration
- Web service provider's Interface and Implementation classes.
- Crypto properties and keystore files
- MANIFEST.MF

Web service provider WSDL

The web service provider is a contract-first endpoint. All the WS-trust and security policies for it are declared in the WSDL, `HolderOfKeyService.wsdl`. For this scenario a ws-requester is required to present a SAML 2.0 token of `SymmetricKey` `keyType`, issued from a designed STS. The address of the STS is provided in the WSDL. A transport binding policy is used. The token is declared to be signed and endorsed, `sp:SignedEndorsingSupportingTokens`. A detailed explanation of the security settings are provided in the comments in the listing below.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<definitions
targetNamespace="http://www.jboss.org/jbossws/ws-extensions/holderofkeywssecuritypolicy"
    name="HolderOfKeyService"
    xmlns:tns="http://www.jboss.org/jbossws/ws-extensions/holderofkeywssecuritypolicy"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:wsp="http://www.w3.org/ns/ws-policy"
    xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata"

    xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
    xmlns:wsaws="http://www.w3.org/2005/08/addressing"
    xmlns:wsx="http://schemas.xmlsoap.org/ws/2004/09/mex"
    xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702"
    xmlns:t="http://docs.oasis-open.org/ws-sx/ws-trust/200512">

    <types>
    <xsd:schema>
    <xsd:import
namespace="http://www.jboss.org/jbossws/ws-extensions/holderofkeywssecuritypolicy"
        schemaLocation="HolderOfKeyService_schema1.xsd"/>
    </xsd:schema>
```




```
</types>
<message name="sayHello">
  <part name="parameters" element="tns:sayHello"/>
</message>
<message name="sayHelloResponse">
  <part name="parameters" element="tns:sayHelloResponse"/>
</message>
<portType name="HolderOfKeyIface">
  <operation name="sayHello">
    <input message="tns:sayHello"/>
    <output message="tns:sayHelloResponse"/>
  </operation>
</portType>
<!--
  The wsp:PolicyReference binds the security requirements on all the endpoints.
  The wsp:Policy wsu:Id="#TransportSAML2HolderOfKeyPolicy" element is defined later in
this file.
-->
<binding name="HolderOfKeyServicePortBinding" type="tns:HolderOfKeyIface">
  <wsp:PolicyReference URI="#TransportSAML2HolderOfKeyPolicy" />
  <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
  <operation name="sayHello">
    <soap:operation soapAction="" />
    <input>
      <soap:body use="literal"/>
    </input>
    <output>
      <soap:body use="literal"/>
    </output>
  </operation>
</binding>
<!--
  The soap:address has been defined to use JBoss's https port, 8443. This is
  set in conjunction with the sp:TransportBinding policy for https.
-->
<service name="HolderOfKeyService">
  <port name="HolderOfKeyServicePort" binding="tns:HolderOfKeyServicePortBinding">
    <soap:address
location="https://@jboss.bind.address@:8443/jaxws-samples-wsse-policy-trust-holderofkey/HolderOfKey
</port>
  </service>

  <wsp:Policy wsu:Id="TransportSAML2HolderOfKeyPolicy">
    <wsp:ExactlyOne>
      <wsp:All>
<!--
  The wsam:Addressing element, indicates that the endpoints of this
  web service MUST conform to the WS-Addressing specification. The
  attribute wsp:Optional="false" enforces this assertion.
-->
    <wsam:Addressing wsp:Optional="false">
      <wsp:Policy />
    </wsam:Addressing>
<!--
  The sp:TransportBinding element indicates that security is provided by the
  message exchange transport medium, https. WS-Security policy specification
  defines the sp:HttpsToken for use in exchanging messages transmitted over HTTPS.
```



```
-->
    <sp:TransportBinding
      xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
      <wsp:Policy>
        <sp:TransportToken>
          <wsp:Policy>
            <sp:HttpsToken>
              <wsp:Policy/>
            </sp:HttpsToken>
          </wsp:Policy>
        </sp:TransportToken>
      <!--
        The sp:AlgorithmSuite element, requires the TripleDes algorithm suite
        be used in performing cryptographic operations.
      -->
        <sp:AlgorithmSuite>
          <wsp:Policy>
            <sp:TripleDes />
          </wsp:Policy>
        </sp:AlgorithmSuite>
      <!--
        The sp:Layout element, indicates the layout rules to apply when adding
        items to the security header. The sp:Lax sub-element indicates items
        are added to the security header in any order that conforms to
        WSS: SOAP Message Security.
      -->
        <sp:Layout>
          <wsp:Policy>
            <sp:Lax />
          </wsp:Policy>
        </sp:Layout>
        <sp:IncludeTimestamp />
      </wsp:Policy>
    </sp:TransportBinding>

    <!--
      The sp:SignedEndorsingSupportingTokens, when transport level security level is
      used there will be no message signature and the signature generated by the
      supporting token will sign the Timestamp.
    -->
      <sp:SignedEndorsingSupportingTokens
        xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
        <wsp:Policy>
          <!--
            The sp:IssuedToken element asserts that a SAML 2.0 security token of type
            Bearer is expected from the STS. The
            sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/IncludeToken/AlwaysToRe
            attribute instructs the runtime to include the initiator's public key
            with every message sent to the recipient.

            The sp:RequestSecurityTokenTemplate element directs that all of the
            children of this element will be copied directly into the body of the
            RequestSecurityToken (RST) message that is sent to the STS when the
            initiator asks the STS to issue a token.
          -->
            <sp:IssuedToken
```



```
sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/IncludeToken/AlwaysToRe
<sp:RequestSecurityTokenTemplate>

<t:TokenType>http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.1#SAMLV2.0</t:TokenType>
<!--
  KeyType of "SymmetricKey", the client must prove to the WS service that it
  possesses a particular symmetric session key.
-->

<t:KeyType>http://docs.oasis-open.org/ws-sx/ws-trust/200512/SymmetricKey</t:KeyType>
  </sp:RequestSecurityTokenTemplate>
  <wsp:Policy>
    <sp:RequireInternalReference />
  </wsp:Policy>
<!--
  The sp:Issuer element defines the STS's address and endpoint information
  This information is used by the STSClient.
-->
  <sp:Issuer>

<wsaws:Address>http://@jboss.bind.address@:8080/jaxws-samples-wsse-policy-trust-sts-holderofkey/Se
<wsaws:Metadata
  xmlns:wsdli="http://www.w3.org/2006/01/wsdli-instance"

wsdli:wsdliLocation="http://@jboss.bind.address@:8080/jaxws-samples-wsse-policy-trust-sts-holderofk
<wsaw:ServiceName
  xmlns:wsaw="http://www.w3.org/2006/05/addressing/wsdli"
  xmlns:stsns="http://docs.oasis-open.org/ws-sx/ws-trust/200512/"
  EndpointName="UT_Port">stsns:SecurityTokenService</wsaw:ServiceName>
  </wsaws:Metadata>
</sp:Issuer>

  </sp:IssuedToken>
</wsp:Policy>
</sp:SignedEndorsingSupportingTokens>
<!--
  The sp:Wss11 element declares WSS: SOAP Message Security 1.1 options
  to be supported by the STS.  These particular elements generally refer
  to how keys are referenced within the SOAP envelope.  These are normally
  handled by CXF.
-->
  <sp:Wss11>
    <wsp:Policy>
      <sp:MustSupportRefIssuerSerial />
      <sp:MustSupportRefThumbprint />
      <sp:MustSupportRefEncryptedKey />
    </wsp:Policy>
  </sp:Wss11>
<!--
  The sp:Trust13 element declares controls for WS-Trust 1.3 options.
  They are policy assertions related to exchanges specifically with
  client and server challenges and entropy behaviors.  Again these are
  normally handled by CXF.
-->
  <sp:Trust13>
    <wsp:Policy>
      <sp:MustSupportIssuedTokens />
      <sp:RequireClientEntropy />
```



```
<sp:RequireServerEntropy />
</wsp:Policy>
</sp:Trust13>
</wsp:All>
</wsp:ExactlyOne>
</wsp:Policy>

</definitions>
```

SSL configuration

<https://docs.jboss.org/author/display/JBWS/WS-Trust+and+STS#WS-TrustandSTS-SSLconfiguration>

This web service is using https, therefore the JBoss server must be configured to provide SSL support in the Web subsystem. There are 2 components to SSL configuration.

- create a certificate keystore
- declare an SSL connector in the Web subsystem of the JBoss server configuration file.

Follow the directions in the, "*Using the pure Java implementation supplied by JSSE*" section in the [\[SSL Setup Guide\]](#).

Here is an example of an SSL connector declaration.

```
<subsystem xmlns="urn:jboss:domain:web:1.4" default-virtual-server="default-host"
native="false">
.....
  <connector name="jbws-https-connector" protocol="HTTP/1.1" scheme="https"
socket-binding="https" secure="true" enabled="true">
    <ssl key-alias="tomcat" password="changeit"
certificate-key-file="/myJbossHome/security/test.keystore" verify-client="false"/>
  </connector>
...

```

Web service Interface

The web service provider interface class, HolderOfKeyface, is a simple straight forward web service definition.

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.holderofkey;

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

@WebService
(
  targetNamespace = "http://www.jboss.org/jbossws/ws-extensions/holderofkeywssecuritypolicy"
)
public interface HolderOfKeyIface {
    @WebMethod
    String sayHello();
}

```



Web service Implementation

The web service provider implementation class, `HolderOfKeyImpl`, is a simple POJO. It uses the standard `WebService` annotation to define the service endpoint. In addition there are two Apache CXF annotations, `EndpointProperties` and `EndpointProperty` used for configuring the endpoint for the CXF runtime. These annotations come from the [Apache WSS4J project](#), which provides a Java implementation of the primary WS-Security standards for Web Services. These annotations are programmatically adding properties to the endpoint. With plain Apache CXF, these properties are often set via the `<jaxws:properties>` element on the `<jaxws:endpoint>` element in the Spring config; these annotations allow the properties to be configured in the code.

WSS4J uses the `Crypto` interface to get keys and certificates for signature creation/verification, as is asserted by the WSDL for this service. The WSS4J configuration information being provided by `HolderOfKeyImpl` is for `Crypto`'s Merlin implementation. More information will be provided about this in the [keystore](#) section.

The first `EndpointProperty` statement in the listing disables ensurance of compliance with the Basic Security Profile 1.1. The next `EndpointProperty` statements declares the Java properties file that contains the (Merlin) crypto configuration information. The last `EndpointProperty` statement declares the `STSHolderOfKeyCallbackHandler` implementation class. It is used to obtain the user's password for the certificates in the keystore file.

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.holderofkey;

import org.apache.cxf.annotations.EndpointProperties;
import org.apache.cxf.annotations.EndpointProperty;

import javax.jws.WebService;

@WebService
(
    portName = "HolderOfKeyServicePort",
    serviceName = "HolderOfKeyService",
    wsdlLocation = "WEB-INF/wsdl/HolderOfKeyService.wsdl",
    targetNamespace =
"http://www.jboss.org/jbossws/ws-extensions/holderofkeywssecuritypolicy",
    endpointInterface =
"org.jboss.test.ws.jaxws.samples.wsse.policy.trust.holderofkey.HolderOfKeyIface"
)
@EndpointProperties(value = {
    @EndpointProperty(key = "ws-security.is-bsp-compliant", value = "false"),
    @EndpointProperty(key = "ws-security.signature.properties", value =
"serviceKeystore.properties"),
    @EndpointProperty(key = "ws-security.callback-handler", value =
"org.jboss.test.ws.jaxws.samples.wsse.policy.trust.holderofkey.HolderOfKeyCallbackHandler")
})
public class HolderOfKeyImpl implements HolderOfKeyIface
{
    public String sayHello()
    {
        return "Holder-Of-Key WS-Trust Hello World!";
    }
}
```



Crypto properties and keystore files

WSS4J's Crypto implementation is loaded and configured via a Java properties file that contains Crypto configuration data. The file contains implementation-specific properties such as a keystore location, password, default alias and the like. This application is using the Merlin implementation. File `serviceKeystore.properties` contains this information.

File `servicestore.jks`, is a Java KeyStore (JKS) repository. It contains self signed certificates for `myservicekey` and `mystskey`. *Self signed certificates are not appropriate for production use.*

```
org.apache.ws.security.crypto.provider=org.apache.ws.security.components.crypto.Merlin
org.apache.ws.security.crypto.merlin.keystore.type=jks
org.apache.ws.security.crypto.merlin.keystore.password=sspass
org.apache.ws.security.crypto.merlin.keystore.alias=myservicekey
org.apache.ws.security.crypto.merlin.keystore.file=servicestore.jks
```

MANIFEST.MF

<https://docs.jboss.org/author/display/JBWS/WS-Trust+and+STS#WS-TrustandSTS-MANIFEST.MF>

When deployed on WildFly this application requires access to the JBossWs and CXF APIs provided in module `org.jboss.ws.cxf.jbossws-cxf-client`. The dependency statement directs the server to provide them at deployment.

