

Arquillian: An integration testing framework for Containers

Reference Guide

1.0.0-SNAPSHOT

by Dan Allen, Aslak Knutsen, Pete Muir, Andrew Rubinger, and Karel Piwko

Preface: Test in the container!	vii
1. Introduction	1
1.1. Mission statement	1
1.2. Architecture overview	2
1.3. Integration testing in Java EE	3
1.3.1. Testing the real component	4
1.3.2. Finding a happy medium	4
1.3.3. Controlling the test classpath	4
1.4. Usage scenarios	5
2. Introductory examples	7
2.1. Testing an EJB	10
2.2. Testing CDI beans	11
2.3. Testing JPA	12
2.4. Testing JMS	14
3. Getting started	17
3.1. Setting up Arquillian in a Maven project	17
3.2. Writing your first Arquillian test	18
3.3. Setting up and running the test in Maven	21
3.4. Setting up and running the test in Eclipse	23
3.5. Setting up and running the test in NetBeans	25
4. Target containers	27
4.1. Container varieties	27
4.2. Container management	28
4.3. Supported containers	28
4.4. Container configuration	29
5. Test enrichment	31
5.1. Injection into the test case	31
5.2. Active scopes	32
6. Test execution	33
6.1. Anatomy of a test	33
6.2. ShrinkWrap packaging	33
6.3. Test archive deployment	34
6.4. Enriching the test class	34
6.5. Negotiating test execution	34
6.6. Test run modes	35
6.6.1. Mode: in-container	35
6.6.2. Mode: as-client	36
6.6.3. Mode: mixed	37
7. Debugging remote tests	39
7.1. Debugging in Eclipse	39
7.1.1. Attaching the IDE debugger to the container	39
7.1.2. Launching the test in debug mode	40
7.1.3. Stepping into external libraries	40
7.2. Assertions in remote tests	41

7.2.1. Enabling assertions in JBoss AS	41
8. Build system integration	43
8.1. Arquillian's active build ingredient	43
8.2. Integrating Arquillian into a Gradle build	44
8.2.1. apply from: common	44
8.2.2. Strategy #1: Container-specific test tasks	47
8.2.3. Strategy #2: Test profiles	51
8.3. Integrating Arquillian into an Ant (+Ivy) build	55
9. Advanced use cases	57
9.1. Descriptor deployment	57
9.2. Resource injection	57
9.3. Multiple Deployments	58
9.4. Multiple Containers	58
9.5. Protocol selection	60
10. Extending Arquillian	63
11. Complete Extension/Framework Reference	65
11.1. Performance	65
11.2. JSFUnit	66
11.3. Drone	68
12. Complete Container Reference	79
12.1. JBoss AS 5 - Remote	79
12.1.1. Configuration	79
12.2. JBoss AS 5.1 - Remote	80
12.2.1. Configuration	80
12.3. JBoss AS 5.1 - Managed	81
12.3.1. Configuration	81
12.4. JBoss AS 6.0 - Remote	83
12.4.1. Configuration	83
12.5. JBoss AS 6.0 - Managed	84
12.5.1. Configuration	84
12.6. JBoss AS 6.0 - Embedded	85
12.6.1. Configuration	86
12.7. JBoss Reloaded 1.0 - Embedded	87
12.7.1. Configuration	88
12.8. GlassFish 3.1 - Embedded	88
12.8.1. Configuration	88
12.9. GlassFish 3.1 - Remote	89
12.9.1. Configuration	90
12.10. Tomcat 6.0 - Embedded	91
12.10.1. Configuration	92
12.11. Jetty 6.1 - Embedded	94
12.11.1. Configuration	94
12.12. Jetty 7.0 - Embedded	96
12.12.1. Configuration	96

12.13. Weld SE 1.0 - Embedded	98
12.13.1. Configuration	98
12.14. Weld SE 1.1 - Embedded	99
12.14.1. Configuration	99
12.15. Weld EE 1.1 - Embedded	100
12.15.1. Configuration	101
12.16. Apache OpenWebBeans 1.0 - Embedded	103
12.16.1. Configuration	103
12.17. Apache OpenEJB 3.1 - Embedded	105
12.17.1. Configuration	105
13. Complete Protocol Reference	107
13.1. Local	107
13.1.1. Configuration	107
13.2. Servlet 2.5	107
13.2.1. Configuration	108
13.3. Servlet 3.0	108
13.3.1. Configuration	108

Preface: Test in the container!

Ever since the inception of Java EE, testing enterprise applications has been a major pain point. Testing business components, in particular, can be very challenging. Often, a vanilla unit test isn't sufficient for validating such a component's behavior. *Why is that?* The reason is that components in an enterprise application rarely perform operations which are strictly self-contained. Instead, they interact with or provide services for the greater system. They also have declarative functionality which gets applied at runtime. You could say "no business component is an island."

The way the component interacts with the system is just as important as the work it performs. Even with the application separated into more layers than your favorite Mexican dip, to validate the correctness of a component, you have to observe it carrying out its work—*in situ*. Unit tests and mock testing can only take you so far. Business logic aside, how do you test your component's "enterprise" semantics?

Especially true of business components, you eventually have to ensure that the declarative services, such as dependency injection and transaction control, actually get applied and work as expected. It means interacting with databases or remote systems and ensuring that the component plays well with its collaborators. What happens when your Message Driven Bean can't parse the XML message? Will the right component be injected? You may just need to write a test to explore how the declarative services behave, or that your application is configured correctly to use them. This style of testing needed here is referred to as integration testing, and it's an essential part of the enterprise development process.

Arquillian, a new testing framework developed at JBoss.org, empowers the developer to write integration tests for business objects that are executed inside a container or that interact with the container as a client. The container may be an embedded or remote Servlet container, Java EE application server, Java SE CDI environment or any other container implementation provided. Arquillian strives to make integration testing no more complicated than basic unit testing.

The importance of Arquillian in the Java EE space cannot be emphasized enough. If writing good tests for Java EE projects is some dark art in which knowledge is shared only by the Java gurus, people are either going to be turned off of Java EE or a lot of fragile applications are going to be written. Arquillian is set to become the first comprehensive solution for testing Java EE applications, namely because it leverages the container rather than a contrived runtime environment.

This guide documents Arquillian's architecture, how to get started using it and how to extend it. If you have questions, please use the discussion forum in the top-level [Arquillian space](#) on JBoss.org. We also provide a [JIRA issue tracking system](#) for bug reports and feature requests. If you are interested in the development of Arquillian, or want to translate this documentation into your language, we welcome you to join us in the [Arquillian Development subspace](#) on JBoss.org.

Introduction

We believe that integration testing should be no more complex than writing a basic unit test. We created Arquillian to realize that goal. One of the major complaints we've heard about Seam 2 testing (i.e., SeamTest) was, not that it isn't possible, but that it isn't flexible and it's difficult to setup. We wanted to correct those shortcomings with Arquillian.

