JBoss JCA 1.0 Developer's Guide

Connecting your Enterprise Information Systems

Table of Contents

1 About JBoss JCA

The goal of the JBoss JCA project is to provide an implementation of the Java Connector Architecture 1.6 specification.

The specification can be found here: http://www.jcp.org/en/jsr/detail?id=322.

1.1. The team

Jesper Pedersen acts as the lead for the JBoss JCA project. He can be reached at jesper (dot) pedersen (at) jboss (dot) org.

Jeff Zhang is a core developer on the JBoss JCA project. He can be reached at jizhang (at) redhat (dot) com.

Gurkan Erdogdu is a core developer on the JBoss JCA project. He can be reached at gurkanerdogdu (at) yahoo (dot) com.

Stefano Maestri is a core developer on the JBoss JCA project. He can be reached at stefano.maestri (at) javalinux (dot) it.

1.2. Thanks to

Dimitris Andreadis, Carlo de Wolf, Jason Green, Jonathan Halliday, Søren Hilmer, Vicky Kak, Aslak Knutsen, Sacha Labourey, Alexey Loubyansky, Patrick MacDonald, Andrig Miller, Andrew Lee Rubinger, Anil Saldhana and Scott Stark.

And a special thanks goes to Adrian Brock and Ales Justin for their continuous support of this project.

2 Introduction

The Java Connector Architecture (JCA) defines a standard architecture for connecting the Java EE platform to heterogeneous Enterprise Information Systems (EIS). Examples of EISs include Enterprise Resource Planning (ERP), mainframe transaction processing (TP), database and messaging systems.

The connector architecture defines a set of scalable, secure, and transactional mechanisms that enable the integration of EISs with application servers and enterprise applications.

The connector architecture also defines a Common Client Interface (CCI) for EIS access. The CCI defines a client API for interacting with heterogeneous EISs.

The connector architecture enables an EIS vendor to provide a standard resource adapter for its EIS. A resource adapter is a system-level software driver that is used by a Java application to connect to an EIS. The resource adapter plugs into an application server and provides connectivity between the EIS, the application server, and the enterprise application. The resource adapter serves as a protocol adapter that allows any arbitrary EIS communication protocol to be used for connectivity. An application server vendor extends its system once to support the connector architecture and is then assured of seamless connectivity to multiple EISs. Likewise, an EIS vendor provides one standard resource adapter which has the capability to plug in to any application server that supports the connector architecture.

2.1. What's New

The Java Connector Architecture 1.6 specification adds the following major areas:

- Ease of Development: The use of annotations reduces or completely eliminates the need to deal with a deployment descriptor in many cases. The use of annotations also reduces the need to keep the deployment descriptor synchronized with changes to source code.
- Generic work context contract: A generic contract that enables a resource adapter to control the execution context of a Work instance that it has submitted to the application server for execution.
- Security work context: A standard contract that enables a resource adapter to establish security information while submiting a Work instance for execution to a WorkManager and while delivering messages to message endpoints residing in the application server.
- Standalone Container Environment: A defined set of services that makes up a standalone execution environment for resource adapters.

3 Building

3.1. Prerequisites

3.1.1. Java Development Kit (JDK)

You must have one of the following JDKs installed in order to build the project:

- Sun JDK 1.6.x
- Sun JDK 1.7.x

Remember to ensure that "javac" and "java" are in your path (or symlinked).

```
JAVA_HOME=/location/to/javahome
export JAVA_HOME

PATH=$JAVA_HOME/bin:$PATH
export PATH
```

3.1.2. Apache Ant

You must have Apache Ant 1.8.1+ installed on your system.

Remember to ensure that "ant" are in your path (or symlinked).

```
ANT_HOME=/location/to/anthome
export ANT_HOME

PATH=$ANT_HOME/bin:$PATH
export PATH
```

3.1.3. Apache Ivy

The JBoss JCA project uses Apache Ivy for dependency management.

Apache Ivy is automatically downloaded and included in the development environment, so no additional setup is required.

3.1.4. Subversion

You must have Subversion 1.5+ installed on your system.

Remember to ensure that "svn" are in your path (or symlinked).

3.2. Obtaining the source code

3.2.1. Anonymous SVN access

The anonymous SVN repository is located under:

```
svn co http://anonsvn.jboss.org/repos/jbossas/projects/jboss-jca/trunk/ jbossjca-trunk
```

3.2.2. Developer SVN access

The developer SVN repository is located under:

```
svn co https://svn.jboss.org/repos/jbossas/projects/jboss-jca/trunk/ jbossjca-trunk
```

3.2.3. SVN modules

We have the following modules for the project:

- trunk

  The head of development targeting the next upcoming release.