```
Manifest-Version:1.0
Ant-Version: Apache Ant1.8.2
Created-By:1.7.0_25-b15 (Oracle Corporation)
Dependencies: org.jboss.ws.cxf.jbossws-cxf-client
```

Security Token Service

This section examines the crucial elements in providing the Security Token Service functionality for providing a SAML Holder-Of-Key token. The components that will be discussed are.

- Security Domain
- STS's WSDL
- STS's implementation class
- STSBearerCallbackHandler
- Crypto properties and keystore files
- MANIFEST.MF



Security Domain

The STS requires a JBoss security domain be configured. The `jboss-web.xml` descriptor declares a named security domain, "JBossWS-trust-sts" to be used by this service for authentication. This security domain requires two properties files and the addition of a security-domain declaration in the JBoss server configuration file.

For this scenario the domain needs to contain user *alice*, password *clarinet*, and role *friend*. See the listings below for `jbossws-users.properties` and `jbossws-roles.properties`. In addition the following XML must be added to the JBoss security subsystem in the server configuration file. Replace "**SOME_PATH**" with appropriate information.

```
<security-domain name="JBossWS-trust-sts">
  <authentication>
    <login-module code="UsersRoles" flag="required">
      <module-option name="usersProperties" value="/SOME_PATH/jbossws-users.properties"/>
      <module-option name="unauthenticatedIdentity" value="anonymous"/>
      <module-option name="rolesProperties" value="/SOME_PATH/jbossws-roles.properties"/>
    </login-module>
  </authentication>
</security-domain>
```

jboss-web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jboss-web PUBLIC "-//JBoss//DTD Web Application 2.4//EN" ">
<jboss-web>
  <security-domain>java:/jaas/JBossWS-trust-sts</security-domain>
</jboss-web>
```



jbossws-users.properties

```
# A sample users.properties file for use with the UsersRolesLoginModule
alice=clarinet
```



jbossws-roles.properties

```
# A sample roles.properties file for use with the UsersRolesLoginModule
alice=friend
```

STS's WSDL

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions
  targetNamespace="http://docs.oasis-open.org/ws-sx/ws-trust/200512/"
```



```
xmlns:tns="http://docs.oasis-open.org/ws-sx/ws-trust/200512/"
xmlns:wstrust="http://docs.oasis-open.org/ws-sx/ws-trust/200512/"
xmlns:wSDL="http://schemas.xmlsoap.org/wSDL/"
xmlns:soap="http://schemas.xmlsoap.org/wSDL/soap/"
xmlns:wsap10="http://www.w3.org/2006/05/addressing/wSDL"
xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
xmlns:wsp="http://www.w3.org/ns/ws-policy"
xmlns:wst="http://docs.oasis-open.org/ws-sx/ws-trust/200512"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata">

<wSDL:types>
  <xs:schema elementFormDefault="qualified"
    targetNamespace='http://docs.oasis-open.org/ws-sx/ws-trust/200512'>

    <xs:element name='RequestSecurityToken'
      type='wst:AbstractRequestSecurityTokenType' />
    <xs:element name='RequestSecurityTokenResponse'
      type='wst:AbstractRequestSecurityTokenType' />

    <xs:complexType name='AbstractRequestSecurityTokenType'>
      <xs:sequence>
        <xs:any namespace='##any' processContents='lax' minOccurs='0'
          maxOccurs='unbounded' />
      </xs:sequence>
      <xs:attribute name='Context' type='xs:anyURI' use='optional' />
      <xs:anyAttribute namespace='##other' processContents='lax' />
    </xs:complexType>
    <xs:element name='RequestSecurityTokenCollection'
      type='wst:RequestSecurityTokenCollectionType' />
    <xs:complexType name='RequestSecurityTokenCollectionType'>
      <xs:sequence>
        <xs:element name='RequestSecurityToken'
          type='wst:AbstractRequestSecurityTokenType' minOccurs='2'
          maxOccurs='unbounded' />
      </xs:sequence>
    </xs:complexType>

    <xs:element name='RequestSecurityTokenResponseCollection'
      type='wst:RequestSecurityTokenResponseCollectionType' />
    <xs:complexType name='RequestSecurityTokenResponseCollectionType'>
      <xs:sequence>
        <xs:element ref='wst:RequestSecurityTokenResponse' minOccurs='1'
          maxOccurs='unbounded' />
      </xs:sequence>
      <xs:anyAttribute namespace='##other' processContents='lax' />
    </xs:complexType>

  </xs:schema>
</wSDL:types>

<!-- WS-Trust defines the following GEDs -->
<wSDL:message name="RequestSecurityTokenMsg">
  <wSDL:part name="request" element="wst:RequestSecurityToken"/>
</wSDL:message>
<wSDL:message name="RequestSecurityTokenResponseMsg">
  <wSDL:part name="response"
    element="wst:RequestSecurityTokenResponse"/>
</wSDL:message>
```




```
</wsdl:message>
<wsdl:message name="RequestSecurityTokenCollectionMsg">
  <wsdl:part name="requestCollection"
    element="wst:RequestSecurityTokenCollection"/>
</wsdl:message>
<wsdl:message name="RequestSecurityTokenResponseCollectionMsg">
  <wsdl:part name="responseCollection"
    element="wst:RequestSecurityTokenResponseCollection"/>
</wsdl:message>

<!-- This portType an example of a Requestor (or other) endpoint that
      Accepts SOAP-based challenges from a Security Token Service -->
<wsdl:portType name="WSSecurityRequestor">
  <wsdl:operation name="Challenge">
    <wsdl:input message="tns:RequestSecurityTokenResponseMsg"/>
    <wsdl:output message="tns:RequestSecurityTokenResponseMsg"/>
  </wsdl:operation>
</wsdl:portType>

<!-- This portType is an example of an STS supporting full protocol -->
<wsdl:portType name="STS">
  <wsdl:operation name="Cancel">
    <wsdl:input
      wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Cancel"
      message="tns:RequestSecurityTokenMsg"/>
    <wsdl:output
      wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/CancelFinal"
      message="tns:RequestSecurityTokenResponseMsg"/>
  </wsdl:operation>
  <wsdl:operation name="Issue">
    <wsdl:input
      wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Issue"
      message="tns:RequestSecurityTokenMsg"/>
    <wsdl:output
      wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTRC/IssueFinal"
      message="tns:RequestSecurityTokenResponseCollectionMsg"/>
  </wsdl:operation>
  <wsdl:operation name="Renew">
    <wsdl:input
      wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Renew"
      message="tns:RequestSecurityTokenMsg"/>
    <wsdl:output
      wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/RenewFinal"
      message="tns:RequestSecurityTokenResponseMsg"/>
  </wsdl:operation>
  <wsdl:operation name="Validate">
    <wsdl:input
      wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Validate"
      message="tns:RequestSecurityTokenMsg"/>
    <wsdl:output
      wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/ValidateFinal"
      message="tns:RequestSecurityTokenResponseMsg"/>
  </wsdl:operation>
  <wsdl:operation name="KeyExchangeToken">
    <wsdl:input
      wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/KET"
      message="tns:RequestSecurityTokenMsg"/>
    <wsdl:output
```



```
        wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/KETFinal"
        message="tns:RequestSecurityTokenResponseMsg"/>
    </wsdl:operation>
    <wsdl:operation name="RequestCollection">
        <wsdl:input message="tns:RequestSecurityTokenCollectionMsg"/>
        <wsdl:output message="tns:RequestSecurityTokenResponseCollectionMsg"/>
    </wsdl:operation>
</wsdl:portType>

<!-- This portType is an example of an endpoint that accepts
     Unsolicited RequestSecurityTokenResponse messages -->
<wsdl:portType name="SecurityTokenResponseService">
    <wsdl:operation name="RequestSecurityTokenResponse">
        <wsdl:input message="tns:RequestSecurityTokenResponseMsg"/>
    </wsdl:operation>
</wsdl:portType>

<wsdl:binding name="UT_Binding" type="wstrust:STS">
    <wsp:PolicyReference URI="#UT_policy"/>
    <soap:binding style="document"
        transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="Issue">
        <soap:operation
            soapAction="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Issue"/>
        <wsdl:input>
            <wsp:PolicyReference
                URI="#Input_policy"/>
            <soap:body use="literal"/>
        </wsdl:input>
        <wsdl:output>
            <wsp:PolicyReference
                URI="#Output_policy"/>
            <soap:body use="literal"/>
        </wsdl:output>
    </wsdl:operation>
    <wsdl:operation name="Validate">
        <soap:operation
            soapAction="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Validate"/>
        <wsdl:input>
            <wsp:PolicyReference
                URI="#Input_policy"/>
            <soap:body use="literal"/>
        </wsdl:input>
        <wsdl:output>
            <wsp:PolicyReference
                URI="#Output_policy"/>
            <soap:body use="literal"/>
        </wsdl:output>
    </wsdl:operation>
    <wsdl:operation name="Cancel">
        <soap:operation
            soapAction="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Cancel"/>
        <wsdl:input>
            <soap:body use="literal"/>
        </wsdl:input>
        <wsdl:output>
            <soap:body use="literal"/>
        </wsdl:output>
    </wsdl:operation>
</wsdl:binding>
</wsdl:service>
</wsdl:definitions>
```



```
</wsdl:operation>
<wsdl:operation name="Renew">
  <soap:operation
    soapAction="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Renew"/>
  <wsdl:input>
    <soap:body use="literal"/>
  </wsdl:input>
  <wsdl:output>
    <soap:body use="literal"/>
  </wsdl:output>
</wsdl:operation>
<wsdl:operation name="KeyExchangeToken">
  <soap:operation
    soapAction="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/KeyExchangeToken"/>
  <wsdl:input>
    <soap:body use="literal"/>
  </wsdl:input>
  <wsdl:output>
    <soap:body use="literal"/>
  </wsdl:output>
</wsdl:operation>
<wsdl:operation name="RequestCollection">
  <soap:operation
    soapAction="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/RequestCollection"/>
  <wsdl:input>
    <soap:body use="literal"/>
  </wsdl:input>
  <wsdl:output>
    <soap:body use="literal"/>
  </wsdl:output>
</wsdl:operation>
</wsdl:binding>

<wsdl:service name="SecurityTokenService">
  <wsdl:port name="UT_Port" binding="tns:UT_Binding">
    <soap:address location="http://localhost:8080/SecurityTokenService/UT"/>
  </wsdl:port>
</wsdl:service>

<wsp:Policy wsu:Id="UT_policy">
  <wsp:ExactlyOne>
    <wsp:All>
      <wsap10:UsingAddressing/>
      <sp:SymmetricBinding
        xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
        <wsp:Policy>
          <sp:ProtectionToken>
            <wsp:Policy>
              <sp:X509Token
                sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/IncludeToken/Never">
                <wsp:Policy>
                  <sp:RequireDerivedKeys/>
                  <sp:RequireThumbprintReference/>
                  <sp:WssX509V3Token10/>
                </wsp:Policy>
              </sp:X509Token>
            </wsp:Policy>
          </sp:ProtectionToken>
        </wsp:Policy>
      </sp:SymmetricBinding>
    </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>
```



