

Developer Guide

Preface

About IronJacamar

The goal of the IronJacamar project is to provide an implementation of the Java Connector Architecture 1.7 specification.

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

The IronJacamar project is licensed under the GNU LESSER GENERAL PUBLIC LICENSE 2.1 (LGPL 2.1) license.

Why IronJacamar ?

The Java EE Connector Architecture container can be viewed as a foundation inside an application server as it provides connectivity to the other containers such that they can communicate with EISes. Iron is often used as foundation in building houses too.

The Jacamar bird family which lives in Central and South America are glossy elegant birds with long bills and tails. Why we picked the Jacamar family is left as an exercise for the reader :)

Versions

This section contains the highlights of the IronJacamar releases. A full description of each release can be found through our issue tracking system at <http://issues.jboss.org/browse/BJCA>.

IronJacamar 2.0

- New architecture
- Focus on EE 6+ interaction patterns
- Similar configuration to IronJacamar 1.x deployments
- High amount of reuse of IronJacamar 1.x components

IronJacamar 1.2

Highlights as compared to IronJacamar 1.1:

- Support for graceful shutdown of `ConnectionManager` and `WorkManager`
- Support for connectable `XAResources`
- Support tracking of connection handles across transaction boundaries
- Additional statistics for pools
- Event tracer for easier debugging

IronJacamar 1.1

Highlights as compared to IronJacamar 1.0:

- Java EE Connector Architecture 1.7 certified (standalone / Java EE7)
- Lazy connection manager (JCA chapter 7.16)
- Distributed work manager (JCA chapter 10.3.11)
- Advanced pool capacity policies and flush strategies
- Enhanced Arquillian integration
- Eclipse development plugin
- Enterprise Information System testing server
- Resource adapter information tool
- Migration tools

IronJacamar 1.0

Highlights as compared to previous Java EE Connector Architecture containers inside JBoss Application Server:

- Java EE Connector Architecture 1.6 certified (standalone / Java EE6)
- POJO container environment
- New configuration schemas which focuses on usability
- Fast XML and annotation parsing for quick deployment
- Reauthentication support
- Prefill support for security backed domains
- Support for pool flushing strategies
- Embedded environment for ease of development with Arquillian and ShrinkWrap integration
- New management and statistics integration for components
- Code generator for resource adapters
- Validator tool for resource adapters

The team

Jesper Pedersen acts as the lead for the IronJacamar project. He can be reached at [jesper \(dot\) pedersen \(at\) ironjacamar \(dot\) org](mailto:jesper.pedersen@ironjacamar.org).

Stefano Maestri is a core developer on the IronJacamar project. He can be reached at [stefano \(dot\) maestri \(at\) ironjacamar \(dot\) org](mailto:stefano.maestri@ironjacamar.org).

Lin Gao is a core developer on the IronJacamar project. He can be reached at [lin \(dot\) gao \(at\) ironjacamar \(dot\) org](mailto:lin.gao@ironjacamar.org).

Tom Jenkinson is an advocate for the IronJacamar project. He can be reached at tom (dot) jenkinson (at) ironjacamar (dot) org.

Johnaton Lee helps out in the IronJacamar community with identifying issues, and fixing them. He can be reached at johnathonlee (at) ironjacamar (dot) org.

Tyronne Wickramarathne helps out in the IronJacamar community with identifying issues, and fixing them. He can be reached at tyronne (at) ironjacamar (dot) org.

Thanks to

Adrian Brock, Carlo de Wolf, Gurkan Erdogan, Bruno Georges, Paul Gier, Jason Greene, Stefan Guillhen, Jonathan Halliday, Søren Hilmer, Ales Justin, Vicky Kak, Aslak Knutsen, Sacha Labourey, Mark Little, Alexey Loubyansky, Patrick MacDonald, Scott Marlow, Shelly McGowan, Andrig Miller, Marcus Moyses, John O'Hara, Weston Price, Andrew Lee Rubinger, Heiko Rupp, Anil Saldhana, Scott Stark, Clebert Suconic, Andy Taylor, Vladimir Vasilev, Vladimir Rastseluev, Jeff Zhang, Tyronne Wickramarathne, Dimitris Andreadis, Jeremy Whiting, Yang Yong and Leslie York.