Testing needs vary greatly, which is why it's so vital that, with Arquillian (and ShrinkWrap), we have decomposed the problem into its essential elements. The result is a completely flexible and portable integration testing framework.

1.1. Mission statement

Arquillian is the missing link in Java EE development. Developers have long had to fend for themselves in the testing stage, burdened with bootstrapping the infrastructure on which the test depends. That's time lost, and it places a high barrier to entry on integration testing. Arquillian tears down that barrier.

Arquillian is a container-oriented test framework. It picks up where unit tests leave off, targeting the integration of application code inside a real runtime environment. Just as Java EE 5 simplified the server programming model by providing declarative services for POJOs, Arquillian equips tests with container lifecycle management and enrichment.

With Arquillian, you write a basic test case and annotate it with declarative behavior that says, "run with Arquillian." Launching the test is as simple as right-clicking the test class in the IDE and selecting Run As > JUnit or TestNG test. Based on the classpath configuration, Arquillian starts or binds to the target container (JBoss AS, GlassFish, OpenEJB, etc) and deploys the test case bundled with the test archive defined in the `@Deployment` method. Your test executes inside the container and enjoys all the same services as an application component. That means you get dependency and resource injection into the test, you can access EJBs, you can load a persistence unit, you can get a handle to a database connection, etc. Yet, on the surface, it looks like any other unit test. (Arquillian also has a client execution mode, which only deploys the test archive, not the test case).

Instead of bringing your runtime to the test, Arquillian brings your test to the runtime.

Features of Arquillian include:

- Runnable from both JUnit and TestNG
- Abstracts out server lifecycle and deployment
- Injects resources like managed beans, EJBs or objects from JNDI into the test instance
- Zero reliance upon a formal build; can be run or debugged from IDEs like Eclipse, IDEA, NetBeans

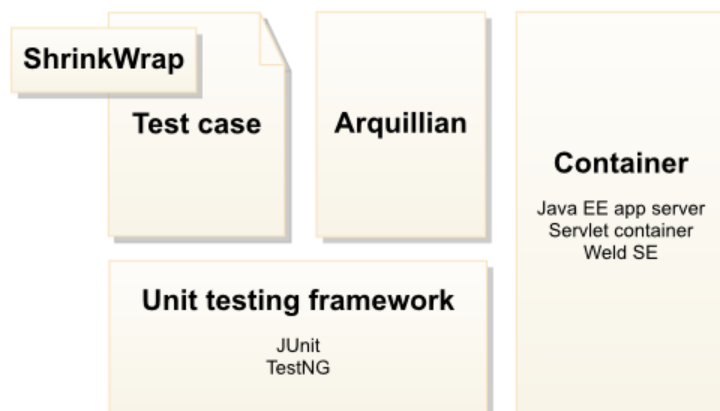
- Supports remote and embedded containers: JBoss AS, GlassFish, Jetty, Tomcat, OpenEJB, OSGi and more on the way
- Enables pass-by-reference between the test and the server, even if the server is in another JVM from the test launcher
- Provides an extensible SPI - plug in your own containers and take advantage of the Arquillian bus to provide services to the test

No longer does writing a test involve system administration tasks. No more custom scripts or copy-paste Maven configuration. No more full builds. No more test classpath mayhem. No more looking up resources manually in JNDI. No more reliance on coarse-grained, black-box testing.

Arquillian keeps you focused on the test, while enjoying the services provided by the container. And it's turning heads.

1.2. Architecture overview

Arquillian combines a unit testing framework (JUnit or TestNG), ShrinkWrap, and one or more supported target containers (Java EE container, servlet container, Java SE CDI environment, etc) to provide a simple, flexible and pluggable integration testing environment.



The Arquillian test infrastructure

At the core, Arquillian provides a *custom test runner for JUnit and TestNG* that turns control of the test execution lifecycle from the unit testing framework to Arquillian. From there, Arquillian can delegate to service providers to setup the environment to execute the tests inside or against the container. An Arquillian test case looks just like a regular JUnit or TestNG test case with two declarative enhancements, which will be covered later.

Since Arquillian works by replacing the test runner, Arquillian tests can be executed using existing test IDE, Ant and Maven test plugins without any special configuration. Test results are reported just like you would expect. That's what we mean when we say using Arquillian is no more complicated than basic unit testing.

At this point, it's appropriate to pause and define the three aspects of an Arquillian test case. This terminology will help you better understand the explanations of how Arquillian works.

1. container — a runtime environment for a deployment
2. deployment — the process of dispatching an artifact to a container to make it operational
3. archive — a packaged assembly of code, configuration and resources

The test case is dispatched to the container's environment through coordination with *ShrinkWrap*, which is used to declaratively define a custom Java EE archive that encapsulates the test class and its dependent resources. Arquillian packages the ShrinkWrap-defined archive at runtime and deploys it to the *target container*. It then negotiates the execution of the test methods and captures the test results using remote communication with the server. Finally, Arquillian undeploys the test archive. We'll go into more detail about how Arquillian works in a later chapter.

So what is the target container? Some proprietary testing container that emulates the behavior of the technology (Java EE)? Nope, it's pluggable. It can be your actual target runtime, such as JBoss AS, GlassFish or Tomcat. It can even be an embedded container such as JBoss Embedded AS, GlassFish Embedded or Weld SE. All of this is made possible by a RPC-style (or local, if applicable) communication between the test runner and the environment, negotiating which tests are run, the execution, and communicating back the results. This means two things for the developer:

- You develop Arquillian tests just like you would a regular unit test and
- the container in which you run the tests can be easily swapped, or you can use each one.

With that in mind, let's consider where we are today with integration testing in Java EE and why an easy solution is needed.

1.3. Integration testing in Java EE

Integration testing is very important in Java EE. The reason is two-fold:

- Business components often interact with resources or sub-system provided by the container
- Many declarative services get applied to the business component at runtime

The first reason is inherent in enterprise applications. For the application to perform any sort of meaningful work, it has to pull the strings on other components, resources (e.g., a database) or systems (e.g., a web service). Having to write any sort of test that requires an enterprise resource (database connection, entity manager, transaction, injection, etc) is a non-starter because the developer has no idea what to even use. Clearly there is a need for a simple solution, and Arquillian fills that void.

Some might argue that, as of Java EE 5, the business logic performed by most Java EE components can now be tested outside of the container because they are POJOs. But let's not forget that in order to isolate the business logic in Java EE components from infrastructure services (transactions, security, etc), many of those services were pushed into declarative programming

constructs. At some point you want to make sure that the infrastructure services are applied correctly and that the business logic functions properly within that context, justifying the second reason that integration testing is important in Java EE.

1.3.1. Testing the real component

The reality is that you aren't really testing your component until you test it in situ. It's all too easy to create a test that puts on a good show but doesn't provide any real guarantee that the code under test functions properly in a production environment. The show typically involves mock components and/or bootstrapped environments that cater to the test. Such "unit tests" can't verify that the declarative services kick in as they should. While unit tests certainly have value in quickly testing algorithms and business calculations within methods, there still need to be tests that exercise the component as a complete service.