```
</sp:ProtectionToken>
<sp:AlgorithmSuite>
  <wsp:Policy>
    <sp:Basic256/>
  </wsp:Policy>
</sp:AlgorithmSuite>
<sp:Layout>
  <wsp:Policy>
    <sp:Lax/>
  </wsp:Policy>
</sp:Layout>
<sp:IncludeTimestamp/>
<sp:EncryptSignature/>
<sp:OnlySignEntireHeadersAndBody/>
</wsp:Policy>
</sp:SymmetricBinding>
<sp:SignedSupportingTokens
  xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
  <wsp:Policy>
    <sp:UsernameToken

sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/IncludeToken/AlwaysToRe
<wsp:Policy>
  <sp:WssUsernameToken10/>
  </wsp:Policy>
  </sp:UsernameToken>
</wsp:Policy>
</sp:SignedSupportingTokens>
<sp:Wss11
  xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
  <wsp:Policy>
    <sp:MustSupportRefKeyIdentifier/>
    <sp:MustSupportRefIssuerSerial/>
    <sp:MustSupportRefThumbprint/>
    <sp:MustSupportRefEncryptedKey/>
  </wsp:Policy>
</sp:Wss11>
<sp:Trust13
  xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
  <wsp:Policy>
    <sp:MustSupportIssuedTokens/>
    <sp:RequireClientEntropy/>
    <sp:RequireServerEntropy/>
  </wsp:Policy>
</sp:Trust13>
</wsp:All>
</wsp:ExactlyOne>
</wsp:Policy>

<wsp:Policy wsu:Id="Input_policy">
  <wsp:ExactlyOne>
    <wsp:All>
      <sp:SignedParts
        xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
        <sp:Body/>
        <sp:Header Name="To"
          Namespace="http://www.w3.org/2005/08/addressing"/>
        <sp:Header Name="From"
```



```
        Namespace="http://www.w3.org/2005/08/addressing"/>
    <sp:Header Name="FaultTo"
        Namespace="http://www.w3.org/2005/08/addressing"/>
    <sp:Header Name="ReplyTo"
        Namespace="http://www.w3.org/2005/08/addressing"/>
    <sp:Header Name="MessageID"
        Namespace="http://www.w3.org/2005/08/addressing"/>
    <sp:Header Name="RelatesTo"
        Namespace="http://www.w3.org/2005/08/addressing"/>
    <sp:Header Name="Action"
        Namespace="http://www.w3.org/2005/08/addressing"/>
    </sp:SignedParts>
</wsp:All>
</wsp:ExactlyOne>
</wsp:Policy>

<wsp:Policy wsu:Id="Output_policy">
    <wsp:ExactlyOne>
        <wsp:All>
            <sp:SignedParts
                xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
                <sp:Body/>
                <sp:Header Name="To"
                    Namespace="http://www.w3.org/2005/08/addressing"/>
                <sp:Header Name="From"
                    Namespace="http://www.w3.org/2005/08/addressing"/>
                <sp:Header Name="FaultTo"
                    Namespace="http://www.w3.org/2005/08/addressing"/>
                <sp:Header Name="ReplyTo"
                    Namespace="http://www.w3.org/2005/08/addressing"/>
                <sp:Header Name="MessageID"
                    Namespace="http://www.w3.org/2005/08/addressing"/>
                <sp:Header Name="RelatesTo"
                    Namespace="http://www.w3.org/2005/08/addressing"/>
                <sp:Header Name="Action"
                    Namespace="http://www.w3.org/2005/08/addressing"/>
            </sp:SignedParts>
        </wsp:All>
    </wsp:ExactlyOne>
</wsp:Policy>

</wsdl:definitions>
```

STS's implementation class

The Apache CXF's STS, `SecurityTokenServiceProvider`, is a web service provider that is compliant with the protocols and functionality defined by the WS-Trust specification. It has a modular architecture. Many of its components are configurable or replaceable and there are many optional features that are enabled by implementing and configuring plug-ins. Users can customize their own STS by extending from `SecurityTokenServiceProvider` and overriding the default settings. Extensive information about the CXF's STS configurable and pluggable components can be found [here](#).



This STS implementation class, `SampleSTSHolderOfKey`, is a POJO that extends from `SecurityTokenServiceProvider`. Note that the class is defined with a `WebServiceProvider` annotation and not a `WebService` annotation. This annotation defines the service as a Provider-based endpoint, meaning it supports a more messaging-oriented approach to Web services. In particular, it signals that the exchanged messages will be XML documents of some type. `SecurityTokenServiceProvider` is an implementation of the `javax.xml.ws.Provider` interface. In comparison the `WebService` annotation defines a (service endpoint interface) SEI-based endpoint which supports message exchange via SOAP envelopes.

As was done in the `HolderOfKeyImpl` class, the WSS4J annotations `EndpointProperties` and `EndpointProperty` are providing endpoint configuration for the CXF runtime. The first `EndpointProperty` statements declares the Java properties file that contains the (Merlin) crypto configuration information. WSS4J reads this file and extra required information for message handling. The last `EndpointProperty` statement declares the `STSHolderOfKeyCallbackHandler` implementation class. It is used to obtain the user's password for the certificates in the keystore file.

In this implementation we are customizing the operations of token issuance and their static properties.

`StaticSTSProperties` is used to set select properties for configuring resources in the STS. You may think this is a duplication of the settings made with the WSS4J annotations. The values are the same but the underlying structures being set are different, thus this information must be declared in both places.

The `setIssuer` setting is important because it uniquely identifies the issuing STS. The issuer string is embedded in issued tokens and, when validating tokens, the STS checks the issuer string value. Consequently, it is important to use the issuer string in a consistent way, so that the STS can recognize the tokens that it has issued.

The `setEndpoints` call allows the declaration of a set of allowed token recipients by address. The addresses are specified as reg-ex patterns.

`TokenIssueOperation` has a modular structure. This allows custom behaviors to be injected into the processing of messages. In this case we are overriding the `SecurityTokenServiceProvider`'s default behavior and performing SAML token processing. CXF provides an implementation of a `SAMLTokenProvider` which we are using rather than writing our own.

Learn more about the `SAMLTokenProvider` [here](#).

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.stsholderofkey;

import org.apache.cxf.annotations.EndpointProperties;
import org.apache.cxf.annotations.EndpointProperty;
import org.apache.cxf.sts.StaticSTSProperties;
import org.apache.cxf.sts.operation.TokenIssueOperation;
import org.apache.cxf.sts.service.ServiceMBean;
import org.apache.cxf.sts.service.StaticService;
import org.apache.cxf.sts.token.provider.SAMLTokenProvider;
import org.apache.cxf.ws.security.sts.provider.SecurityTokenServiceProvider;

import javax.xml.ws.WebServiceProvider;
import java.util.Arrays;
import java.util.LinkedList;
import java.util.List;
```



```
/**
 * User: rsearls
 * Date: 3/14/14
 */
@WebServiceProvider(serviceName = "SecurityTokenService",
    portName = "UT_Port",
    targetNamespace = "http://docs.oasis-open.org/ws-sx/ws-trust/200512/",
    wsdlLocation = "WEB-INF/wsdl/holderofkey-ws-trust-1.4-service.wsdl")
//be sure to have dependency on org.apache.cxf module when on AS7, otherwise Apache CXF
annotations are ignored
@EndpointProperties(value = {
    @EndpointProperty(key = "ws-security.signature.properties", value =
"stsKeystore.properties"),
    @EndpointProperty(key = "ws-security.callback-handler", value =
"org.jboss.test.ws.jaxws.samples.wsse.policy.trust.stsholderofkey.STSHolderOfKeyCallbackHandler")
})
class SampleSTSHolderOfKey extends SecurityTokenServiceProvider
{

    public SampleSTSHolderOfKey() throws Exception
    {
        super();

        StaticSTSPProperties props = new StaticSTSPProperties();
        props.setSignatureCryptoProperties("stsKeystore.properties");
        props.setSignatureUsername("mystskey");
        props.setCallbackHandlerClass(STSHolderOfKeyCallbackHandler.class.getName());
        props.setEncryptionCryptoProperties("stsKeystore.properties");
        props.setEncryptionUsername("myservicekey");
        props.setIssuer("DoubleItSTSIssuer");

        List<ServiceMBean> services = new LinkedList<ServiceMBean>();
        StaticService service = new StaticService();
        service.setEndpoints(Arrays.asList(

"https://localhost:(\\d)*/jaxws-samples-wsse-policy-trust-holderofkey/HolderOfKeyService",

"https://\\[::1\\]:(\\d)*/jaxws-samples-wsse-policy-trust-holderofkey/HolderOfKeyService",

"https://\\[0:0:0:0:0:0:1\\]:(\\d)*/jaxws-samples-wsse-policy-trust-holderofkey/HolderOfKeyService
));

        services.add(service);

        TokenIssueOperation issueOperation = new TokenIssueOperation();
        issueOperation.getTokenProviders().add(new SAMLTokenProvider());
        issueOperation.setServices(services);
        issueOperation.setStsProperties(props);
        this.setIssueOperation(issueOperation);
    }
}
```



HolderOfKeyCallbackHandler

STSHolderOfKeyCallbackHandler is a callback handler for the WSS4J Crypto API. It is used to obtain the password for the private key in the keystore. This class enables CXF to retrieve the password of the user name to use for the message signature.

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.stsholderofkey;

import org.jboss.wsf.stack.cxf.extensions.security.PasswordCallbackHandler;

import java.util.HashMap;
import java.util.Map;

/**
 * User: rsearls
 * Date: 3/19/14
 */
public class STSHolderOfKeyCallbackHandler extends PasswordCallbackHandler
{
    public STSHolderOfKeyCallbackHandler()
    {
        super(getInitMap());
    }

    private static Map<String, String> getInitMap()
    {
        Map<String, String> passwords = new HashMap<String, String>();
        passwords.put("mystskey", "stskpass");
        passwords.put("alice", "clarinet");
        return passwords;
    }
}
```

Crypto properties and keystore files

WSS4J's Crypto implementation is loaded and configured via a Java properties file that contains Crypto configuration data. The file contains implementation-specific properties such as a keystore location, password, default alias and the like. This application is using the Merlin implementation. File `stsKeystore.properties` contains this information.

File `servicestore.jks`, is a Java KeyStore (JKS) repository. It contains self signed certificates for `myservicekey` and `mystskey`. *Self signed certificates are not appropriate for production use.*

```
org.apache.ws.security.crypto.provider=org.apache.ws.security.components.crypto.Merlin
org.apache.ws.security.crypto.merlin.keystore.type=jks
org.apache.ws.security.crypto.merlin.keystore.password=stsspass
org.apache.ws.security.crypto.merlin.keystore.file=stsstore.jks
```




MANIFEST.MF

When deployed on WildFly, this application requires access to the JBossWs and CXF APIs provided in modules `org.jboss.ws.cxf.jbossws-cxf-client` and `org.apache.cxf`. The Apache CXF internals, `org.apache.cxf.impl`, are needed to build the STS configuration in the `SampleSTSHolderOfKey` constructor. The dependency statement directs the server to provide them at deployment.

```
Manifest-Version:1.0
Ant-Version: Apache Ant1.8.2
Created-By:1.7.0_25-b15 (Oracle Corporation)
Dependencies: org.jboss.ws.cxf.jbossws-cxf-client,org.apache.cxf.impl
```

Web service requester

This section examines the crucial elements in calling a web service that implements endpoint security as described in the SAML Holder-Of-Key scenario. The components that will be discussed are.

- web service requester's implementation
- `ClientCallbackHandler`
- Crypto properties and keystore files



Web service requester Implementation

The ws-requester, the client, uses standard procedures for creating a reference to the web service. To address the endpoint security requirements, the web service's "Request Context" is configured with the information needed in message generation. In addition, the STSClient that communicates with the STS is configured with similar values. Note the key strings ending with a ".it" suffix. This suffix flags these settings as belonging to the STSClient. The internal CXF code assigns this information to the STSClient that is auto-generated for this service call.

There is an alternate method of setting up the STSClient. The user may provide their own instance of the STSClient. The CXF code will use this object and not auto-generate one. When providing the STSClient in this way, the user must provide a `org.apache.cxf.Bus` for it and the configuration keys must not have the ".it" suffix. This is used in the `ActAs` and `OnBehalfOf` examples.