License

Copyright © 2014 Red Hat, Inc. and others.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA").

An explanation of CC-BY-SA is available at <http://creativecommons.org/licenses/by-sa/3.0/>. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Chapter 1. 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), databases 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.

1.1. What's New

1.1.1. Java Connector Architecture 1.7

The Java Connector Architecture 1.7 specification adds the following areas:

- Adds an activation name for message endpoints to uniquely identify them
- Deployment annotations for connection factories and administration objects



The deployment annotations are only meant for developer usage, and should not be used in test or production environments.

The IronJacamar standalone and embedded distributions doesn't support these annotations.

1.1.2. Java Connector Architecture 1.6

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 submitting 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.

1.2. Overview

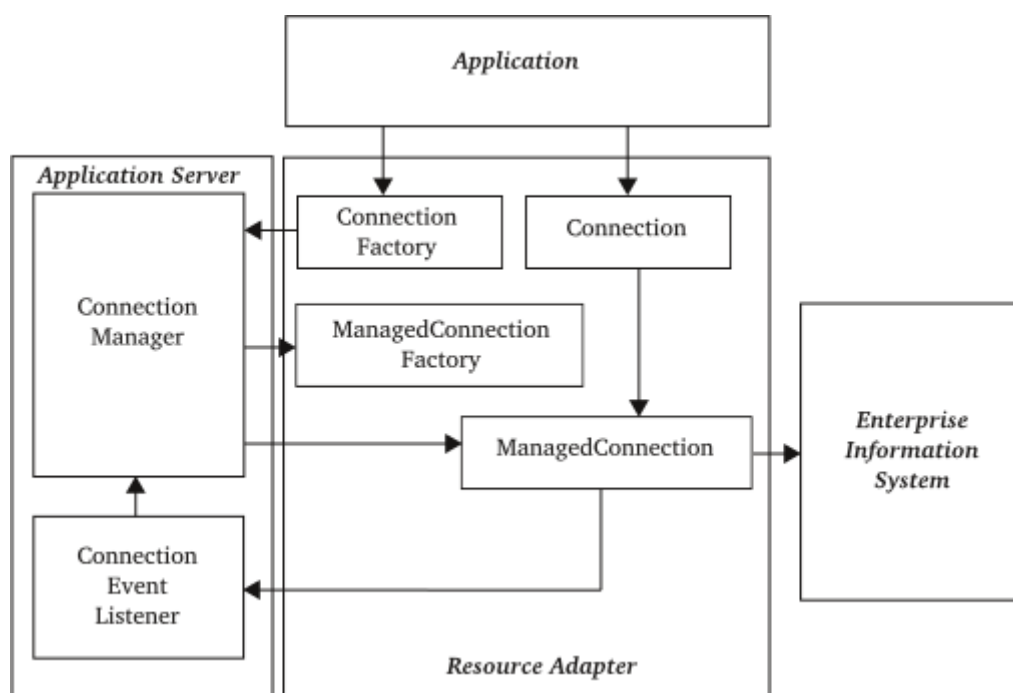
The Java EE Connector Architecture features three different types of resource adapters

- Outbound: The resource adapter allows the application to communicate to the Enterprise Information System (EIS).
- Inbound: The resource adapter allows messages to flow from the Enterprise Information System (EIS) to the application.
- Bi-directional: The resource adapter features both an outbound and an inbound part.

For more information about Java EE Connector Architecture see the specification.