Rather than instantiating component classes in the test using Java's new operator, which is customary in a unit test, Arquillian allows you to inject the container-managed instance of the component directly into your test class (or you can look it up in JNDI) so that you are testing the actual component, just as it runs inside the application.

1.3.2. Finding a happy medium

Do you really need to run the test in a real container when a Java SE CDI environment would do?

It's true, some tests can work without a full container. For instance, you can run certain tests in a Java SE CDI environment with Arquillian. Let's call these "standalone" tests, whereas tests which do require a full container are called "integration" tests. Every standalone test can also be run as an integration test, but not the other way around. While the standalone tests don't need a full container, it's also important to run them as integration tests as a final check just to make sure that there is nothing they conflict with (or have side effects) when run in a real container.

It might be a good strategy to make as many tests work in standalone mode as possible to ensure a quick test run, but ultimately you should consider running all of your tests in the target container. As a result, you'll likely enjoy a more robust code base.

We've established that integration testing is important, but how can integration testing be accomplished without involving every class in the application? That's the benefit that ShrinkWrap brings to Arquillian.

1.3.3. Controlling the test classpath

One huge advantage ShrinkWrap brings to Arquillian is classpath control. The classpath of a test run has traditionally been a kitchen sink of all production classes and resources with the test classes and resources layered on top. This can make the test run indeterministic, or it can just be hard to isolate test resources from the main resources.

Arquillian uses ShrinkWrap to create "micro deployments" for each test, giving you fine-grained control over what you are testing and what resources are available at the time the test is executed.

An archive can include classes, resources and libraries. This not only frees you from the classpath hell that typically haunts test runners (Eclipse, Maven), it also gives you the option to focus on the interaction between an subset of production classes, or to easily swap in alternative classes. Within that grouping you get the self-assembly of services provided by Java EE—the very integration which is being tested.

Let's move on and consider some typical usage scenarios for Arquillian.

1.4. Usage scenarios

With the strategy defined above, where the test case is executed in the container, you should get the sense of the freedom you have to test a broad range of situations that may have seemed unattainable when you only had the primitive unit testing environment. In fact, anything you can do in an application you can now do in your test class.

A fairly common scenario is testing an EJB session bean. As you are inside the container, you can simply do a JNDI lookup to get the EJB reference and your test becomes a client of the EJB. But having to use JNDI to get a reference to the EJB is inconvenient (at least to Java EE 5 developers that have become accustomed to annotation-based dependency injection). Arquillian allows you to use the `@EJB` annotation to inject the reference to an EJB session bean into your test class.

EJB session beans are one type of Java EE resource you may want to access. But that's just the beginning. You can access any resource available in a Java EE container, from a `UserTransaction` to a `DataSource` to a mail session. Any of these resources can be injected directly into your test class using the Java EE 5 `@Resource` annotation.

Resource injections are convenient, but they are so Java EE 5. In Java EE 6, when you think dependency injection, you think JSR-299: CDI. Your test class can access any bean in the ShrinkWrap-defined archive, provided the archive contains a `beans.xml` file to make it a bean archive. And you can inject bean instances directly into your class using the `@Inject` annotation, or you can inject an `Instance` reference to the bean, allowing you to create a bean instance when needed in the test. Of course, you can do anything else you can do with CDI within your test as well.

Another important scenario in integration testing is performing data access. If the ShrinkWrap-defined archive contains a `persistence.xml` descriptor, the persistence unit will be started when the archive is deployed and you can perform persistence operations. You can obtain a reference to an `EntityManager` by injecting it into your class with `@PersistenceContext` or from a CDI producer-field. Alternatively, you can execute the persistence operation indirectly through an EJB session bean or a managed bean.

Those examples should give you an idea of some of the tasks that are possible from within an Arquillian-enhanced test case. Now that you have plenty of motivation for using Arquillian, let's look at how to get started using Arquillian.

Introductory examples

The following examples demonstrate the use of Arquillian. Currently Arquillian is distributed as a Maven only project, so you'll need to grab the examples from Git. You can choose between a [JUnit example](http://github.com/arquillian/arquillian/tree/1.0.0-SNAPSHOT/examples/junit) [http://github.com/arquillian/arquillian/tree/1.0.0-SNAPSHOT/examples/junit] and a [TestNG example](http://github.com/arquillian/arquillian/tree/1.0.0-SNAPSHOT/examples/testng) [http://github.com/arquillian/arquillian/tree/1.0.0-SNAPSHOT/examples/testng]. In this tutorial we show you how to use both.

```
git clone git://github.com/arquillian/arquillian.git arquillian
cd arquillian
git checkout 1.0.0-SNAPSHOT
```

```
cd examples/junit
```

```
cd examples/testng
```

Running these tests from the command line is easy. The examples run against all the servers supported by Arquillian (of course, you must choose a container that is capable of deploying EJBs for these tests). To run the test, we'll use Maven. For this tutorial, we'll use JBoss AS 6 (currently at Milestone 3), for which we use the `jbossas-remote-6` profile.

First, make sure you have a copy of JBoss AS; you can download it from [jboss.org](http://www.jboss.org/jbossas/downloads) [http://www.jboss.org/jbossas/downloads]. We strongly recommend you use a clean copy of JBoss AS. Unzip JBoss AS to a directory of your choice and start it; we'll use `$JBOSS_HOME` to refer to this location throughout the tutorial.

```
$ unzip jboss-6.0.0.Final.zip && mv jboss-6.0.0.Final $JBOSS_HOME && $JBOSS_HOME/bin/run.sh
```