```
String serviceURL = "https://" + getServerHost() +
":8443/jaxws-samples-wsse-policy-trust-holderofkey/HolderOfKeyService";

final QName serviceName = new
QName("http://www.jboss.org/jbossws/ws-extensions/holderofkeywssecuritypolicy",
"HolderOfKeyService");
final URL wsdlURL = new URL(serviceURL + "?wsdl");
Service service = Service.create(wsdlURL, serviceName);
HolderOfKeyIface proxy = (HolderOfKeyIface) service.getPort(HolderOfKeyIface.class);

Map<String, Object> ctx = ((BindingProvider)proxy).getRequestContext();

// set the security related configuration information for the service "request"
ctx.put(SecurityConstants.CALLBACK_HANDLER, new ClientCallbackHandler());
ctx.put(SecurityConstants.SIGNATURE_PROPERTIES,
Thread.currentThread().getContextClassLoader().getResource(
"META-INF/clientKeystore.properties"));
ctx.put(SecurityConstants.ENCRYPT_PROPERTIES,
Thread.currentThread().getContextClassLoader().getResource(
"META-INF/clientKeystore.properties"));
ctx.put(SecurityConstants.SIGNATURE_USERNAME, "myclientkey");
ctx.put(SecurityConstants.ENCRYPT_USERNAME, "myservicekey");

/-- Configuration settings that will be transfered to the STSClient
// "alice" is the name provided for the WSS Username. Her password will
// be retrieved from the ClientCallbackHander by the STSClient.
ctx.put(SecurityConstants.USERNAME + ".it", "alice");
ctx.put(SecurityConstants.CALLBACK_HANDLER + ".it", new ClientCallbackHandler());
ctx.put(SecurityConstants.ENCRYPT_PROPERTIES + ".it",
Thread.currentThread().getContextClassLoader().getResource(
"META-INF/clientKeystore.properties"));
ctx.put(SecurityConstants.ENCRYPT_USERNAME + ".it", "mystskey");
ctx.put(SecurityConstants.STS_TOKEN_USERNAME + ".it", "myclientkey");
ctx.put(SecurityConstants.STS_TOKEN_PROPERTIES + ".it",
Thread.currentThread().getContextClassLoader().getResource(
"META-INF/clientKeystore.properties"));
ctx.put(SecurityConstants.STS_TOKEN_USE_CERT_FOR_KEYINFO + ".it", "true");

proxy.sayHello();
```



ClientCallbackHandler

ClientCallbackHandler is a callback handler for the WSS4J Crypto API. It is used to obtain the password for the private key in the keystore. This class enables CXF to retrieve the password of the user name to use for the message signature. Note that "alice" and her password have been provided here. This information is not in the (JKS) keystore but provided in the WildFly security domain. It was declared in file `jbossws-users.properties`.

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.shared;

import java.io.IOException;
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.UnsupportedCallbackException;
import org.apache.ws.security.WSPasswordCallback;

public class ClientCallbackHandler implements CallbackHandler {

    public void handle(Callback[] callbacks) throws IOException,
        UnsupportedCallbackException {
        for (int i = 0; i < callbacks.length; i++) {
            if (callbacks[i] instanceof WSPasswordCallback) {
                WSPasswordCallback pc = (WSPasswordCallback) callbacks[i];
                if ("myclientkey".equals(pc.getIdentifier())) {
                    pc.setPassword("ckpass");
                    break;
                } else if ("alice".equals(pc.getIdentifier())) {
                    pc.setPassword("clarinet");
                    break;
                } else if ("bob".equals(pc.getIdentifier())) {
                    pc.setPassword("trombone");
                    break;
                } else if ("myservicekey".equals(pc.getIdentifier())) { // rls test added for
bearer test
                    pc.setPassword("skpass");
                    break;
                }
            }
        }
    }
}
```



Crypto properties and keystore files

WSS4J's Crypto implementation is loaded and configured via a Java properties file that contains Crypto configuration data. The file contains implementation-specific properties such as a keystore location, password, default alias and the like. This application is using the Merlin implementation. File `clientKeystore.properties` contains this information.

File `clientstore.jks`, is a Java KeyStore (JKS) repository. It contains self signed certificates for `myservicekey` and `mystskey`. *Self signed certificates are not appropriate for production use.*

```
org.apache.ws.security.crypto.provider=org.apache.ws.security.components.crypto.Merlin
org.apache.ws.security.crypto.merlin.keystore.type=jks
org.apache.ws.security.crypto.merlin.keystore.password=cspass
org.apache.ws.security.crypto.merlin.keystore.alias=myclientkey
org.apache.ws.security.crypto.merlin.keystore.file=META-INF/clientstore.jks
```

35.3.14 Reliable Messaging

JBoss Web Services inherits full WS-Reliable Messaging capabilities from the underlying Apache CXF implementation. At the time of writing, Apache CXF provides support for the [WS-Reliable Messaging 1.0](#) (February 2005) version of the specification.

Enabling WS-Reliable Messaging

WS-Reliable Messaging is implemented internally in Apache CXF through a set of interceptors that deal with the low level requirements of the reliable messaging protocol. In order for enabling WS-Reliable Messaging, users need to either:

- consume a WSDL contract that specifies proper WS-Reliable Messaging policies / assertions
- manually add / configure the reliable messaging interceptors
- specify the reliable messaging policies in an optional CXF Spring XML descriptor
- specify the Apache CXF reliable messaging feature in an optional CXF Spring XML descriptor

The former approach relies on the Apache CXF WS-Policy engine and is the only portable one. The other approaches are Apache CXF proprietary ones, however they allow for fine-grained configuration of protocol aspects that are not covered by the WS-Reliable Messaging Policy. More details are available in the [Apache CXF documentation](#).

Example

In this example we configure WS-Reliable Messaging endpoint and client through the WS-Policy support.



Endpoint

We go with a contract-first approach, so we start by creating a proper WSDL contract, containing the WS-Reliable Messaging and WS-Addressing policies (the latter is a requirement of the former):

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions name="SimpleService"
targetNamespace="http://www.jboss.org/jbossws/ws-extensions/wsrn"
  xmlns:tns="http://www.jboss.org/jbossws/ws-extensions/wsrn"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:wSDL="http://schemas.xmlsoap.org/wsdl/"
  xmlns:wsp="http://www.w3.org/2006/07/ws-policy">

  <wsdl:types>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:tns="http://www.jboss.org/jbossws/ws-extensions/wsrn"
  attributeFormDefault="unqualified" elementFormDefault="unqualified"
  targetNamespace="http://www.jboss.org/jbossws/ws-extensions/wsrn">
<xsd:element name="ping" type="tns:ping"/>
<xsd:complexType name="ping">
<xsd:sequence/>
</xsd:complexType>
<xsd:element name="echo" type="tns:echo"/>
<xsd:complexType name="echo">
<xsd:sequence>
<xsd:element minOccurs="0" name="arg0" type="xsd:string"/>
</xsd:sequence>
</xsd:complexType>
<xsd:element name="echoResponse" type="tns:echoResponse"/>
<xsd:complexType name="echoResponse">
<xsd:sequence>
<xsd:element minOccurs="0" name="return" type="xsd:string"/>
</xsd:sequence>
</xsd:complexType>
</xsd:schema>
  </wsdl:types>
  <wsdl:message name="echoResponse">
    <wsdl:part name="parameters" element="tns:echoResponse">
      </wsdl:part>
    </wsdl:message>
  <wsdl:message name="echo">
    <wsdl:part name="parameters" element="tns:echo">
      </wsdl:part>
    </wsdl:message>
  <wsdl:message name="ping">
    <wsdl:part name="parameters" element="tns:ping">
      </wsdl:part>
    </wsdl:message>
  <wsdl:portType name="SimpleService">
    <wsdl:operation name="ping">
      <wsdl:input name="ping" message="tns:ping">
        </wsdl:input>
      </wsdl:operation>
    <wsdl:operation name="echo">
      <wsdl:input name="echo" message="tns:echo">

```



```
</wsdl:input>
  <wsdl:output name="echoResponse" message="tns:echoResponse">
</wsdl:output>
</wsdl:operation>
</wsdl:portType>
<wsdl:binding name="SimpleServiceSoapBinding" type="tns:SimpleService">
  <wsp:Policy>
    <!-- WS-Addressing and basic WS-Reliable Messaging policy assertions -->
    <wsa:UsingAddressing xmlns:wsa="http://www.w3.org/2006/05/addressing/wsdl"/>
    <wsrmp:RMAssertion xmlns:wsrmp="http://schemas.xmlsoap.org/ws/2005/02/rm/policy"/>
    <!-- ----->
  </wsp:Policy>
  <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
  <wsdl:operation name="ping">
    <soap:operation soapAction="" style="document"/>
    <wsdl:input name="ping">
      <soap:body use="literal"/>
    </wsdl:input>
  </wsdl:operation>
  <wsdl:operation name="echo">
    <soap:operation soapAction="" style="document"/>
    <wsdl:input name="echo">
      <soap:body use="literal"/>
    </wsdl:input>
    <wsdl:output name="echoResponse">
      <soap:body use="literal"/>
    </wsdl:output>
  </wsdl:operation>
</wsdl:binding>
<wsdl:service name="SimpleService">
  <wsdl:port name="SimpleServicePort" binding="tns:SimpleServiceSoapBinding">
    <soap:address location="http://localhost:8080/jaxws-samples-wsrm-api"/>
  </wsdl:port>
</wsdl:service>
</wsdl:definitions>
```

Then we use the *wconsume* tool to generate both standard JAX-WS client and endpoint.

We provide a basic JAX-WS implementation for the endpoint, nothing special in it:



```
package org.jboss.test.ws.jaxws.samples.wsrn.service;

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

@WebService
(
    name = "SimpleService",
    serviceName = "SimpleService",
    wsdlLocation = "WEB-INF/wsdl/SimpleService.wsdl",
    targetNamespace = "http://www.jboss.org/jbossws/ws-extensions/wsrn"
)
public class SimpleServiceImpl
{
    @Oneway
    @WebMethod
    public void ping()
    {
        System.out.println("ping()");
    }

    @WebMethod
    public String echo(String s)
    {
        System.out.println("echo(" + s + ")");
        return s;
    }
}
```

Finally we package the generated POJO endpoint together with a basic web.xml the usual way and deploy to the application server. The webservices stack automatically detects the policies and enables WS-Reliable Messaging.

Client

The endpoint advertises his RM capabilities (and requirements) through the published WSDL and the client is required to also enable WS-RM for successfully exchanging messages with the server.

So a regular JAX WS client is enough if the user does not need to tune any specific detail of the RM subsystem.