3.3. Compiling the source code

In order to build the JBoss JCA project you execute:

```
ant <target>
```

where target is one of

- jars

  Builds the JAR archives in the distribution.

- test

  Builds the JAR archives in the distribution and runs all the test cases.

- docs

  Builds the API documentation for the project.

- standalone

  Builds the standalone environment using JBoss Microcontainer.

- sjc

  Builds the standalone environment using JBoss JCA/SJC.

- release

  Builds a release of the project.

- clean

  Cleans the project of temporary files.

- clean-cache

  Cleans the Apache Ivy repository.

See the full list of targets in the main build.xml file.

An example to get the JBoss JCA/SJC built and running:

```
ant clean sjc
cd target/sjc/bin
./run.sh
```

4

Issue tracking

4.1. Location

The JIRA issue tracking for the project is located at https://jira.jboss.org/jira/browse/JBJCA
.

4.2. Components

The project is divided into the following components:

Table 4.1. Project components

Component	**Description**
Build	The build environment for the project.
Common	Common interfaces and classes that are shared between multiple components.
Core	The core implementation of the project.
Deployer	The deployers for the project.
Documentation	The documentation (Users Guide / Developers Guide) for the project.
Fungal	The JCA/Fungal kernel.
JDBC	A JDBC resource adapter.
JMS	A generic JMS resource adapter.
Timer	A timer resource adapter.

4.3. Categories

The system contains the following categoies:

Table 4.2. JIRA categories

Category	Description
Feature Request	Request for a feature made by the community.
Bug	Software defect in the project.
Task	Development task created by a member of the team.
Release	Issue which holds informations about a release.
Thirdparty Change	Identifies a thirdparty library dependency.

The other categories in the JIRA installation are not used by this project.

4.4. Life cycle

All issues folows the following life cycle:

Table 4.3. JIRA Lifecycle

Lifecycle	Description
Open	An issue currently not implemented.
Coding in Progress	An issue currently being worked on.
Reopen	An issue that needs further work after it has been resolved.
Resolved	An issue which has been implemented.
Closed	An issue that has been resolved and is include in a release.

Note: Thirdparty issues can't be resolved nor closed during a development cycle. These are resolved and closed as part of the release procedure of the project. The reason for this is that the library in question can receive further updates during the active development cycle.

4.5. Priorities

All issues are assigned one of the following priorities:

Table 4.4. JIRA Priorities

Priority	Description
Blocker	An issue that needs to be fixed before the release.
Critical	An issue that is critical for the release.
Major	The default priority for an issue.
Minor	An issue that is optional for a release.

Priority	**Description**
Trivial	An issue that is optional for a release and have a lower priority than Minor.

5 Testing

5.1. Overall goals

The overall goals of our test environment is to execute tests that ensures that we have full coverage of the JCA specification as well as our implementation.

The full test suite is executed using

```
ant test
```

A single test case can be executed using

```
ant -Dmodule=embedded -Dtest=org.jboss.jca.embedded.unit.ShrinkWrapTestCase one-test
```

where `-Dmodule` specifies which module to execute the test case in. This parameter defaults to `core`. The `-Dtest` parameter specifies the test case itself.

You can also execute all test cases of a single module using

```
ant -Dmodule=embedded module-test
```

where `-Dmodule` specifies which module to execute the test cases in. This parameter defaults to `core`.

The build script does not fail in case of test errors or failure.

You can control the behavior by using the `junit.haltonerror` and `junit.haltonfailure` properties in the main `build.xml` file. Default value for both is `no`.

You can of course change them statically in the `build.xml` file or temporary using `-Djunit.haltonerror=yes`. There are other `jnuit.*` properties defined in the main `build.xml` that can be controlled in the same way.

5.1.1. Specification

The purpose of the specification tests is to test our implementation against the actual specification text.

Each test can only depend on:

- The official Java Connector Architecture API (javax.resource)
- Interfaces and classes in the test suite that extends/implements the official API

The test cases should be created in such a way such that they are easily identified by chapter, section and paragraph. For example:

```
org.jboss.jca.core.spec.chaper10.section3
```

5.1.2. JBoss specific interfaces

The purpose of the JBoss specific interfaces tests is to test our specific interfaces.