Now, we tell Maven to run the tests, for both JUnit and TestNG:

```
$ mvn test -Pjbossas-remote-6

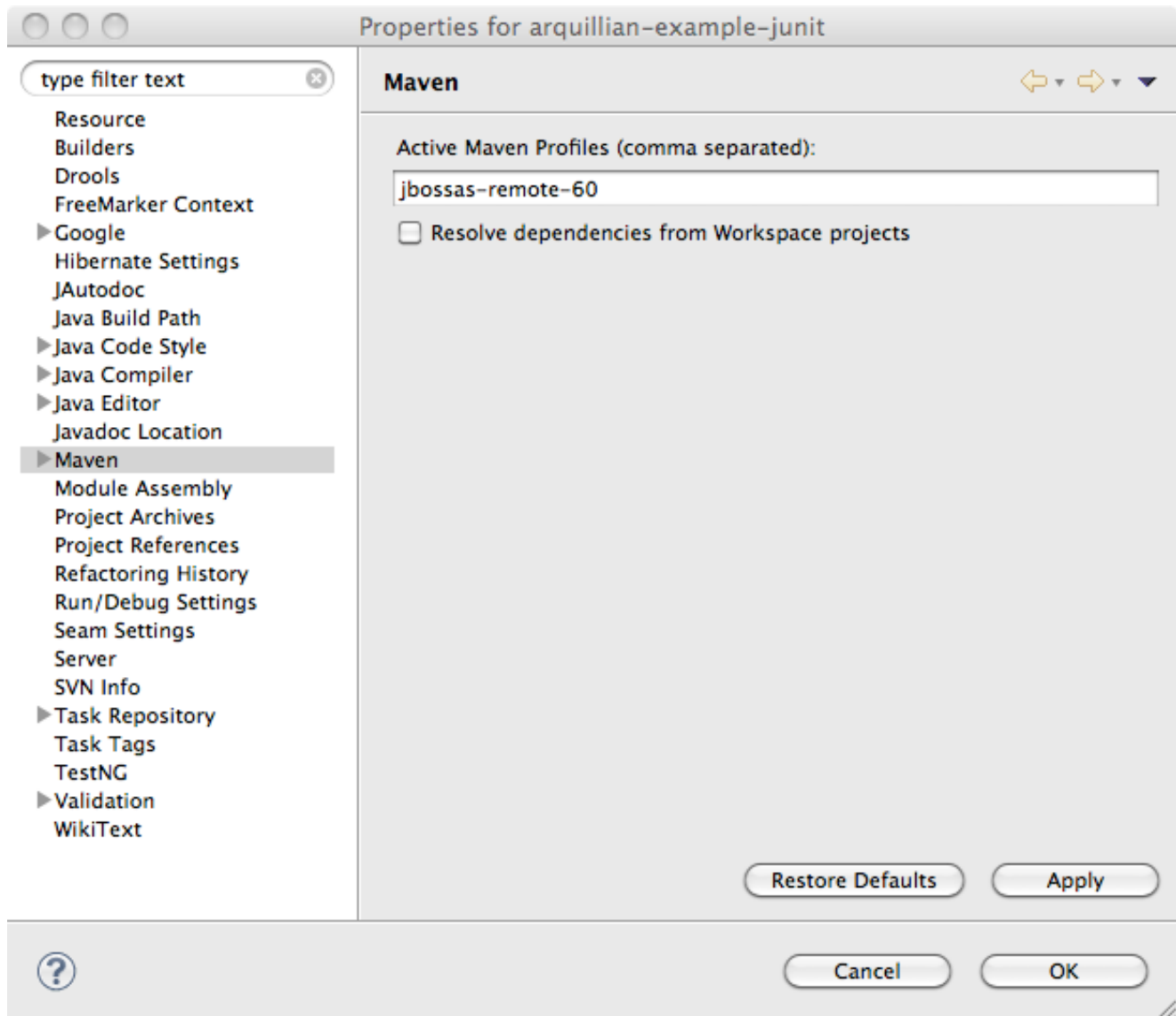
$ cd ../arquillian-example-junit/
$ mvn test -Pjbossas-remote-6
```

You can also run the tests in an IDE. We'll show you how to run the tests in Eclipse, with m2eclipse installed, next.

Before running an Arquillian test in Eclipse, you must have the plugin for the unit testing framework you are using installed. Eclipse ships with the JUnit plugin, so you are already setup if you selected JUnit. If you are writing your tests with TestNG, you need the Eclipse [TestNG plugin](http://testng.org) [http://testng.org].

Since the examples in this guide are based on a Maven 2 project, you will also need the m2eclipse plugin. Instructions for using the m2eclipse update site to add the m2eclipse plugin to Eclipse are provided on the m2eclipse home page. For more, read the m2eclipse [reference guide](http://www.sonatype.com/books/m2eclipse-book/reference) [http://www.sonatype.com/books/m2eclipse-book/reference].

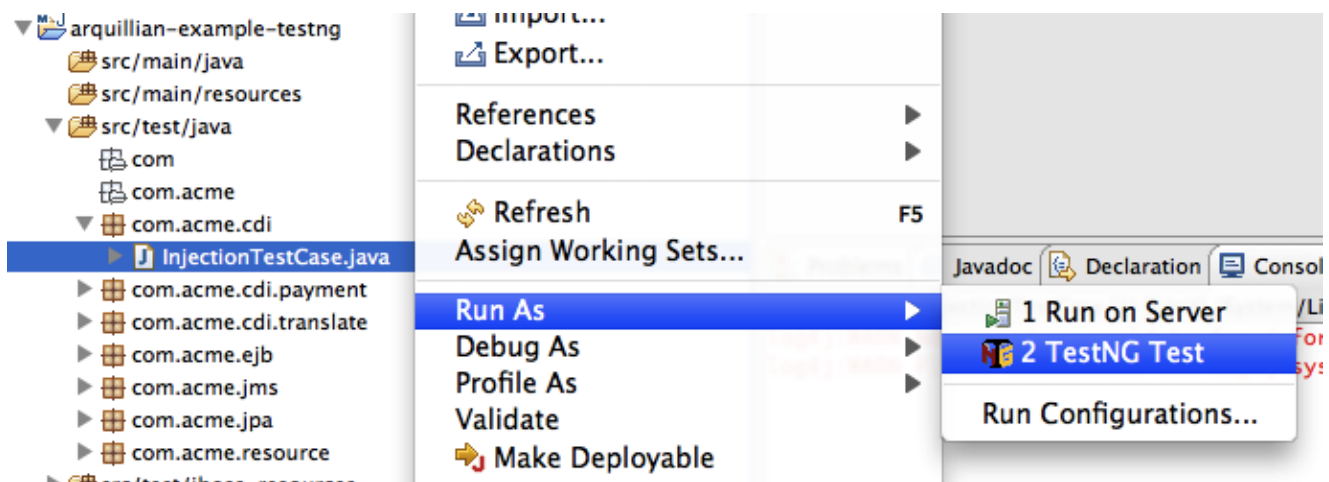
Once the plugins are installed, import your Maven project into the Eclipse workspace. Before executing the test, you need to enable the profile for the target container, as we did on the command line. We'll go ahead and activate the profile globally for the project (we also need the `default` profile, read the note above for more). Right click on the project and select Properties. Select the Maven property sheet and in the first form field, enter `jbossas-remote-6`; you also need to tell Maven to not resolve dependencies from the workspace (this interferes with resource loading):



Maven settings for project

Click OK and accept the project changes. Before we execute tests, make sure that Eclipse has properly processed all the resource files by running a full build on the project by selecting Clean from Project menu. Now you are ready to execute tests.

Assuming you have JBoss AS started from running the tests on the command line, you can now execute the tests. Right click on the InjectionTestCase.java file in the Package Explorer and select Run As... > JUnit Test or Run As... > TestNG Test depending on which unit testing framework the test is using.



Running the test from Eclipse using TestNG

You can now execute all the tests from Eclipse!