```
QName serviceName = new QName("http://www.jboss.org/jbossws/ws-extensions/wsrn",
"SimpleService");
URL wsdlURL = new URL("http://localhost:8080/jaxws-samples-wsrn-api?wsdl");
Service service = Service.create(wsdlURL, serviceName);
proxy = (SimpleService)service.getPort(SimpleService.class);
proxy.echo("Hello World!");
```



Additional configuration

Fine-grained tuning of WS-Reliable Messaging engine requires setting up proper RM features and attach them for instance to the client proxy. Here is an example:

```
package org.jboss.test.ws.jaxws.samples.wsrn.client;

//...
import javax.xml.ws.Service;
import org.apache.cxf.ws.rm.feature.RMFeature;
import org.apache.cxf.ws.rm.manager.AcksPolicyType;
import org.apache.cxf.ws.rm.manager.DestinationPolicyType;
import org.jboss.test.ws.jaxws.samples.wsrn.generated.SimpleService;

// ...
Service service = Service.create(wsdlURL, serviceName);

RMFeature feature = new RMFeature();
RMAssertion rma = new RMAssertion();
RMAssertion.BaseRetransmissionInterval bri = new RMAssertion.BaseRetransmissionInterval();
bri.setMilliseconds(4000L);
rma.setBaseRetransmissionInterval(bri);
AcknowledgementInterval ai = new AcknowledgementInterval();
ai.setMilliseconds(2000L);
rma.setAcknowledgementInterval(ai);
feature.setRMAssertion(rma);
DestinationPolicyType dp = new DestinationPolicyType();
AcksPolicyType ap = new AcksPolicyType();
ap.setIntraMessageThreshold(0);
dp.setAcksPolicy(ap);
feature.setDestinationPolicy(dp);

SimpleService proxy = (SimpleService)service.getPort(SimpleService.class, feature);
proxy.echo("Hello World");
```

The same can of course be achieved by factoring the feature into a custom pojo extending `org.apache.cxf.ws.rm.feature.RMFeature` and setting the obtained property in a client configuration:



```
package org.jboss.test.ws.jaxws.samples.wsrn.client;

import org.apache.cxf.ws.rm.feature.RMFeature;
import org.apache.cxf.ws.rm.manager.AcksPolicyType;
import org.apache.cxf.ws.rm.manager.DestinationPolicyType;
import org.apache.cxf.ws.rmp.v200502.RMAssertion;
import org.apache.cxf.ws.rmp.v200502.RMAssertion.AcknowledgementInterval;

public class CustomRMFeature extends RMFeature
{
    public CustomRMFeature() {
        super();
        RMAssertion rma = new RMAssertion();
        RMAssertion.BaseRetransmissionInterval bri = new RMAssertion.BaseRetransmissionInterval();
        bri.setMilliseconds(4000L);
        rma.setBaseRetransmissionInterval(bri);
        AcknowledgementInterval ai = new AcknowledgementInterval();
        ai.setMilliseconds(2000L);
        rma.setAcknowledgementInterval(ai);
        super.setRMAssertion(rma);
        DestinationPolicyType dp = new DestinationPolicyType();
        AcksPolicyType ap = new AcksPolicyType();
        ap.setIntraMessageThreshold(0);
        dp.setAcksPolicy(ap);
        super.setDestinationPolicy(dp);
    }
}
```

... this is how the `jaxws-client-config.xml` descriptor would look:

```
<?xml version="1.0" encoding="UTF-8"?>

<jaxws-config xmlns="urn:jboss:jbossws-jaxws-config:4.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:javaee="http://java.sun.com/xml/ns/javaee"
xsi:schemaLocation="urn:jboss:jbossws-jaxws-config:4.0 schema/jbossws-jaxws-config_4_0.xsd">

<client-config>
<config-name>Custom Client Config</config-name>
<property>
<property-name>cxf.features</property-name>
<property-value>org.jboss.test.ws.jaxws.samples.wsrn.client.CustomRMFeature</property-value>
</property>
</client-config>

</jaxws-config>
```

... and this is how the client would set the configuration:



```
import org.jboss.ws.api.configuration.ClientConfigUtil;
import org.jboss.ws.api.configuration.ClientConfigurer;

//...
Service service = Service.create(wsdlURL, serviceName);
SimpleService proxy = (SimpleService)service.getPort(SimpleService.class);

ClientConfigurer configurer = ClientConfigUtil.resolveClientConfigurer();
configurer.setConfigProperties(proxy, "META-INF/jaxws-client-config.xml", "Custom Client
Config");
proxy.echo("Hello World!");
```

35.3.15 SOAP over JMS

JBoss Web Services allows communication over the *JMS* transport. The functionality comes from Apache CXF support for the [SOAP over Java Message Service 1.0](#) specification, which is aimed at a set of standards for interoperable transport of *SOAP* messages over *JMS*.

On top of Apache CXF functionalities, the JBossWS integration allows users to deploy *WS* archives containing both *JMS* and *HTTP* endpoints the same way as they do for basic *HTTP* *WS* endpoints (in *war* archives). The webservices layer of WildFly takes care of looking for *JMS* endpoints in the deployed archive and starts them delegating to the Apache CXF core similarly as with *HTTP* endpoints.



Configuring SOAP over JMS

As per specification, the *SOAP over JMS* transport configuration is controlled by proper elements and attributes in the `binding` and `service` elements of the WSDL contract. So a *JMS* endpoint is usually developed using a contract-first approach.

The [Apache CXF documentation](#) covers all the details of the supported configurations. The minimum configuration implies:

- setting a proper JMS URI in the `soap:address` location [1]
- providing a JNDI connection factory name to be used for connecting to the queues [2]
- setting the transport binding [3]

```
<wsdl:definitions name="HelloWorldService" targetNamespace="http://org.jboss.ws/jaxws/cxf/jms"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://org.jboss.ws/jaxws/cxf/jms"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soapjms="http://www.w3.org/2010/soapjms/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  ...

  <wsdl:binding name="HelloWorldServiceSoapBinding" type="tns:HelloWorld">
    <soap:binding style="document" transport="http://www.w3.org/2010/soapjms/" /> <!-- 3 -->
    <wsdl:operation name="echo">
      <soap:operation soapAction="" style="document" />
      <wsdl:input name="echo">
        <soap:body use="literal" />
      </wsdl:input>
      <wsdl:output name="echoResponse">
        <soap:body use="literal" />
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="HelloWorldService">
    <soapjms:jndiConnectionFactoryName>java:/ConnectionFactory</soapjms:jndiConnectionFactoryName>
    <!-- 2 -->
    <wsdl:port binding="tns:HelloWorldServiceSoapBinding" name="HelloWorldImplPort">
      <soap:address location="jms:queue:testQueue" /> <!-- 1 -->
    </wsdl:port>
  </wsdl:service>
```

Apache CXF takes care of setting up the JMS transport for endpoint implementations whose `@WebService` annotation points to a port declared for JMS transport as explained above.



JBossWS currently supports POJO endpoints only for JMS transport use. The endpoint classes can be deployed as part of *jar* or *war* archives.

The *web.xml* descriptor in *war* archives doesn't need any entry for JMS endpoints.



Examples

JMS endpoint only deployment

In this example we create a simple endpoint relying on *SOAP over JMS* and deploy it as part of a jar archive.

The endpoint is created using wsconsume tool from a WSDL contract such as:


```
<?xml version='1.0' encoding='UTF-8'?>
<wsdl:definitions name="HelloWorldService" targetNamespace="http://org.jboss.ws/jaxws/cxf/jms"
  xmlns:ns1="http://schemas.xmlsoap.org/soap/http"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://org.jboss.ws/jaxws/cxf/jms"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soapjms="http://www.w3.org/2010/soapjms/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <wsdl:types>
<xs:schema elementFormDefault="unqualified" targetNamespace="http://org.jboss.ws/jaxws/cxf/jms"
  version="1.0" xmlns:tns="http://org.jboss.ws/jaxws/cxf/jms"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="echo" type="tns:echo"/>
<xs:element name="echoResponse" type="tns:echoResponse"/>
<xs:complexType name="echo">
  <xs:sequence>
    <xs:element minOccurs="0" name="arg0" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="echoResponse">
  <xs:sequence>
    <xs:element minOccurs="0" name="return" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
</xs:schema>
</wsdl:types>
<wsdl:message name="echoResponse">
  <wsdl:part element="tns:echoResponse" name="parameters">
</wsdl:part>
</wsdl:message>
<wsdl:message name="echo">
  <wsdl:part element="tns:echo" name="parameters">
</wsdl:part>
</wsdl:message>
<wsdl:portType name="HelloWorld">
  <wsdl:operation name="echo">
    <wsdl:input message="tns:echo" name="echo">
</wsdl:input>
    <wsdl:output message="tns:echoResponse" name="echoResponse">
</wsdl:output>
  </wsdl:operation>
</wsdl:portType>
<wsdl:binding name="HelloWorldServiceSoapBinding" type="tns:HelloWorld">
  <soap:binding style="document" transport="http://www.w3.org/2010/soapjms/">
<wsdl:operation name="echo">
  <soap:operation soapAction="" style="document"/>
  <wsdl:input name="echo">
```



```
<soap:body use="literal"/>
</wsdl:input>
<wsdl:output name="echoResponse">
  <soap:body use="literal"/>
</wsdl:output>
</wsdl:operation>
</wsdl:binding>
<wsdl:service name="HelloWorldService">


<soapjms:jndiConnectionFactoryName>java:jms/RemoteConnectionFactory</soapjms:jndiConnectionFactoryName>
<soapjms:jndiInitialContextFactory>org.jboss.naming.remote.client.InitialContextFactory</soapjms:jndiInitialContextFactory>
<soapjms:jndiURL>http-remoting://myhost:8080</soapjms:jndiURL>
  <wsdl:port binding="tns:HelloWorldServiceSoapBinding" name="HelloWorldImplPort">
    <soap:address location="jms:queue:testQueue"/>
  </wsdl:port>
</wsdl:service>
<wsdl:service name="HelloWorldServiceLocal">

<soapjms:jndiConnectionFactoryName>java:/ConnectionFactory</soapjms:jndiConnectionFactoryName>
  <wsdl:port binding="tns:HelloWorldServiceSoapBinding" name="HelloWorldImplPort">
    <soap:address location="jms:queue:testQueue"/>
  </wsdl:port>
</wsdl:service>
</wsdl:definitions>
```

 The *HelloWorldImplPort* here is meant for using the *testQueue* that has to be created before deploying the endpoint.

At the time of writing, *java:/ConnectionFactory* is the default connection factory JNDI location on WildFly

For allowing remote JNDI lookup of the connection factory, a specific service (*HelloWorldService*) for remote clients is added to the WSDL. The *java:jms/RemoteConnectionFactory* is the JNDI location of the same connection factory mentioned above, except it's exposed for remote lookup. The *soapjms:jndiInitialContextFactory* and *soap:jmsjndiURL* complete the remote connection configuration, specifying the initial context factory class to use and the JNDI registry address.

 Have a look at the application server domain for finding out the configured connection factory JNDI locations.


The endpoint implementation is a basic JAX-WS POJO using *@WebService* annotation to refer to the consumed contract:



```
package org.jboss.test.ws.jaxws.cxf.jms;

import javax.jws.WebService;


@WebService
(
    portName = "HelloWorldImplPort",
    serviceName = "HelloWorldServiceLocal",
    wsdlLocation = "META-INF/wsdl/HelloWorldService.wsdl",
    endpointInterface = "org.jboss.test.ws.jaxws.cxf.jms.HelloWorld",
    targetNamespace = "http://org.jboss.ws/jaxws/cxf/jms"
)
public class HelloWorldImpl implements HelloWorld
{
    public String echo(String input)
    {
        return input;
    }
}
```

 The endpoint implementation references the HelloWorldServiceLocal wsdl service, so that the local JNDI connection factory location is used for starting the endpoint on server side.

That's pretty much all. We just need to package the generated service endpoint interface, the endpoint implementation and the WSDL file in a *jar* archive and deploy it:

```
alessio@inuyasha /dati/jboss/ws/stack/cxf/trunk $ jar -tvf
./modules/testsuite/cxf-tests/target/test-libs/jaxws-cxf-jms-only-deployment.jar
 0 Thu Jun 23 15:18:44 CEST 2011 META-INF/
129 Thu Jun 23 15:18:42 CEST 2011 META-INF/MANIFEST.MF
 0 Thu Jun 23 15:18:42 CEST 2011 org/
 0 Thu Jun 23 15:18:42 CEST 2011 org/jboss/
 0 Thu Jun 23 15:18:42 CEST 2011 org/jboss/test/
 0 Thu Jun 23 15:18:42 CEST 2011 org/jboss/test/ws/
 0 Thu Jun 23 15:18:42 CEST 2011 org/jboss/test/ws/jaxws/
 0 Thu Jun 23 15:18:42 CEST 2011 org/jboss/test/ws/jaxws/cxf/
 0 Thu Jun 23 15:18:42 CEST 2011 org/jboss/test/ws/jaxws/cxf/jms/
313 Thu Jun 23 15:18:42 CEST 2011 org/jboss/test/ws/jaxws/cxf/jms/HelloWorld.class
1173 Thu Jun 23 15:18:42 CEST 2011 org/jboss/test/ws/jaxws/cxf/jms/HelloWorldImpl.class
 0 Thu Jun 23 15:18:40 CEST 2011 META-INF/wsdl/
3074 Thu Jun 23 15:18:40 CEST 2011 META-INF/wsdl/HelloWorldService.wsdl
```



 A dependency on `org.hornetq` module needs to be added in MANIFEST.MF when deploying to WildFly.

```
Manifest-Version: 1.0


Ant-Version: Apache Ant 1.7.1

Created-By: 17.0-b16 (Sun Microsystems Inc.)