Each test can depend on:

- The official Java Connector Architecture API (javax.resource)
- The JBoss JCA specific APIs (org.jboss.jca.xxx.api)
- Interfaces and classes in the test suite that extends/implements these APIs

The test cases lives in a package that have a meaningful name of the component it tests. For example:

```
org.jboss.jca.core.workmanager
```

These test cases can use both the embedded JCA environment or be implemented as standard POJO based JUnit test cases.

5.1.3. JBoss specific implementation

The purpose of the JBoss specific implementation tests is to test our specific implementation. These tests should cover all methods are not exposed through the interface.

Each test can depend on:

- The official Java Connector Architecture API (javax.resource)
- The JBoss JCA specific APIs (org.jboss.jca.xxx.api)
- The JBoss JCA specific implementation (org.jboss.jca.xxx.yyy)
- Interfaces and classes in the test suite

The test cases lives in a package that have a meaningful name of the component it tests. For example:

```
org.jboss.jca.core.workmanager
```

These test cases can use both the embedded JCA environment or be implemented as standard POJO based JUnit test cases.

5.2. Testing principle and style

Our tests follows the Behavior Driven Development (BDD) technique. In BDD you focus on specifying the behaviors of a class and write code (tests) that verify that behavior.

You may be thinking that BDD sounds awfully similar to Test Driven Development (TDD). In some ways they are similar: they both encourage writing the tests first and to provide full coverage of the code. However, TDD doesn't really provide a guide on which kind of tests you should be writing.

BDD provides you with guidance on how to do testing by focusing on what the behavior of a class is supposed to be. We introduce BDD to our testing environment by extending the standard JUnit 4.x test framework with BDD capabilities using assertion and mocking frameworks.

The BDD tests should

- Clearly define `given-when-then` conditions
- The method name defines what is expected: f.ex. shouldReturnFalseIfMethodXIsCalledWithNullString()
- Easy to read the assertions by using Hamcrest Matchers [http://code.google.com/p/hamcrest/]
- Use `given` facts whenever possible to make the test case more readable. It could be the name of the deployed resource adapter, or using the BDD Mockito class [http://mockito.googlecode.com/svn/branches/1.8.0/javadoc/org/mockito/BDDMockito.html] to mock the fact.

We are using two different kind of tests:

- Integration Tests: The goal of these test cases is to validate the whole process of deployment, and interacting with a sub-system by simulating a critical condition.
- Unit Tests: The goal of these test cases is to stress test some internal behaviour by mocking classes to perfectly reproduce conditions to test.

5.2.1. Integration Tests

The integration tests simulate a real condition using a particular deployment artifacts packaged as resource adapters.

The resource adapters are created using either the main build environment or by using ShrinkWrap [http://community.jboss.org/wiki/ShrinkWrap]. Using resource adapters within the test cases will allow you to debug both the resource adapters themself or the JCA container.

The resource adapters represent the [given] facts of our BDD tests, the deployment of the resource adapters represent the [when] phase, while the [then] phase is verified by assertion.

Note that some tests consider an exception a normal output condition using the JUnit 4.x `@Exception(expected = "SomeClass.class")` annotation to identify and verify this situation.

5.2.2. Unit Tests

We are mocking our input/output conditions in our unit tests using the Mockito [http://mockito.googlecode.com] framework to verify class and method behaviors.

An example:

```
@Test
public void printFailuresLogShouldReturnNotEmptyStringForWarning() throws Throwable
{
   //given
   RADeployer deployer = new RADeployer();
   File mockedDirectory = mock(File.class);
   given(mockedDirectory.exists()).willReturn(false);

   Failure failure = mock(Failure.class);
   given(failure.getSeverity()).willReturn(Severity.WARNING);

   List failures = Arrays.asList(failure);
   FailureHelper fh = mock(FailureHelper.class);
   given(fh.asText((ResourceBundle) anyObject())).willReturn("myText");

   deployer.setArchiveValidationFailOnWarn(true);

   //when
   String returnValue = deployer.printFailuresLog(null, mock(Validator.class),
                                                  failures, mockedDirectory, fh);

   //then
   assertThat(returnValue, is("myText"));
}
```

As you can see the BDD style respects the test method name and using the `given-when-then` sequence in order.

5.3. Quality Assurance

In addition to the test suite the JBoss JCA project deploys various tools to increase the stability of the project.

The following sections will describe each of these tools.

5.3.1. Checkstyle

Checkstyle is a tool that verifies that the formatting of the source code in the project is consistent.

This allows for easier readability and a consistent feel of the project.

The goal is to have zero errors in the report. The checkstyle report is generated using

```
ant checkstyle
```

The report is generated into

```
reports/checkstyle
```

The home of checkstyle is located here: http://checkstyle.sourceforge.net/.