1.2.1. Outbound resource adapter

The Java Connector Architecture specification consists of a number of outbound components:



The application uses the

- ConnectionFactory: The connection factory is looked up in Java Naming and Directory Interface (JNDI) and is used to create a connection.
- Connection: The connection contains the Enterprise Information System (EIS) specific operations.

The resource adapter contains

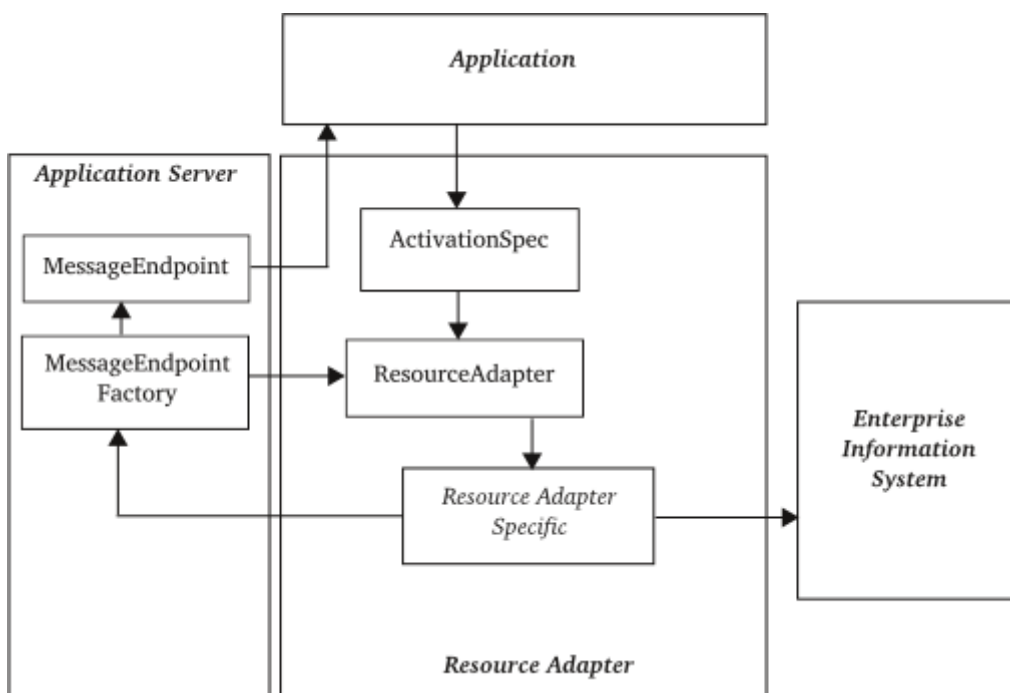
- **ManagedConnectionFactory**: The managed connection factory creates managed connections.
- **ManagedConnection**: The managed connection represents a physical connection to the target Enterprise Information System (EIS). The managed connection notifies the application server of events such as connection closed and connection error.

IronJacamar - the application server - contains

- **ConnectionManager**: The connection manager handles all managed connections in regards to pooling, transaction and security.
- **ConnectionEventListener**: The connection event listener allows the connection manager to know the status of each managed connection.

1.2.2. Inbound resource adapter

The Java Connector Architecture specification consists of a number of inbound components:



The application uses the

- **ActivationSpec**: The activation specification specifies the different properties that the application is looking for from the resource adapter and hence the Enterprise Information System (EIS). This specification can be hidden from the user by a facade provided by the application server.

The resource adapter contains

- **ResourceAdapter**: The resource adapter provides the activation point for inbound communication.
- **Resource adapter specific**: The resource adapter specific code handles communication with the Enterprise Information System (EIS) and deliver messages through the

MessageEndpointFactory.

IronJacamar - the application server - contains

- MessageEndpointFactory: The MessageEndpointFactory is registered with the ResourceAdapter instance and creates the MessageEndpoint instances.
- MessageEndpoint: The MessageEndpoint contains the actual message from the Enterprise Information System (EIS) which the application uses. This could for example be a message driven Enterprise JavaBean (EJB/MDB).