Dependencies: org.hornetq
```

A JAX-WS client can interact with the JMS endpoint the usual way:

```
URL wsdlUrl = ...
//start another bus to avoid affecting the one that could already be assigned to the current
thread - optional but highly suggested
Bus bus = BusFactory.newInstance().createBus();
BusFactory.setThreadDefaultBus(bus);
try
{
    QName serviceName = new QName("http://org.jboss.ws/jaxws/cxf/jms", "HelloWorldService");
    Service service = Service.create(wsdlUrl, serviceName);
    HelloWorld proxy = (HelloWorld) service.getPort(new
QName("http://org.jboss.ws/jaxws/cxf/jms", "HelloWorldImplPort"), HelloWorld.class);
    setupProxy(proxy);
    proxy.echo("Hi");
}
finally
{
    bus.shutdown(true);
}
```

 The WSDL location URL needs to be retrieved in a custom way, depending on the client application. Given the endpoint is JMS only, there's no automatically published WSDL contract.

in order for performing the remote invocation (which internally goes through remote JNDI lookup of the connection factory), the calling user credentials need to be set into the Apache CXF JMSConduit:



```
private void setupProxy(HelloWorld proxy) {
    JMSConduit conduit = (JMSConduit)ClientProxy.getClient(proxy).getConduit();
    JNDIConfiguration jndiConfig = conduit.getJmsConfig().getJndiConfig();
    jndiConfig.setConnectionUserName("user");
    jndiConfig.setConnectionPassword("password");
    Properties props = conduit.getJmsConfig().getJndiTemplate().getEnvironment();
    props.put(Context.SECURITY_PRINCIPAL, "user");
    props.put(Context.SECURITY_CREDENTIALS, "password");
}
```



Have a look at the WildFly domain and messaging configuration for finding out the actual security requirements. At the time of writing, a user with `guest` role is required and that's internally checked using the `other` security domain.

Of course once the endpoint is exposed over JMS transport, any plain JMS client can also be used to send messages to the webservice endpoint. You can have a look at the SOAP over JMS spec details and code the client similarly to

```
Properties env = new Properties();
env.put(Context.INITIAL_CONTEXT_FACTORY,
"org.jboss.naming.remote.client.InitialContextFactory");
env.put(Context.PROVIDER_URL, "http-remoting://myhost:8080");
env.put(Context.SECURITY_PRINCIPAL, "user");
env.put(Context.SECURITY_CREDENTIALS, "password");
InitialContext context = new InitialContext(env);
QueueConnectionFactory connectionFactory =
(QueueConnectionFactory)context.lookup("jms/RemoteConnectionFactory");
Queue reqQueue = (Queue)context.lookup("jms/queue/test");
Queue resQueue = (Queue)context.lookup("jms/queue/test");
QueueConnection con = connectionFactory.createQueueConnection("user", "password");
QueueSession session = con.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
QueueReceiver receiver = session.createReceiver(resQueue);
ResponseListener responseListener = new ResponseListener(); //a custom response listener...
receiver.setMessageListener(responseListener);
con.start();
TextMessage message = session.createTextMessage(reqMessage);
message.setJMSReplyTo(resQueue);

//setup SOAP-over-JMS properties...
message.setStringProperty("SOAPJMS_contentType", "text/xml");
message.setStringProperty("SOAPJMS_requestURI", "jms:queue:testQueue");

QueueSender sender = session.createSender(reqQueue);
sender.send(message);
sender.close();

...
```




JMS and HTTP endpoints deployment

In this example we create a deployment containing an endpoint that serves over both HTTP and JMS transports.

We from a WSDL contract such as below (please note we've two binding / portType for the same service):

```
<?xml version='1.0' encoding='UTF-8'?>
<wsdl:definitions name="HelloWorldService" targetNamespace="http://org.jboss.ws/jaxws/cxf/jms"
  xmlns:ns1="http://schemas.xmlsoap.org/soap/http"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://org.jboss.ws/jaxws/cxf/jms"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soapjms="http://www.w3.org/2010/soapjms/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <wsdl:types>
<xs:schema elementFormDefault="unqualified" targetNamespace="http://org.jboss.ws/jaxws/cxf/jms"
version="1.0"
  xmlns:tns="http://org.jboss.ws/jaxws/cxf/jms" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<xs:element name="echo" type="tns:echo"/>
<xs:element name="echoResponse" type="tns:echoResponse"/>
<xs:complexType name="echo">
  <xs:sequence>
    <xs:element minOccurs="0" name="arg0" type="xsd:string"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="echoResponse">
  <xs:sequence>
    <xs:element minOccurs="0" name="return" type="xsd:string"/>
  </xs:sequence>
</xs:complexType>
</xs:schema>
</wsdl:types>
<wsdl:message name="echoResponse">
  <wsdl:part element="tns:echoResponse" name="parameters">
</wsdl:part>
</wsdl:message>
<wsdl:message name="echo">
  <wsdl:part element="tns:echo" name="parameters">
</wsdl:part>
</wsdl:message>
<wsdl:portType name="HelloWorld">
  <wsdl:operation name="echo">
    <wsdl:input message="tns:echo" name="echo">
</wsdl:input>
    <wsdl:output message="tns:echoResponse" name="echoResponse">
</wsdl:output>
  </wsdl:operation>
</wsdl:portType>
<wsdl:binding name="HelloWorldServiceSoapBinding" type="tns:HelloWorld">
  <soap:binding style="document" transport="http://www.w3.org/2010/soapjms/">
<wsdl:operation name="echo">
  <soap:operation soapAction="" style="document"/>
  <wsdl:input name="echo">
```



```
<soap:body use="literal"/>
</wsdl:input>
<wsdl:output name="echoResponse">
  <soap:body use="literal"/>
</wsdl:output>
</wsdl:operation>
</wsdl:binding>
<wsdl:binding name="HttpHelloWorldServiceSoapBinding" type="tns:HelloWorld">
  <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
  <wsdl:operation name="echo">
    <soap:operation soapAction="" style="document"/>
    <wsdl:input name="echo">
      <soap:body use="literal"/>
    </wsdl:input>
    <wsdl:output name="echoResponse">
      <soap:body use="literal"/>
    </wsdl:output>
  </wsdl:operation>
</wsdl:binding>
<wsdl:service name="HelloWorldService">

<soapjms:jndiConnectionFactoryName>java:jms/RemoteConnectionFactory</soapjms:jndiConnectionFactoryName>
<soapjms:jndiInitialContextFactory>org.jboss.naming.remote.client.InitialContextFactory</soapjms:jndiInitialContextFactory>
<soapjms:jndiURL>http-remoting://localhost:8080</soapjms:jndiURL>
  <wsdl:port binding="tns:HelloWorldServiceSoapBinding" name="HelloWorldImplPort">
    <soap:address location="jms:queue:testQueue"/>
  </wsdl:port>
  <wsdl:port binding="tns:HttpHelloWorldServiceSoapBinding" name="HttpHelloWorldImplPort">
    <soap:address location="http://localhost:8080/jaxws-cxf-jms-http-deployment"/>
  </wsdl:port>
</wsdl:service>
<wsdl:service name="HelloWorldServiceLocal">

<soapjms:jndiConnectionFactoryName>java:/ConnectionFactory</soapjms:jndiConnectionFactoryName>
  <wsdl:port binding="tns:HelloWorldServiceSoapBinding" name="HelloWorldImplPort">
    <soap:address location="jms:queue:testQueue"/>
  </wsdl:port>
</wsdl:definitions>
```

The same considerations of the previous example regarding the JMS queue and JNDI connection factory still apply.

Here we can implement the endpoint in multiple ways, either with a common implementation class that's extended by the JMS and HTTP ones, or keep the two implementation classes independent and just have them implement the same service endpoint interface:



```
package org.jboss.test.ws.jaxws.cxf.jms_http;

import javax.jws.WebService;

@WebService
(
    portName = "HelloWorldImplPort",
    serviceName = "HelloWorldServiceLocal",
    wsdlLocation = "WEB-INF/wsdl/HelloWorldService.wsdl",
    endpointInterface = "org.jboss.test.ws.jaxws.cxf.jms_http.HelloWorld",
    targetNamespace = "http://org.jboss.ws/jaxws/cxf/jms"
)
public class HelloWorldImpl implements HelloWorld
{
    public String echo(String input)
    {
        System.out.println("input: " + input);
        return input;
    }
}
```

```
package org.jboss.test.ws.jaxws.cxf.jms_http;

import javax.jws.WebService;

@WebService
(
    portName = "HttpHelloWorldImplPort",
    serviceName = "HelloWorldService",
    wsdlLocation = "WEB-INF/wsdl/HelloWorldService.wsdl",
    endpointInterface = "org.jboss.test.ws.jaxws.cxf.jms_http.HelloWorld",
    targetNamespace = "http://org.jboss.ws/jaxws/cxf/jms"
)
public class HttpHelloWorldImpl implements HelloWorld
{
    public String echo(String input)
    {
        System.out.println("input (http): " + input);
        return "(http) " + input;
    }
}
```

Both classes are packaged together the service endpoint interface and the WSDL file in a *war* archive:



```
alessio@inuyasha /dati/jboss/ws/stack/cxf/trunk $ jar -tvf
./modules/testsuite/cxf-spring-tests/target/test-libs/jaxws-cxf-jms-http-deployment.war
 0 Thu Jun 23 15:18:44 CEST 2011 META-INF/
129 Thu Jun 23 15:18:42 CEST 2011 META-INF/MANIFEST.MF
 0 Thu Jun 23 15:18:44 CEST 2011 WEB-INF/
569 Thu Jun 23 15:18:40 CEST 2011 WEB-INF/web.xml
 0 Thu Jun 23 15:18:44 CEST 2011 WEB-INF/classes/
 0 Thu Jun 23 15:18:42 CEST 2011 WEB-INF/classes/org/
 0 Thu Jun 23 15:18:42 CEST 2011 WEB-INF/classes/org/jboss/
 0 Thu Jun 23 15:18:42 CEST 2011 WEB-INF/classes/org/jboss/test/
 0 Thu Jun 23 15:18:42 CEST 2011 WEB-INF/classes/org/jboss/test/ws/
 0 Thu Jun 23 15:18:42 CEST 2011 WEB-INF/classes/org/jboss/test/ws/jaxws/
 0 Thu Jun 23 15:18:42 CEST 2011 WEB-INF/classes/org/jboss/test/ws/jaxws/cxf/
 0 Thu Jun 23 15:18:42 CEST 2011 WEB-INF/classes/org/jboss/test/ws/jaxws/cxf/jms_http/
318 Thu Jun 23 15:18:42 CEST 2011
WEB-INF/classes/org/jboss/test/ws/jaxws/cxf/jms_http/HelloWorld.class
1192 Thu Jun 23 15:18:42 CEST 2011
WEB-INF/classes/org/jboss/test/ws/jaxws/cxf/jms_http/HelloWorldImpl.class
1246 Thu Jun 23 15:18:42 CEST 2011
WEB-INF/classes/org/jboss/test/ws/jaxws/cxf/jms_http/HttpHelloWorldImpl.class
 0 Thu Jun 23 15:18:40 CEST 2011 WEB-INF/wsdl/
3068 Thu Jun 23 15:18:40 CEST 2011 WEB-INF/wsdl/HelloWorldService.wsdl
```

A trivial web.xml descriptor is also included to trigger the HTTP endpoint publish:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
  version="2.4">
  <servlet>
    <servlet-name>EndpointServlet</servlet-name>
    <servlet-class>org.jboss.test.ws.jaxws.cxf.jms_http.HttpHelloWorldImpl</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>EndpointServlet</servlet-name>
    <url-pattern>*</url-pattern>
  </servlet-mapping>
</web-app>
```



Here too the MANIFEST.MF needs to declare a dependency on *org.hornetq* module when deploying to WildFly.