5.3.2. Findbugs

Findbugs is a tool that scans your project for bugs and provides reports based on its findings.

This tool helps lower of the number of bugs found in the JBoss JCA project.

The goal is to have zero errors in the report and as few exclusions in the filter as possible. The findbugs report is generated using

```
ant findbugs
```

The report is generated into

```
reports/findbugs
```

The home of findbugs is located here: http://findbugs.sourceforge.net/.

5.3.3. Cobertura

Cobertura generates a test suite matrix for your project which helps you identify where you need additional test coverage.

The reports that the tool provides makes sure that the JBoss JCA project has the correct test coverage.

The goal is to have as high code coverage as possible in all areas. The Cobertura report is generated using

```
ant cobertura
```

The report is generated into

```
reports/cobertura
```

The home of Cobertura is located here: http://cobertura.sourceforge.net/.

5.3.4. Tattletale

Tattletale generates reports about different quality matrix of the dependencies within the project.

The reports that the tool provides makes sure that the JBoss JCA project doesn't for example have cyclic dependencies within the project.

The goal is to have as no issues flagged by the tool. The Tattletale reports are generated using

```
ant tattletale
```

The reports are generated into

```

reports/tattletale
```

The home of Tattletale is located here: http://www.jboss.org/tattletale.

6 Metadata

6.1. Core Metadata

The metadata for the JBoss JCA project is split up into the following areas

- Java Connector Architecture Metadata
- JBoss Metadata
- DataSource Metadata

The implementation of these areas is done in the JBoss Metadata (JBMETA) project, which is a common project for all metadata inside JBoss.

The issue tracking system for JBoss Metadata is located here: https://jira.jboss.org/jira/browse/JBMETA.

6.1.1. Java Connector Architecture Metadata

The Java Connector Architecture (JCA) metadata implement the metadata defined in the JCA specifications. We have metadata representing the following standards:

- Java Connector Architecture 1.0
- Java Connector Architecture 1.5
- Java Connector Architecture 1.6

These metadata versions have a common super class

```
org.jboss.metadata.rar.spec.ConnectorMetaData
```

which allow the developer to for example query the version of the metadata.

The metadata is part of the

```
org.jboss.metadata.rar.jboss.RARDeploymentMetaData
```

structure.

6.1.2. JBoss Metadata

The JBoss metadata implement JBoss specific extensions to the standard Java Connector Architecture metadata. We have metadata representing:

- JBoss RA 1.0

The metadata can be found in:

```
org.jboss.metadata.rar.jboss.JBossRAMetaData
```

The metadata is part of the

```
org.jboss.metadata.rar.jboss.RARDeploymentMetaData
```

structure.

6.1.3. DataSource Metadata

TODO

6.2. Metadata Repository

The metadata repository serves as a central point for all the metadata in the systems.

6.2.1. Interface

The interface of the metadata repository is located in:

```
org.jboss.jca.core.api.MetaDataRepository
```

providing methods to query and update the repository.

6.2.2. Bean

The JBoss Microcontainer bean for the metadata repository can be defined as:

```
<bean name="MetaDataRepository"
      interface="org.jboss.jca.core.api.MetaDataRepository"
      class="org.jboss.jca.core.mdr.MetaDataRepositoryImpl">
</bean>
```

7 Deployers

7.1. RAR Deployer

7.2. DataSource Deployer

8 Standalone

8.1. Overview

The standalone JBoss JCA container implements Chapter 3 Section 5 of the JCA 1.6 specification which defines a standalone JCA environment.

The standalone container has the following layout:

- `$JBOSS_JCA_HOME/bin/`

  contains the run scripts and the SJC kernel.

- `$JBOSS_JCA_HOME/config/`

  contains the configuration of the container.

- `$JBOSS_JCA_HOME/deploy/`

  contains the user deployments.

- `$JBOSS_JCA_HOME/lib/`

  contains all the libraries used by the container.

- `$JBOSS_JCA_HOME/log/`

  contains the log files.

- `$JBOSS_JCA_HOME/tmp/`

  contains temporary files.

To start the container execute the following

```
cd $JBOSS_JCA_HOME/bin
./run.sh
```

.

8.2. JBoss Microcontainer

8.3. JBoss JCA/SJC

Warning

This standalone configuration is for development purposes only.

The JBoss JCA/SJC uses the Fungal kernel for its run-time environment.

The homepage for the Fungal is http://jesperpedersen.github.com/fungal
.

SJC is short for "Simple JCA Container".