2.1. Testing an EJB

Here's a JUnit Arquillian test that validates the behavior of the EJB session bean `GreetingManager`. Arquillian looks up an instance of the EJB session bean in the test archive and injects it into the matching field type annotated with `@EJB`.

```
import javax.ejb.EJB;
import org.jboss.arquillian.api.Deployment;
import org.jboss.arquillian.junit.Arquillian;
import org.jboss.shrinkwrap.api.ShrinkWrap;
import org.jboss.shrinkwrap.api.spec.JavaArchive;
import org.junit.Assert;
import org.junit.Test;
import org.junit.runner.RunWith;

@RunWith(Arquillian.class)
public class InjectionTestCase {
    @Deployment
    public static JavaArchive createTestArchive() {
        return ShrinkWrap.create(JavaArchive.class, "test.jar")
            .addClasses(GreetingManager.class, GreetingManagerBean.class);
    }

    @EJB
    private GreetingManager greetingManager;

    @Test
    public void shouldBeAbleToInjectEJB() throws Exception {
```

```

    String userName = "Earthlings";
    Assert.assertEquals("Hello " + userName, greetingManager.greet(userName));
}
}

```

The TestNG version of this test looks identical, except that it extends the `org.jboss.arquillian.testng.Arquillian` class rather than being annotated with `@RunWith`.

2.2. Testing CDI beans

Here's an example of a JUnit Arquillian test that validates the `GreetingManager` EJB session bean again, but this time it's injected into the test class using the `@Inject` annotation. You could also make `GreenManager` a basic managed bean and inject it with the same annotation. The test also verifies that the CDI `BeanManager` instance is available and gets injected. Notice that to inject beans with CDI, you have to add a `beans.xml` file to the test archive.

```

import javax.enterprise.inject.spi.BeanManager;
import javax.inject.Inject;
import org.jboss.arquillian.api.Deployment;
import org.jboss.arquillian.junit.Arquillian;
import org.jboss.shrinkwrap.api.ShrinkWrap;
import org.jboss.shrinkwrap.api ArchivePaths;
import org.jboss.shrinkwrap.api.spec.JavaArchive;
import org.jboss.shrinkwrap.api.asset.EmptyAsset;
import org.junit.Assert;
import org.junit.Test;
import org.junit.runner.RunWith;
import com.acme.ejb.GreetingManager;
import com.acme.ejb.GreetingManagerBean;

@RunWith(Arquillian.class)
public class InjectionTestCase
{
    @Deployment
    public static JavaArchive createTestArchive() {
        return ShrinkWrap.create(JavaArchive.class, "test.jar")
            .addClasses(GreetingManager.class, GreetingManagerBean.class)
            .addAsManifestResource(EmptyAsset.INSTANCE, ArchivePaths.create("beans.xml"));
    }

    @Inject GreetingManager greetingManager;

    @Inject BeanManager beanManager;

```

```
@Test
public void shouldBeAbleToInjectCDI() throws Exception {
    String userName = "Earthlings";
    Assert.assertNotNull("Should have the injected the CDI bean manager", beanManager);
    Assert.assertEquals("Hello " + userName, greetingManager.greet(userName));
}
}
```

2.3. Testing JPA

In order to test JPA, you need both a database and a persistence unit. For the sake of example, let's assume we are going to use the default datasource provided by the container and that the tables will be created automatically when the persistence unit starts up. Here's a persistence unit configuration that satisfies that scenario.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0"
  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_1_0.xsd">
  <persistence-unit name="users" transaction-type="JTA">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <jta-data-source>java:/DefaultDS</jta-data-source>
    <properties>
      <property name="hibernate.hbm2ddl.auto" value="create-drop" />
      <property name="hibernate.dialect" value="org.hibernate.dialect.HSQLDialect" />
    </properties>
  </persistence-unit>
</persistence>
```

Now let's assume that we have an EJB session bean that injects a persistence context and is responsible for storing and retrieving instances of our domain class, `User`. We've catered it a bit to the test for purpose of demonstration.

```
public @Stateless class UserRepositoryBean implements UserRepository {
    @PersistenceContext EntityManager em;

    public void storeAndFlush(User u) {
        em.persist(u);
    }
}
```

```

    em.flush();
}

public List<User> findByLastName(String lastName) {
    return em.createQuery("select u from User u where u.lastName = :lastName")
        .setParameter("lastName", lastName)
        .getResultList();
}
}

```

Now let's create an Arquillian test to ensure we can persist and subsequently retrieve a user. Notice that we'll need to add the persistence unit descriptor to the test archive so that the persistence unit is booted in the test archive.

```

public class UserRepositoryTest extends Arquillian {
    @Deployment
    public static JavaArchive createTestArchive() {
        return ShrinkWrap.create(JavaArchive.class, "test.jar")
            .addClasses(User.class, UserRepository.class, UserRepositoryBean.class)
            .addAsManifestResource(
                "test-persistence.xml",
                ArchivePaths.create("persistence.xml"));
    }

    private static final String FIRST_NAME = "Agent";
    private static final String LAST_NAME = "Kay";

    @EJB
    private UserRepository userRepository;

    @Test
    public void testCanPersistUserObject() {
        User u = new User(FIRST_NAME, LAST_NAME);
        userRepository.storeAndFlush(u);

        List<User> users = userRepository.findByLastName(LAST_NAME);

        Assert.assertNotNull(users);
        Assert.assertTrue(users.size() == 1);

        Assert.assertEquals(users.get(0).getLastName(), LAST_NAME);
        Assert.assertEquals(users.get(0).getFirstName(), FIRST_NAME);
    }
}

```

```
}
```

2.4. Testing JMS

Here's another JUnit Arquillian test that exercises with JMS, something that may have previously seemed very tricky to test. The test uses a utility class `QueueRequestor` to encapsulate the low-level code for sending and receiving a message using a queue.

```
import javax.annotation.Resource;
import javax.jms.*;
import org.jboss.arquillian.api.Deployment;
import org.jboss.arquillian.junit.Arquillian;
import org.jboss.shrinkwrap.api.ShrinkWrap;
import org.jboss.shrinkwrap.api.spec.JavaArchive;
import org.junit.Assert;
import org.junit.Test;
import org.junit.runner.RunWith;
import com.acme.ejb.MessageEcho;
import com.acme.util.jms.QueueRequestor;

@RunWith(Arquillian.class)
public class InjectionTestCase {
    @Deployment
    public static JavaArchive createTestArchive() {
        return ShrinkWrap.create(JavaArchive.class, "test.jar")
            .addClasses(MessageEcho.class, QueueRequestor.class);
    }

    @Resource(mappedName = "/queue/DLQ")
    private Queue dlq;

    @Resource(mappedName = "/ConnectionFactory")
    private ConnectionFactory factory;

    @Test
    public void shouldBeAbleToSendMessage() throws Exception {

        String messageBody = "ping";

        Connection connection = factory.createConnection();
        Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
        QueueRequestor requestor = new QueueRequestor((QueueSession) session, dlq);
```

```
connection.start();

Message request = session.createTextMessage(messageBody);
Message response = requestor.request(request, 5000);

        Assert.assertEquals("Should have responded with same
message", messageBody, ((TextMessage) response).getText());
    }
}
```

That should give you a taste of what Arquillian tests look like. To learn how to setup Arquillian in your application and start developing tests with it, refer to the [Chapter 3, Getting started](#) chapter.

Getting started

We've promised you that integration testing with Arquillian is no more complicated than writing a unit test. Now it's time to prove it to you. In this chapter, we'll look at what is required to setup Arquillian in your project, how to write an Arquillian test case, how to execute the test case and how the test results are displayed. That sounds like a lot, but you'll be writing your own Arquillian tests in no time. (You'll also learn about [Chapter 7, Debugging remote tests](#) in Chapter 7).

3.1. Setting up Arquillian in a Maven project

The quickest way to get started with Arquillian is to add it to an existing Maven 2 project. Regardless of whether you plan to use Maven as your project build, we recommend that you take your first steps with Arquillian this way so as to get to your first green bar with the least amount of distraction.

The first thing you should do is define a Maven property for the version of Arquillian you are going to use. This way, you only have to maintain the version in one place and can reference it using the Maven variable syntax everywhere else in your build file.