Finally, the JAX-WS client can interact with both JMS and HTTP endpoints as usual:



```
//start another bus to avoid affecting the one that could already be assigned to current thread
- optional but highly suggested
Bus bus = BusFactory.newInstance().createBus();
BusFactory.setThreadDefaultBus(bus);
try
{
    QName serviceName = new QName("http://org.jboss.ws/jaxws/cxf/jms", "HelloWorldService");
    Service service = Service.create(wsdlUrl, serviceName);

    //JMS test
    HelloWorld proxy = (HelloWorld) service.getPort(new
QName("http://org.jboss.ws/jaxws/cxf/jms", "HelloWorldImplPort"), HelloWorld.class);
    setupProxy(proxy);
    proxy.echo("Hi");
    //HTTP test
    HelloWorld httpProxy = (HelloWorld) service.getPort(new
QName("http://org.jboss.ws/jaxws/cxf/jms", "HttpHelloWorldImplPort"), HelloWorld.class);
    httpProxy.echo("Hi");
}
finally
{
    bus.shutdown(true);
}
```

Use of Endpoint.publish() API

An alternative to deploying an archive containing JMS endpoints is in starting them directly using the JAX-WS `Endpoint.publish(..)` API.

That's as easy as doing:

```
Object implementor = new HelloWorldImpl();
Endpoint ep = Endpoint.publish("jms:queue:testQueue", implementor);
try
{
    //use or let others use the endpoint
}
finally
{
    ep.stop();
}
```

where `HelloWorldImpl` is a POJO endpoint implementation referencing a JMS *port* in a given WSDL contract, as explained in the previous examples.

The main difference among the deployment approach is in the direct control and responsibility over the endpoint lifecycle (*start/publish* and *stop*).



35.3.16 HTTP Proxy

The HTTP Proxy related functionalities of JBoss Web Services are provided by the Apache CXF http transport layer.

The suggested configuration mechanism when running JBoss Web Services is explained below; for further information please refer to the [Apache CXF documentation](#).

Configuration

The HTTP proxy configuration for a given JAX-WS client can be set in the following ways:

- through the `http.proxyHost` and `http.proxyPort` system properties, or
- leveraging the `org.apache.cxf.transport.http.HTTPConduit` options

The former is a JVM level configuration; for instance, assuming the http proxy is currently running at <http://localhost:9934>, here is the setup:

```
System.getProperties().setProperty("http.proxyHost", "localhost");
System.getProperties().setProperty("http.proxyPort", 9934);
```

The latter is a client stub/port level configuration: the setup is performed on the `HTTPConduit` object that's part of the Apache CXF `Client` abstraction.

```
import org.apache.cxf.configuration.security.ProxyAuthorizationPolicy;
import org.apache.cxf.endpoint.Client;
import org.apache.cxf.frontend.ClientProxy;
import org.apache.cxf.transport.http.HTTPConduit;
import org.apache.cxf.transports.http.configuration.HTTPClientPolicy;
import org.apache.cxf.transports.http.configuration.ProxyServerType;
...

Service service = Service.create(wsdlURL, new QName("http://org.jboss.ws/jaxws/cxf/httpproxy",
"HelloWorldService"));
HelloWorld port = (HelloWorld) service.getPort(new
QName("http://org.jboss.ws/jaxws/cxf/httpproxy", "HelloWorldImplPort"), HelloWorld.class);

Client client = ClientProxy.getClient(port);
HTTPConduit conduit = (HTTPConduit)client.getConduit();
ProxyAuthorizationPolicy policy = new ProxyAuthorizationPolicy();
policy.setAuthorizationType("Basic");
policy.setUsername(PROXY_USER);
policy.setPassword(PROXY_PWD);
conduit.setProxyAuthorization(policy);

port.echo("Foo");
```



The `ProxyAuthorizationPolicy` also allows for setting the authorization type as well as the username / password to be used.

Speaking of authorization and authentication, please note that the JDK already features the `java.net.Authenticator` facility, which is used whenever opening a connection to a given URL requiring a http proxy. Users might want to set a custom `Authenticator` for instance when needing to read WSDL contracts before actually calling into the JBoss Web Services / Apache CXF code; here is an example:

```
import java.net.Authenticator;
import java.net.PasswordAuthentication;
...
public class ProxyAuthenticator extends Authenticator
{
    private String user, password;

    public ProxyAuthenticator(String user, String password)
    {
        this.user = user;
        this.password = password;
    }

    protected PasswordAuthentication getPasswordAuthentication()
    {
        return new PasswordAuthentication(user, password.toCharArray());
    }
}

...

Authenticator.setDefault(new ProxyAuthenticator(PROXY_USER, PROXY_PWD));
```



35.3.17 Discovery

Apache CXF includes support for *Web Services Dynamic Discovery* ([WS-Discovery](#)), which is a protocol to enable dynamic discovery of services available on the local network. The protocol implies using a UDP based multicast transport to announce new services and probe for existing services. A managed mode where a discovery proxy is used to reduce the amount of required multicast traffic is also covered by the protocol.

JBossWS integrates the *WS-Discovery* [functionalities](#) provided by Apache CXF into the application server.

Enabling WS-Discovery

Apache CXF enables *WS-Discovery* depending on the availability of its runtime component; given that's always shipped in the application server, JBossWS integration requires using the `cxf.ws-discovery.enabled` [property](#) usage for enabling *WS-Discovery* for a given deployment. By default *WS-Discovery* is disabled on the application server. Below is an example of `jboss-webservices.xml` descriptor to be used for enabling *WS-Discovery*:

```
<webservices xmlns="http://www.jboss.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  version="1.2" xsi:schemaLocation="http://www.jboss.com/xml/ns/javaee">

  <property>
    <name>cxf.ws-discovery.enabled</name>
    <value>true</value>
  </property>

</webservices>
```

By default, a *WS-Discovery* service endpoint (SOAP-over-UDP bound) will be started the first time a *WS-Discovery* enabled deployment is processed on the application server. Every ws endpoint belonging to *WS-Discovery* enabled deployments will be automatically registered into such a *WS-Discovery* service endpoint (Hello messages). The service will reply to *Probe* and *Resolve* messages received on UDP port 3702 (including multicast messages sent to IPv4 address 239.255.255.250, as per [specification](#)). Endpoints will eventually be automatically unregistered using *Bye* messages upon undeployment.

Probing services

Apache CXF comes with a *WS-Discovery* API that can be used to probe / resolve services. When running in-container, a JBoss module [dependency](#) to the `org.apache.cxf.impl` module is to be set to have access to *WS-Discovery* client functionalities.

The `org.apache.cxf.ws.discovery.WSDiscoveryClient` class provides the *probe* and *resolve* methods which also accepts filters on scopes. Users can rely on them for locating available endpoints on the network. Please have a look at the JBossWS testsuite which includes a [sample](#) on CXF *WS-Discovery* usage.



35.3.18 Policy

- [Apache CXF WS-Policy support](#)
 - [Contract-first approach](#)
 - [Code-first approach](#)
- [JBossWS additions](#)
 - [Policy sets](#)

Apache CXF WS-Policy support

JBossWS policy support rely on the Apache CXF WS-Policy framework, which is compliant with the [Web Services Policy 1.5 - Framework](#) and [Web Services Policy 1.5 - Attachment](#) specifications.

Users can work with policies in different ways:

- by adding policy assertions to wsdl contracts and letting the runtime consume them and behave accordingly;
- by specifying endpoint policy attachments using either CXF annotations or features.

Of course users can also make direct use of the Apache CXF policy framework, [defining custom assertions](#), etc.

Finally, JBossWS provides some additional annotations for simplified policy attachment.

Contract-first approach

WS-Policies can be attached and referenced in wsdl elements (the specifications describe all possible alternatives). Apache CXF automatically recognizes, reads and uses policies defined in the wsdl.

Users should hence develop endpoints using the *contract-first* approach, that is explicitly providing the contract for their services. Here is a excerpt taken from a wsdl including a WS-Addressing policy:

```
<wsdl:definitions name="Foo" targetNamespace="http://ws.jboss.org/foo"
...
<wsdl:service name="FooService">
  <wsdl:port binding="tns:FooBinding" name="FooPort">
    <soap:address location="http://localhost:80800/foo"/>
    <wsp:Policy xmlns:wsp="http://www.w3.org/ns/ws-policy">
      <wsam:Addressing xmlns:wsam="http://www.w3.org/2007/02/addressing/metadata">
        <wsp:Policy/>
      </wsam:Addressing>
    </wsp:Policy>
  </wsdl:port>
</wsdl:service>
</wsdl:definitions>
```

Of course, CXF also acts upon policies specified in wsdl documents consumed on client side.



Code-first approach

For those preferring code-first (java-first) endpoint development, Apache CXF comes with `org.apache.cxf.annotations.Policy` and `org.apache.cxf.annotations.Policies` annotations to be used for attaching policy fragments to the wsdl generated at deploy time.

Here is an example of a code-first endpoint including `@Policy` annotation:

```
import javax.jws.WebService;
import org.apache.cxf.annotations.Policy;

@WebService(portName = "MyServicePort",
            serviceName = "MyService",
            name = "MyServiceIface",
            targetNamespace = "http://www.jboss.org/jbossws/foo")
@Policy(placement = Policy.Placement.BINDING, uri = "JavaFirstPolicy.xml")
public class MyServiceImpl {
    public String sayHello() {
        return "Hello World!";
    }
}
```

The referenced descriptor is to be added to the deployment and will include the policy to be attached; the attachment position in the contracts is defined through the `placement` attribute. Here is a descriptor example:

```
<?xml version="1.0" encoding="UTF-8" ?>
<wsp:Policy wsu:Id="MyPolicy" xmlns:wsp="http://www.w3.org/ns/ws-policy"

xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd">
  <wsp:ExactlyOne>
    <wsp:All>
      <sp:SupportingTokens
xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy">
        <wsp:Policy>
          <sp:UsernameToken
sp:IncludeToken="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy/IncludeToken/AlwaysToRecipient"
<wsp:Policy>
            <sp:WssUsernameToken10/>
          </wsp:Policy>
        </sp:UsernameToken>
      </wsp:Policy>
    </sp:SupportingTokens>
  </wsp:All>
</wsp:ExactlyOne>
</wsp:Policy>
```



JBossWS additions

Policy sets

Both approaches above require users to actually write their policies' assertions; while this offer great flexibility and control of the actual contract, providing the assertions might end up being quite a challenging task for complex policies. For this reason, the JBossWS integration provides *policy sets*, which are basically pre-defined groups of policy assertions corresponding to well known / common needs. Each set has a label allowing users to specify it in the `@org.jboss.ws.api.annotation.PolicySets` annotation to have the policy assertions for that set attached to the annotated endpoint. Multiple labels can also be specified. Here is an example of the `@PolicySets` annotation on a service endpoint interface:

```
import javax.jws.WebService;
import org.jboss.ws.api.annotation.PolicySets;

@WebService(name = "EndpointTwo", targetNamespace = "http://org.jboss.ws.jaxws.cxf/jbws3648")
@PolicySets({"WS-RM_Policy_spec_example", "WS-SP-EX223_WSS11_Anonymous_X509_Sign_Encrypt",
"WS-Addressing"})
public interface EndpointTwo
{
    String echo(String input);
}
```

The three sets specified in `@PolicySets` will cause the wsdl generated for the endpoint having this interface to be enriched with some policy assertions for WS-RM, WS-Security and WS-Addressing.

The labels' list of known sets is stored in the `META-INF/policies/org.jboss.wsf.stack.cxf.extensions.policy.PolicyAttachmentStore` file within the `jbossws-cxf-client.jar` (`org.jboss.ws.cxf:jbossws-cxf-client` maven artifact). Actual policy fragments for each set are also stored in the same artifact at `META-INF/policies/<set-label>-<attachment-position>.xml`.