1.3. Project resources

- Web site: <http://www.ironjacamar.org>
- Users forum: <https://groups.google.com/forum/#!forum/ironjacamar-users>
- developers forum: <https://groups.google.com/forum/#!forum/ironjacamar-developers>
- Issue tracker: <https://github.com/ironjacamar/ironjacamar/issues>
- Source code: <https://github.com/ironjacamar/ironjacamar>
- IRC channel: #ironjacamar on freenode.net

1.4. Prerequisites

1.4.1. Java Development Kit (JDK)

You must have the following JDK installed in order to build the project:

- Sun JDK 1.8.x
- OpenJDK 1.8.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
```

1.4.2. Gradle

IronJacamar 2.x build is based on gradle. We are using gradle wrapper to avoid any needs of installation and configuration of the tool.



To build only older version (1.x series) of the project you must have Apache Ant 1.9.4+ installed on your system.

Please refer to specific version documentation for further and more detailed description.

1.4.3. Git

You must have Git installed on your system.

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

1.5. Obtaining the source code

1.5.1. Forking the repository

The IronJacamar repository is located at:

```
https://github.com/ironjacamar/ironjacamar
```

Press the "Fork" button in order to fork the repository to your own GitHub account.

Clone your repository to your machine using

```
git clone git@github.com:<your_account>/ironjacamar.git
```

Next add the upstream repository as a remote location:

```
cd ironjacamar
git remote add upstream git@github.com:ironjacamar/ironjacamar.git
```

1.5.2. Git branches

We have the following branches for the project:

- master

The head of development targeting the next upcoming release. (2.0)

- 1.0

The development targeting the IronJacamar 1.0 releases.

- 1.1

The development targeting the IronJacamar 1.1 releases.

- 1.2

The development targeting the IronJacamar 1.2 releases.

1.6. Compiling the source code

In order to build IronJacamar you should execute from project's main directory:

```
./gradlew <task>
```

Task is option and if you don't specify it gradle would execute default task of the project which is "build"

For a complete list of available tasks and their description just run

```
./gradlew tasks
```

1.7. Creating a patch

NOTE: A prerequisite to submit a patch through github Pull Request is to sign Contribution Licence Agreement here: <https://cla.jboss.org/contributions/index.seam>

Our user guide explains in the "I would like to implement a feature" section how to get started on a writing a new feature or submitting a patch to the project.

You should develop your feature on a Git branch using

```
git checkout -b <feature_name>
```

Once you are done you will need to rebase your work with the latest master.

```
git fetch upstream  
git rebase -i upstream/master
```

You will need to resolve any conflicts of course. Note, that all pull requests must be rebased against upstream master in order to get merged.

Please verify you are able to build the project, all tests pass and the checkstyle is respected. Just run

```
./gradlew --info
```

It should build w/o any error.

Then push the feature to your repository

```
git push origin <feature_name>
```

Go to your account on GitHub.com and submit a pull request via the "Pull request" button

```
https://www.github.com/<your_account>/ironjacamar
```

Remember to select the correct branch, fill in the subject with a short description of the feature, and fill in the description with the full description of the feature.

If your feature / bug-fix applies to multiple branches you will need to submit multiple pull requests - one pull request per branch.

Remember your commit message should contain the issue ID and its description previously created in our issue tracker. Please refer to issue tracking chapter of this document for more information on how to create and manage issues.

1.8. Continuous Integration

When you send a PR Travis CI (<https://travis-ci.org>) will automatically build the project.

During the build, Travis update the status of the commits to one of:

- a warning that the build is still running.
- that the pull request should be merged with caution because the build failed.
- that the pull request can be merged safely because the build was successful.

Travis CI builds a pull request when it is first opened, and when commits are added to the pull request .

1.9. Code review and merging

Finally your PR will be reviewed by a core developer and merged if everything is ok, commented to be refined otherwise.

Don't forget PR and issues are not forums and you will be encouraged discussing design and details through forums, while issue/PRs comments should be used only for specific details or referring to specific line of code.

Chapter 2. Releases

The chapter describes the various releases and their exit criteria.

2.1. Overview

Each release is labelled with a version number and an identifier.