```
<properties>
  <arquillian.version>1.0.0-SNAPSHOT</arquillian.version>
</properties>
```

Make sure you have the correct APIs available for your test. In this test we are going to use CDI:

```
<dependency>
  <groupId>javax.enterprise</groupId>
  <artifactId>cdi-api</artifactId>
  <version>1.0-SP1</version>
</dependency>
```

Next, you'll need to decide whether you are going to write tests in JUnit 4.x or TestNG 5.x. Once you make that decision (use TestNG if you're not sure), you'll need to add either the JUnit or TestNG library to your test build path as well as the corresponding Arquillian library.

If you plan to use *JUnit 4*, begin by adding the following two test-scoped dependencies to the `<dependencies>` section of your `pom.xml`.

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
```

```
<version>4.8.1</version>
<scope>test</scope>
</dependency>

<dependency>
  <groupId>org.jboss.arquillian</groupId>
  <artifactId>arquillian-junit</artifactId>
  <version>${arquillian.version}</version>
  <scope>test</scope>
</dependency>
```

If you plan to use *TestNG*, then add these two test-scoped dependencies instead:

```
<dependency>
  <groupId>org.testng</groupId>
  <artifactId>testng</artifactId>
  <version>5.12.1</version>
  <scope>test</scope>
</dependency>

<dependency>
  <groupId>org.jboss.arquillian</groupId>
  <artifactId>arquillian-testng</artifactId>
  <version>${arquillian.version}</version>
  <scope>test</scope>
</dependency>
```

That covers the libraries you need to write your first Arquillian test case. We'll revisit the `pom.xml` file in a moment to add the library you need to execute the test.

3.2. Writing your first Arquillian test

You're now going to write your first Arquillian test. But in order to write a test, we need to have something to test. So let's first create a managed bean that we can invoke.

We'll help out those Americans still trying to convert to the metric system by providing them a Fahrenheit to Celsius converter.

Here's our `TemperatureConverter`:

```
public class TemperatureConverter {

    public double convertToCelsius(double f) {
```

```

    return ((f - 32) * 5 / 9);
}

public double convertToFahrenheit(double c) {
    return ((c * 9 / 5) + 32);
}
}

```

Now we need to validate that this code runs. We'll be creating a test in the `src/test/java` classpath of the project.

Granted, in this trivial case, we could simply instantiate the implementation class in a unit test to test the calculations. However, let's assume that this bean is more complex, needing to access enterprise services. We want to test it as a full-blown container-managed bean, not just as a simple class instance. Therefore, we'll inject the bean into the test class using the `@Inject` annotation.

You're probably very familiar with writing tests using either JUnit or TestNG. A regular JUnit or TestNG test class requires two enhancements to make it an Arquillian integration test:

- Define the deployment archive for the test using `ShrinkWrap`
- Declare for the test to use the Arquillian test runner

The deployment archive for the test is defined using a static method annotated with Arquillian's `@Deployment` annotation that has the following signature:

```

public static Archive<?> methodName();

```

We'll add the managed bean to the archive so that we have something to test. We'll also add an empty `beans.xml` file, so that the deployment is CDI-enabled:

```

@Deployment
public static JavaArchive createTestArchive() {
    return ShrinkWrap.create(JavaArchive.class, "test.jar")
        .addClasses(TemperatureConverter.class)
        .addAsManifestResource(
            new ByteArrayAsset("<beans/>".getBytes()),
            ArchivePaths.create("beans.xml"));
}

```

The JUnit and TestNG versions of our test class will be nearly identical. They will only differ in how they hook into the Arquillian test runner.

When creating the JUnit version of the Arquillian test case, you will define at least one test method annotated with the JUnit `@Test` annotation and also annotate the class with the `@RunWith` annotation to indicate that Arquillian should be used as the test runner for this class.

Here's the JUnit version of our test class:

```
@RunWith(Arquillian.class)
public class TemperatureConverterTest {
    @Inject
    private TemperatureConverter converter;

    @Deployment
    public static JavaArchive createTestArchive() {
        return ShrinkWrap.create(JavaArchive.class, "test.jar")
            .addClasses(TemperatureConverter.class)
            .addAsManifestResource(
                EmptyAsset.INSTANCE,
                ArchivePaths.create("beans.xml"));
    }

    @Test
    public void testConvertToCelsius() {
        Assert.assertEquals(converter.convertToCelsius(32d), 0d);
        Assert.assertEquals(converter.convertToCelsius(212d), 100d);
    }

    @Test
    public void testConvertToFahrenheit() {
        Assert.assertEquals(converter.convertToFahrenheit(0d), 32d);
        Assert.assertEquals(converter.convertToFahrenheit(100d), 212d);
    }
}
```

TestNG doesn't provide anything like JUnit's `@RunWith` annotation, so instead the TestNG version of the Arquillian test case must extend the Arquillian class and define at least one method annotated with TestNG's `@Test` annotation.

```
public class TemperatureConverterTest extends Arquillian {
    @Inject
    private TemperatureConverter converter;

    @Deployment
    public static JavaArchive createTestArchive() {
```

```

return ShrinkWrap.create(JavaArchive.class, "test.jar")
    .addClasses(TemperatureConverter.class)
    .addAsManifestResource(
        EmptyAsset.INSTANCE,,
        ArchivePaths.create("beans.xml"));
}

@Test
public void testConvertToCelsius() {
    Assert.assertEquals(converter.convertToCelsius(32d), 0d);
    Assert.assertEquals(converter.convertToCelsius(212d), 100d);
}

@Test
public void testConvertToFahrenheit() {
    Assert.assertEquals(converter.convertToFahrenheit(0d), 32d);
    Assert.assertEquals(converter.convertToFahrenheit(100d), 212d);
}
}

```

As you can see, we are not instantiating the bean implementation class directly, but rather using the CDI reference provided by the container at the injection point, just as it would be used in the application. (If the target container supports EJB, you could replace the `@Inject` annotation with `@EJB`). Now let's see if this baby passes!