Here is a list of the available policy sets:

Label	Description
WS-Addressing	Basic WS-Addressing policy
WS-RM_Policy_spec_example	The basic WS-RM policy example in the WS-RM specification



WS-SP-EX2121_SSL_UT_Supporting_Token	The group of policy assertions used in the section 2.1.2.1 example of the WS-Security Policy Examples 1.0 specification
WS-SP-EX2123_WSS10_UT_Mutual_Auth_X509_Sign_Encrypt	The group of policy assertions used in the section 2.1.3 example of the WS-Security Policy Examples 1.0 specification
WS-SP-EX2124_WSS11_User_Name_Cert_Sign_Encrypt	The group of policy assertions used in the section 2.1.4 example of the WS-Security Policy Examples 1.0 specification
WS-SP-EX221_WSS10_Mutual_Auth_X509_Sign_Encrypt	The group of policy assertions used in the section 2.2.1 example of the WS-Security Policy Examples 1.0 specification
WS-SP-EX222_WSS10_Mutual_Auth_X509_Sign_Encrypt	The group of policy assertions used in the section 2.2.2 example of the WS-Security Policy Examples 1.0 specification
WS-SP-EX223_WSS11_Anonymous_X509_Sign_Encrypt	The group of policy assertions used in the section 2.2.3 example of the WS-Security Policy Examples 1.0 specification



WS-SP-EX224_WSS11_Mutual_Auth_X509_Sign_Encrypt	The group of policy assertions used in the section 2.2.4 example of the WS-Security Policy Examples 1.0 specification
AsymmetricBinding_X509v1_TripleDesRsa15_EncryptBeforeSigning_ProtectTokens	A WS-Security policy for asymmetric binding (encrypt before signing) using X.509v1 tokens, 3DES + RSA 1.5 algorithms and with token protections enabled
AsymmetricBinding_X509v1_GCM256OAEP_ProtectTokens	The same as before, but using custom Apache CXF algorithm suite including GCM 256 + RSA OAEP algorithms

⊖ Always verify the contents of the generated wsdl contract, as policy sets are potentially subject to updates between JBossWS releases. This is especially important when dealing with security related policies; the provided sets are to be considered as convenient configuration options only; users remain responsible for the policies in their contracts.

✔ The `org.jboss.wsf.stack.cxf.extensions.policy.Constants` interface has convenient String constants for the available policy set labels.

✔ If you feel a new set should be added, just propose it by writing the user forum!



35.3.19 Published WSDL customization

- [Endpoint address rewrite](#)
- [System property references](#)

Endpoint address rewrite

JBossWS supports the rewrite of the `<soap:address>` element of endpoints published in WSDL contracts. This feature is useful for controlling the server address that is advertised to clients for each endpoint. The rewrite mechanism is configured at server level through a set of elements in the webservices subsystem of the WildFly management model. Please refer to the container documentation for details on the options supported in the selected container version. Below is a list of the elements available in the latest WildFly sources:



Name	Type	Description
modify-wsdl-address	boolean	<p>This boolean enables and disables the address rewrite functionality. When <code>modify-wsdl-address</code> is set to <code>true</code> and the content of <code><soap:address></code> is a valid URL, JBossWS will rewrite the URL using the values of <code>wsdl-host</code> and <code>wsdl-port</code> or <code>wsdl-secure-port</code>.</p> <p>When <code>modify-wsdl-address</code> is set to <code>false</code> and the content of <code><soap:address></code> is a valid URL, JBossWS will not rewrite the URL. The <code><soap:address></code> URL will be used.</p> <p>When the content of <code><soap:address></code> is not a valid URL, JBossWS will rewrite it no matter what the setting of <code>modify-wsdl-address</code>.</p> <p>If <code>modify-wsdl-address</code> is set to <code>true</code> and <code>wsdl-host</code> is not defined or explicitly set to <code>'jbossws.undefined.host'</code> the content of <code><soap:address></code> URL is use. JBossWS uses the requester's host when rewriting the <code><soap:address></code></p> <p>When <code>modify-wsdl-address</code> is not defined JBossWS uses a default value of <code>true</code>.</p>
wsdl-host	string	<p>The hostname / IP address to be used for rewriting <code><soap:address></code>. If <code>wsdl-host</code> is set to <code>jbossws.undefined.host</code>, JBossWS uses the requester's host when rewriting the <code><soap:address></code></p> <p>When <code>wsdl-host</code> is not defined JBossWS uses a default value of <code>'jbossws.undefined.host'</code>.</p>
wsdl-port	int	<p>Set this property to explicitly define the HTTP port that will be used for rewriting the SOAP address.</p> <p>Otherwise the HTTP port will be identified by querying the list of installed HTTP connectors.</p>
wsdl-secure-port	int	<p>Set this property to explicitly define the HTTPS port that will be used for rewriting the SOAP address.</p> <p>Otherwise the HTTPS port will be identified by querying the list of installed HTTPS connectors.</p>
wsdl-uri-scheme	string	<p>This property explicitly sets the URI scheme to use for rewriting <code><soap:address></code>. Valid values are <code>http</code> and <code>https</code>. This configuration overrides scheme computed by processing the endpoint (even if a transport guarantee is specified). The provided values for <code>wsdl-port</code> and <code>wsdl-secure-port</code> (or their default values) are used depending on specified scheme.</p>
wsdl-path-rewrite-rule	string	<p>This string defines a SED substitution command (e.g., <code>'s/regexp/replacement/g'</code>) that JBossWS executes against the path component of each <code><soap:address></code> URL published from the server.</p> <p>When <code>wsdl-path-rewrite-rule</code> is not defined, JBossWS retains the original path component of each <code><soap:address></code> URL.</p> <p>When <code>'modify-wsdl-address'</code> is set to <code>"false"</code> this element is ignored.</p>



Additionally, users can override the server level configuration by requesting a specific rewrite behavior for a given endpoint deployment. That is achieved by setting one of the following properties within a *jboss-webservices.xml* descriptor:

Property	Corresponding server option
<code>wsdl.soapAddress.rewrite.modify-wsdl-address</code>	<code>modify-wsdl-address</code>
<code>wsdl.soapAddress.rewrite.wsdl-host</code>	<code>wsdl-host</code>
<code>wsdl.soapAddress.rewrite.wsdl-port</code>	<code>wsdl-port</code>
<code>wsdl.soapAddress.rewrite.wsdl-secure-port</code>	<code>wsdl-secure-port</code>
<code>wsdl.soapAddress.rewrite.wsdl-path-rewrite-rule</code>	<code>wsdl-path-rewrite-rule</code>
<code>wsdl.soapAddress.rewrite.wsdl-uri-scheme</code>	<code>wsdl-uri-scheme</code>

Here is an example of partial overriding of the default configuration for a specific deployment:

```
<?xml version="1.1" encoding="UTF-8"?>
<webservices version="1.2"
  xmlns="http://www.jboss.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.jboss.com/xml/ns/javaee">
  <property>
    <name>wsdl.soapAddress.rewrite.wsdl-uri-scheme</name>
    <value>https</value>
  </property>
  <property>
    <name>wsdl.soapAddress.rewrite.wsdl-host</name>
    <value>foo</value>
  </property>
</webservices>
```




System property references

System property references wrapped within "@" characters are expanded when found in WSDL attribute and element values. This allows for instance including multiple WS-Policy declarations in the contract and selecting the policy to use depending on a server wide system property; here is an example:

```
<wsdl:definitions ...>
  ...
  <wsdl:binding name="ServiceOneSoapBinding" type="tns:EndpointOne">
    ...
    <wsp:PolicyReference URI="#@org.jboss.wsf.test.JBWS3628TestCase.policy@" />
    <wsdl:operation name="echo">
      ...
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="ServiceOne">
    <wsdl:port binding="tns:ServiceOneSoapBinding" name="EndpointOnePort">
      <soap:address location="http://localhost:8080/jaxws-cxf-jbws3628/ServiceOne"/>
    </wsdl:port>
  </wsdl:service>

  <wsp:Policy
xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
xmlns:wsp="http://www.w3.org/ns/ws-policy" wsu:Id="WS-RM_Policy">
  <wsrmp:RMAssertion xmlns:wsrmp="http://schemas.xmlsoap.org/ws/2005/02/rm/policy">
    ...
  </wsrmp:RMAssertion>
</wsp:Policy>

  <wsp:Policy
xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
xmlns:wsp="http://www.w3.org/ns/ws-policy"
  xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata" wsu:Id="WS-Addressing_policy">
    <wsam:Addressing>
      <wsp:Policy/>
    </wsam:Addressing>
  </wsp:Policy>
</wsdl:definitions>
```

If the ***org.jboss.wsf.test.JBWS3628TestCase.policy*** system property is defined and set to "***WS-Addressing_policy***", WS-Addressing will be enabled for the endpoint defined by the contract above.



35.4 JBoss Modules and WS applications


- [Setting module dependencies](#)
 - [Using MANIFEST.MF](#)
 - [Using JAXB](#)
 - [Using Apache CXF](#)
 - [Client side WS aggregation module](#)
 - [Annotation scanning](#)
 - [Using jboss-deployment-descriptor.xml](#)

The JBoss Web Services functionalities are provided by a given set of modules / libraries installed on WildFly, which are organized into JBoss Modules modules. In particular the *org.jboss.as.webservices.** and *org.jboss.ws.** modules belong to the JBossWS - WildFly integration. Users should not need to change anything in them.

While users are of course allowed to provide their own modules for their custom needs, below is a brief collection of suggestions and hints around modules and webservices development on WildFly.

35.4.1 Setting module dependencies

On WildFly the user deployment classloader does not have any visibility over JBoss internals; so for instance you can't *directly* use JBossWS *implementation* classes unless you explicitly set a dependency to the corresponding module. As a consequence, users need to declare the module dependencies they want to be added to their deployment.

 The JBoss Web Services APIs are always available by default whenever the webservices subsystem is available on AS7. So users just use them, no need for explicit dependencies declaration for those modules.


Using MANIFEST.MF

The convenient method for configuring deployment dependencies is adding them into the MANIFEST.MF file:

```
Manifest-Version: 1.0
Dependencies: org.jboss.ws.cxf.jbossws-cxf-client services export,foo.bar
```

Here above *org.jboss.ws.cxf.jbossws-cxf-client* and *foo.bar* are the modules you want to set dependencies to; *services* tells the modules framework that you want to also import *META-INF/services/..* declarations from the dependency, while *export* exports the classes from the module to any other module that might be depending on the module implicitly created for your deployment.



 When using annotations on your endpoints / handlers such as the Apache CXF ones (`@InInterceptor`, `@GZIP`, ...) remember to add the proper module dependency in your manifest. Otherwise your annotations are not picked up and added to the annotation index by WildFly, resulting in them being completely and silently ignored.

Using JAXB

In order for successfully directly using JAXB contexts, etc. in your client or endpoint running in-container, you need to properly setup a JAXB implementation; that is performed setting the following dependency:

```
Dependencies: com.sun.xml.bind services export
```

Using Apache CXF

In order for using Apache CXF APIs and implementation classes you need to add a dependency to the `org.apache.cxf` (API) module and / or `org.apache.cxf.impl` (implementation) module:

```
Dependencies: org.apache.cxf services
```


However, please note that would not come with any JBossWS-CXF customizations nor additional extensions. For this reason, and generally speaking for simplifying user configuration, a client side aggregation module is available with all the WS dependencies users might need.


Client side WS aggregation module

Whenever you simply want to use all the JBoss Web Services feature/functionality, you can set a dependency to the convenient client module.

```
Dependencies: org.jboss.ws.cxf.jbossws-cxf-client services
```

Please note the `services` option above: that's strictly required in order for you to get the JBossWS-CXF version of classes that are retrieved using the *Service API*, the `org.apache.cxf.Bus` for instance.

 Be careful as issues because of misconfiguration here can be quite hard to track down, because the Apache CXF behaviour would be sensibly different.

 The `services` option is almost always needed when declaring dependencies on `org.jboss.ws.cxf.jbossws-cxf-client` and `org.apache.cxf` modules. The reason for this is in it affecting the loading of classes through the *Service API*, which is what is used to wire most of the JBossWS components as well as all Apache CXF Bus extensions.



Annotation scanning

The application server uses an annotation index for detecting JAX-WS endpoints in user deployments. When declaring WS endpoints whose class belongs to a different module (for instance referring that in the `web.xml` descriptor), be sure to have an `annotations` type dependency in place. Without that, your endpoints would simply be ignored as they won't appear as annotated classes to the webservices subsystem.

```
Dependencies: org.foo annotations
```

Using `jboss-deployment-descriptor.xml`

In some circumstances, the convenient approach of setting module dependencies in `MANIFEST.MF` might not work. An example is the need for importing/exporting specific resources from a given module dependency. Users should hence add a `jboss-deployment-structure.xml` descriptor to their deployment and set module dependencies in it.