```
ironjacamar-<major>.<minor>.<patch>[.<identifier>]
```

where

- Major: The major version number. Signifies major changes in the implementation.
- Minor: The minor version number. Signifies functional changes to a major version.
- Patch: The patch version number. Signifies a binary compatible change to a minor version.
- - None / Final: Stable release
 - CR: Candidate for Release quality. The implementation is functional complete.
 - Beta: Beta quality. The implementation is almost functional complete.
 - Alpha: Alpha quality. The implementation is a snapshot of the development.

2.2. Versioning

Each release will contain a version number which relates to the feature branch where it was created.

2.2.1. Major

A Major version identifier signifies major changes in the implementation such as a change in the architecture.

The features between major versions can be a lot different, and therefore feature regressions may appear.

A Major version will most likely also mean updates to the configuration and required metadata files for deployments.

2.2.2. Minor

A Minor version identifier signifies functional changes to a Major release.

This means that new features have been added to the Major release, and hence may have new configuration options and integration points.

The release is binary compatible to the previous releases - for example 1.0 vs. 1.1.

2.2.3. Patch

A Patch version identifier signifies a binary compatible update to one or more components in a Minor release.

This means that one or more bug fixes to existing components have been integrated in the branch in question.

The release is binary compatible to the previous releases - for example **1.0.0** vs. **1.0.1**.

2.3. Identifiers

Each release will contain an identifier which relates to the release quality.

2.3.1. Alpha releases

An Alpha release is a snapshot of the main development branch which likely will contain new features.



Alpha releases are NOT production quality

An Alpha release is made each month (time-boxed) unless the branch is using an identifier as Beta or higher.

The exit criteria for an Alpha release is that the main test suite is passing.

2.3.2. Beta releases

A Beta release contains major features that are considered almost functional complete. This doesn't mean however that all aspects of each feature is complete and therefore not all options will be active.



Beta releases are NOT production quality

A Beta release will be made once one or more features are almost functional complete and therefore Beta releases aren't time-boxed, but feature-boxed instead.

The exit criteria for a Beta release is that all test suites are passing.

2.3.3. Candidate for Release releases

A Candidate for Release is considered functional complete and candidate for being promoted to a Final release.



Candidate for Release releases are NOT production quality

A Candidate for Release focuses on functionality, but they are time-boxed to a maximum of two weeks between each release.

The exit criteria for a Candidate for Release release is that all test suites are passing.

2.3.4. Final releases

A Final release is considered feature complete and stable.

Typically only one Final release will be released from each branch, unless critical or blocker issues are found in the release. Patch releases will be available from our source control system as tags.

The exit criteria for a Final release is that all test suites are passing.

Chapter 3. Issue tracking

3.1. Location

IronJacamar 2.x series is using Github issue tracker located at <https://github.com/ironjacamar/ironjacamar/issues>

There is also a JIRA issue tracking for bugs impacting older releases. It is located at <http://issues.jboss.org/browse/JBJCA>



For further informations on how Jira issue tracking is organized and how to use it please refer to 1.x documentation

3.2. Labels

Github issue tracking uses a label based system to tag issues. IronJacamar is using different labels to identify:

- Black labels: Components impacted by the issue
- Yellow labels: Issue category
- Blue labels: Issue priority
- Violet labels: rejecting reasons

3.3. Components (black labels)

The project is divided into the following components:

Table 1. Project components

Label name	Description
API	Public API
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.
Documentation	The documentation (Users Guide / Developers Guide) for the project.
Deployer	The deployers for the project.
Embedded	The embedded IronJacamar container.
JDBC	A JDBC resource adapter.
Performance	Performance related work.

Label name	Description
Standalone	The standalone IronJacamar distribution.
Test Suite	The IronJacamar test suite.
Tools	Tools used related work.
Validator	The resource adapter validator.

3.4. Categories (yellow labels)

The system contains the following categories:

Table 2. Issues categories

Label name	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.
Component Update	Identifies a thirdparty library dependency.

3.5. Priorities (blue labels)

All issues are assigned one of the following priorities:

Table 3. Issues Priorities

Label name	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.

3.6. Rejecting reasons (violet labels)

When an issue is rejected it's closed and tagged with one of these labels. A comment should explain why it has been rejected.

Table 4. Rejecting reasons

Label name	Description
wontfix	The issue will not be fixed because it's not a bug and described behaviour is expected,
duplicated	The issue is duplicated. A link to original issue is provided in closing comment.
cannot reproduce	The issue is not reproducible with provided information and description.

3.7. Life cycle

All issues have the following life cycle:

Table 5. JIRA Lifecycle

Lifecycle	Description
Open	An issue currently not implemented.
Closed	An issue that has been resolved. It will be included in target release.

Chapter 4. Testing

4.1. Overall goals

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

The full test suite runs every time you build the project.