3.3. Setting up and running the test in Maven

As we've been emphasizing, this test is going to run inside of a container. That means you have to have a container running somewhere. While you can execute tests in an embedded container or a Java SE CDI environment, we're going to start off by testing using the real deal.

If you haven't already, download the latest version of JBoss AS 6.0 from the [JBoss AS download page](http://www.jboss.org/jbossas/downloads/) [http://www.jboss.org/jbossas/downloads/], extract the distribution and start the container.

Since Arquillian needs to perform JNDI lookups to get references to the components under test, we need to include a `jndi.properties` file on the test classpath. Create the file `src/test/resources/jndi.properties` and populate it with the following contents:

```

java.naming.factory.initial=org.jnp.interfaces.NamingContextFactory
java.naming.factory.url.pkgs=org.jboss.naming:org.jnp.interfaces
java.naming.provider.url=jnp://localhost:1099

```

Next, we're going to return to `pom.xml` to add another dependency. Arquillian picks which container it's going to use to deploy the test archive and negotiate test execution using the service provider

mechanism, meaning which implementation of the `DeployableContainer` SPI is on the classpath. We'll control that through the use of Maven profiles. Add the following profiles to `pom.xml`:

```
<profiles>
  <profile>
    <id>jbossas-remote-6</id>
    <dependencies>
      <dependency>
        <groupId>org.jboss.arquillian.container</groupId>
        <artifactId>arquillian-jbossas-remote-6</artifactId>
        <version>${arquillian.version}</version>
      </dependency>
      <dependency>
        <groupId>org.jboss.jbossas</groupId>
        <artifactId>jboss-as-client</artifactId>
        <version>6.0.0.Final</version>
        <type>pom</type>
      </dependency>
    </dependencies>
  </profile>
</profiles>
```

You would setup a similar profile for each Arquillian-supported container in which you want your tests executed.

All that's left is to execute the tests. In Maven, that's easy. Simply run the Maven test goal with the `jbossas-remote-6` profile activated:

```
mvn test -Pjbossas-remote-6
```

You should see that the two tests pass.

```
-----
T E S T S
-----
```

Running TemperatureConverterTest

Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 1.964 sec

Results :

Tests run: 2, Failures: 0, Errors: 0, Skipped: 0

```
[INFO] -----  
[INFO] BUILD SUCCESSFUL  
[INFO] -----
```

The tests are passing, but we don't see a green bar. To get that visual, we need to run the tests in the IDE. Arquillian tests can be executed using existing IDE plugins for JUnit and TestNG, respectively, or so you've been told. It's once again time to prove it.

3.4. Setting up and running the test in Eclipse

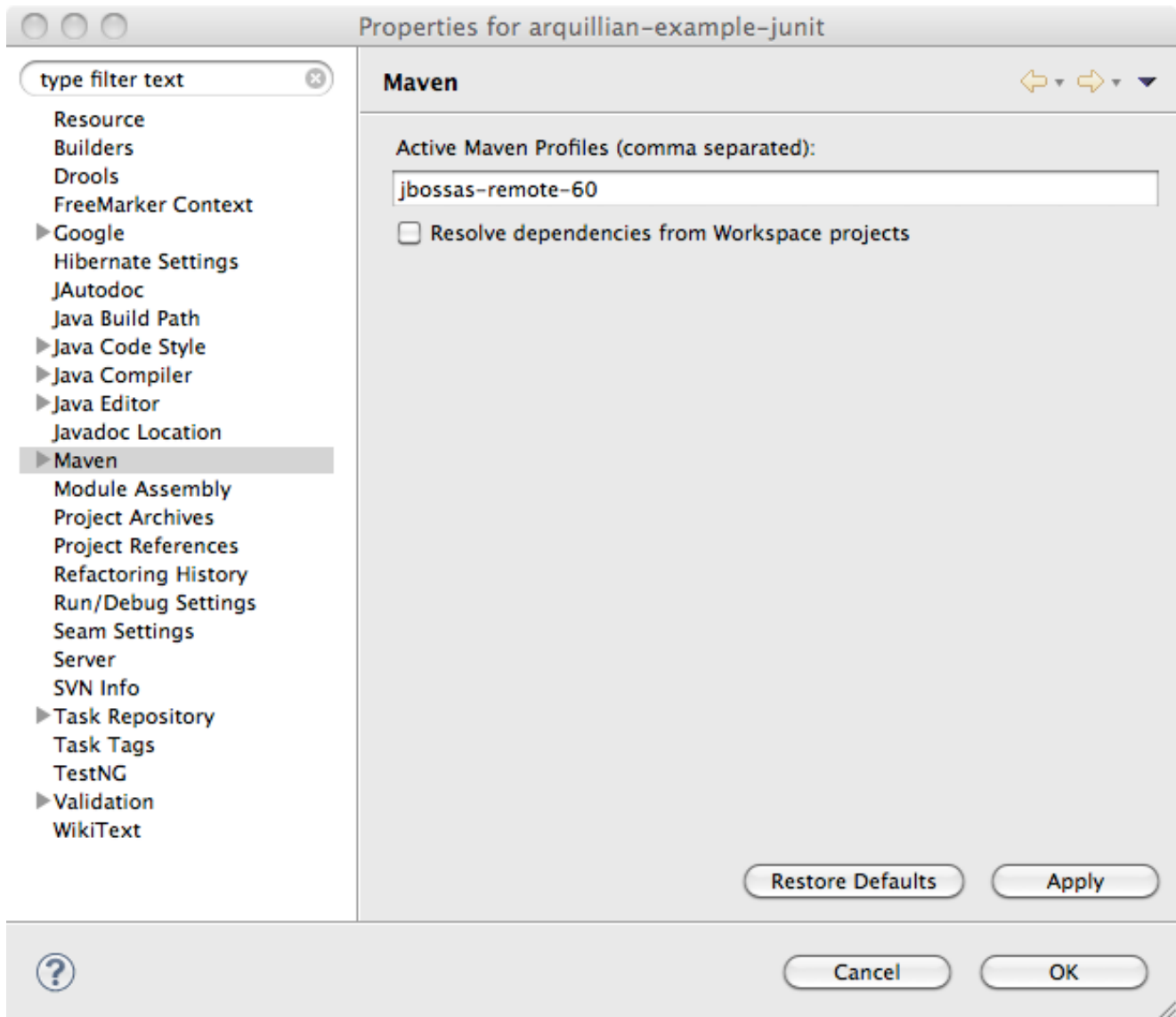
Before running an Arquillian test in Eclipse, you must have the plugin for the unit testing framework you are using installed. Eclipse ships with the JUnit plugin, so you are already setup if you selected JUnit. If you are writing your tests with TestNG, you need the Eclipse [TestNG plugin](http://testng.org) [http://testng.org].



Note

Since the example in this guide is based on a Maven 2 project, you will also need the m2eclipse plugin. Instructions for using the m2eclipse update site to add the m2eclipse plugin to Eclipse are provided on the m2eclipse home page. For more, read the m2eclipse [reference guide](http://www.sonatype.com/books/m2eclipse-book/reference) [http://www.sonatype.com/books/m2eclipse-book/reference].