```
./gradlew --info
```

You can run just testsuite w/o remake a full build

```
---  
./gradlew test --info  
---
```

There is few special tests running multiple instance of embedded environment excluded by default build. You can run also them with

```
---  
./ghradlew test --info -PmiTest=true  
---
```

A single test case can be executed using

```
./gradlew --info test --tests <testname>
```

where <testname> is a fully qualified name of test class Note you can also use * operator to filter test name

```
./gradlew --info test --tests *JGroups*
```

You can also run test in debug mode and attach your favorite remote debugger on port 5005 running

```
./gradlew --info -Dtest.debug
```

Of course you can combine above options for example to debug a single test case.

4.2. Testing principle and style

Our tests follow 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](#)
- 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](#) 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.

4.2.1. Integration Tests

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

The resource adapters are created using either the main build environment or by using [ShrinkWrap](#). Using resource adapters within the test cases will allow you to debug both the resource adapters themselves 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.

This kind of tests run inside our embedded environment and creating deployment descriptor with our builder. Few custom annotations make embedded environment start/deploy/undeploy/stop in test classes very simple and human readable. A test class looks like this:

```

package org.ironjacamar.core.workmanager;

import org.ironjacamar.embedded.Configuration;
import org.ironjacamar.embedded.Deployment;
import org.ironjacamar.embedded.dsl.resourceadapters20.api.ResourceAdaptersDescriptor;
import org.ironjacamar.embedded.junit4.AllChecks;
import org.ironjacamar.embedded.junit4.IronJacamar;
import org.ironjacamar.embedded.junit4.PostCondition;
import org.ironjacamar.embedded.junit4.PreCondition;
import org.ironjacamar.rars.ResourceAdapterFactory;
import org.ironjacamar.rars.wm.WorkConnection;
import org.ironjacamar.rars.wm.WorkConnectionFactory;

import javax.annotation.Resource;

import org.jboss.shrinkwrap.api.spec.ResourceAdapterArchive;

import org.junit.Test;
import org.junit.runner.RunWith;

import static org.junit.Assert.assertNotNull;

/**
 * Basic WorkManager test case
 * @author <a href="mailto:jesper.pedersen@ironjacamar.org">Jesper Pedersen</a>
 */
@RunWith(IronJacamar.class)
@Configuration(full = true)
@PreCondition(condition = AllChecks.class)
@PostCondition(condition = AllChecks.class)
public class WorkManagerTestCase
{
    /** The user transaction */
    @Resource(mappedName = "java:/eis/WorkConnectionFactory")
    private WorkConnectionFactory wcf;

    /**
     * The resource adapter
     * @throws Throwable In case of an error
     */
    @Deployment(order = 1)
    private ResourceAdapterArchive createResourceAdapter() throws Throwable
    {
        return ResourceAdapterFactory.createWorkRar();
    }

    /**
     * The activation
     * @throws Throwable In case of an error
     */
    @Deployment(order = 2)

```

```

private ResourceAdaptersDescriptor createActivation() throws Throwable
{
    return ResourceAdapterFactory.createWorkDeployment(null);
}

/**
 * Deployment
 * @throws Throwable In case of an error
 */
@Test
public void testDeployment() throws Throwable
{
    assertNotNull(wcf);

    WorkConnection wc = wcf.getConnection();
    assertNotNull(wc);

    wc.close();
}
}

```

While ResourceAdapterFactory methods invoked like this

```

/**
 * Create the work.rar
 *
 * @return The resource adapter archive
 */
public static ResourceAdapterArchive createWorkRar()
{
    org.jboss.shrinkwrap.descriptor.api.connector16.ConnectorDescriptor raXml =
    Descriptors

.create(org.jboss.shrinkwrap.descriptor.api.connector16.ConnectorDescriptor.class,
"ra.xml").version("1.6");

    org.jboss.shrinkwrap.descriptor.api.connector16.ResourceadapterType rt =
raXml.getOrCreateResourceadapter()
        .resourceadapterClass(WorkResourceAdapter.class.getName());
    org.jboss.shrinkwrap.descriptor.api.connector16.OutboundResourceadapterType ort
= rt

.getOrCreateOutboundResourceadapter().transactionSupport("NoTransaction").reauthenticat
tionSupport(false);
    org.jboss.shrinkwrap.descriptor.api.connector16.ConnectionDefinitionType cdt =
ort.createConnectionDefinition()

.managedconnectionfactoryClass(WorkManagedConnectionFactory.class.getName())
        .connectionfactoryInterface(WorkConnectionFactory.class.getName())
        .connectionfactoryImplClass(WorkConnectionFactoryImpl.class.getName())

```

```

        .connectionInterface(WorkConnection.class.getName())
        .connectionImplClass(WorkConnectionImpl.class.getName());

    ResourceAdapterArchive raa = ShrinkWrap.create(ResourceAdapterArchive.class,
"work.rar");

    JavaArchive ja = ShrinkWrap.create(JavaArchive.class, "work.jar");
    ja.addPackages(true, WorkConnection.class.getPackage());

    raa.addAsLibrary(ja);
    raa.addAsManifestResource(new StringAsset(raXml.exportAsString()), "ra.xml");

    return raa;
}

/**
 * Create the work.rar deployment
 *
 * @param bc The BootstrapContext name; <code>null</code> if default
 * @return The resource adapter descriptor
 */
public static ResourceAdaptersDescriptor createWorkDeployment(String bc)
{
    ResourceAdaptersDescriptor dashRaXml =
Descriptors.create(ResourceAdaptersDescriptor.class, "work-ra.xml");

    ResourceAdapterType dashRaXmlRt =
dashRaXml.createResourceAdapter().archive("work.rar");
    if (bc != null)
        dashRaXmlRt.bootstrapContext(bc);
    ConnectionDefinitionsType dashRaXmlCdst =
dashRaXmlRt.getOrCreateConnectionDefinitions();
    org.ironjacamar.embedded.dsl.resourceadapters20.api.ConnectionDefinitionType
dashRaXmlCdt = dashRaXmlCdst

.createConnectionDefinition().className(WorkManagedConnectionFactory.class.getName())
        .jndiName("java:/eis/WorkConnectionFactory").id("WorkConnectionFactory");

    org.ironjacamar.embedded.dsl.resourceadapters20.api.PoolType dashRaXmlPt =
dashRaXmlCdt.getOrCreatePool()
        .minPoolSize(0).initialPoolSize(0).maxPoolSize(10);

    return dashRaXml;
}

```

4.2.2. Unit Tests

We are mocking our input/output conditions in our unit tests using the [Mockito](#) 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.

4.3. Quality Assurance

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

The following sections will describe each of these tools.

4.3.1. 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 on every build and can be found under each module's build directory.

4.3.2. JaCoCo

JaCoCo 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 IronJacamar project has the correct test coverage.

The goal is to have as high code coverage as possible in all areas. The JaCoco report is generated at every build

The report is generated into

```
testsuite/build/reports/jacoco
```

The home of JaCoCo is located here: <http://www.eclemma.org/jacoco/>.

4.4. Performance testing

Performance testing can identify areas that need to be improved or completely replaced.

4.4.1. JProfiler

Insert the following line in `run.sh` or `run.bat`:

```
-agentpath:<path>/jprofiler6/bin/linux-x64/libjprofilerti.so=port=8849
```

where the Java command is executed.

The home of JProfiler is located here: <http://www.ej-technologies.com/products/jprofiler/overview.html>.

4.4.2. OProfile

OProfile can give a detailed overview of applications running on the machine, including Java program running with OpenJDK.

The home of OProfile is located here: <http://oprofile.sourceforge.net>.

Installation

Enable the Fedora debug repo:


```
/etc/yum.repos.d/fedora.repo
```

```
[fedora-debuginfo]
name=Fedora $releasever - $basearch - Debug
failovermethod=priority
mirrorlist=https://mirrors.fedoraproject.org/metalink?repo=fedora-debug-
$releasever&arch=$basearch
enabled=1
gpgcheck=1
gpgkey=file:///etc/pki/rpm-gpg/RPM-GPG-KEY-fedora-$basearch
```

Install:

```
dnf install -y oprofile oprofile-jit
dnf install -y yum-plugin-auto-update-debug-info
dnf install -y java-1.6.0-openjdk-debuginfo
```

If you are using Fedora 21 or older, you need to use yum instead of dnf to install OProfile:

```
yum install -y oprofile oprofile-jit
yum install -y yum-plugin-auto-update-debug-info
yum install -y java-1.6.0-openjdk-debuginfo
```

Running

Insert the following line in `run.sh` or `run.bat`:

```
-agentpath:/usr/lib64/oprofile/libjvmti_oprofile.so
```

for 64bit JVMs or

```
-agentpath:/usr/lib/oprofile/libjvmti_oprofile.so
```

for 32 bit JVMs where the Java command is executed.

Now execute:

```
opcontrol --no-vmlinux
opcontrol --start-daemon
```

and use the following commands:

```
opcontrol --start # Starts profiling
opcontrol --dump # Dumps the profiling data out to the default file
opcontrol --stop # Stops profiling
```

Once you are done execute:

```
opcontrol --shutdown # Shuts the daemon down
```

A report can be generated by:

```
opreport -l --output-file=<filename>
```

Remember that this is system wide profiling, so make sure that only the services that you want included are running.

More information is available at <http://oprofile.sourceforge.net/doc/index.html>.

4.4.3. Performance test suite

TODO

Chapter 5. Standalone

5.1. Overview

The standalone IronJacamar 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:

- `$IRON_JACAMAR_HOME/bin/`
contains the run scripts and the SJC kernel.
- `$IRON_JACAMAR_HOME/config/`
contains the configuration of the container.
- `$IRON_JACAMAR_HOME/deploym/`
contains the user deployments.
- `$IRON_JACAMAR_HOME/doc/`
contains the documentation.
- `$IRON_JACAMAR_HOME/lib/`
contains all the libraries used by the container.
- `$IRON_JACAMAR_HOME/log/`
contains the log files.
- `$IRON_JACAMAR_HOME/system/`
contains system deployments files.
- `$IRON_JACAMAR_HOME/tmp/`
contains temporary files.

To start the container execute the following

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

.

5.2. IronJacamar/SJC



This standalone configuration is for development purposes only.

The IronJacamar/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".