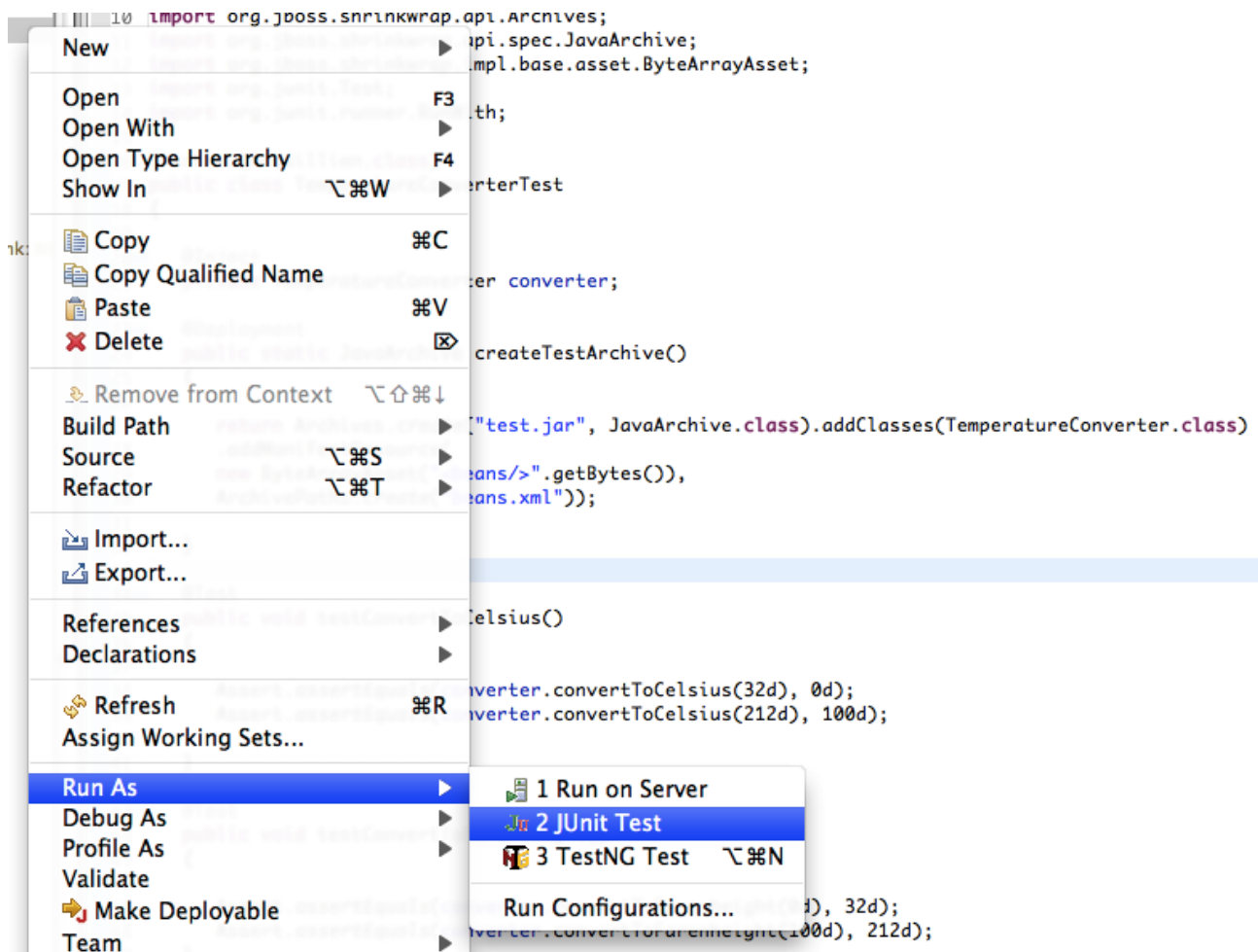
Once the plugins are installed, import your Maven project into the Eclipse workspace. Before executing the test, you need to enable the profile for the target container, as you did in the previous section. We'll go ahead and activate the profile globally for the project. Right click on the project and select Properties. Select the Maven property sheet and in the first form field, enter `jbossas-remote-6`; you also need to tell Maven to not resolve dependencies from the workspace (this interferes with resource loading):



Maven settings for project

Click OK and accept the project changes. Before we execute tests, make sure that Eclipse has properly processed all the resource files by running a full build on the project by selecting Clean from Project menu. Now you are ready to execute tests.

Right click on the TemperatureConverterTest.java file in the Package Explorer and select Run As... > JUnit Test or Run As... > TestNG Test depending on which unit testing framework the test is using.

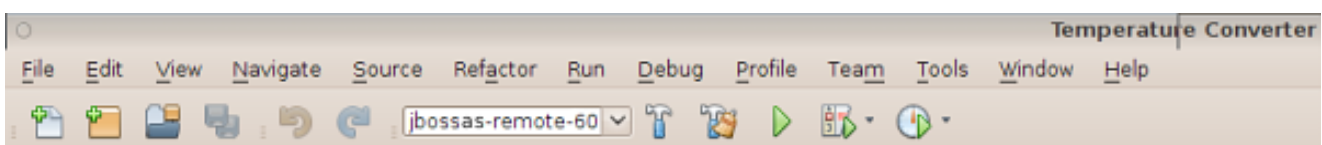


Running the the JUnit test in Eclipse

3.5. Setting up and running the test in NetBeans

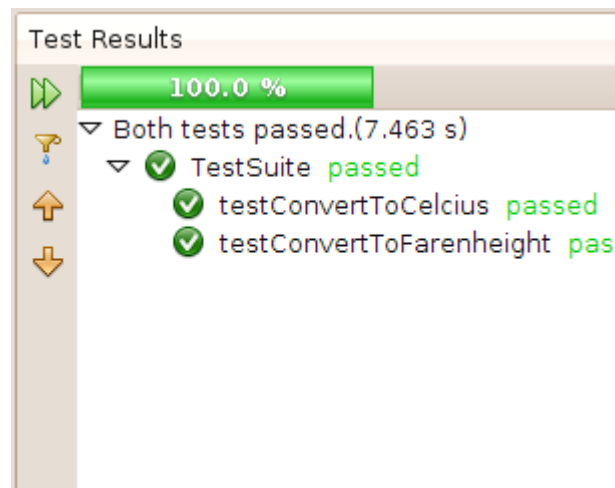
Things get even simpler when using NetBeans 6.8 or better. NetBeans ships with native Maven 2 support and, rather than including a test plugin for each unit testing framework, it has a generic test plugin which delegates to the Maven surefire plugin to execute the tests.

Import your Maven project into NetBeans. Then, look for a select menu in the main toolbar, which you can use to set the active Maven profile. Select the `jbossas-remote-6` profile as shown here:



NetBeans project configuration

Now you are ready to test. Simply right click on the `TemperatureConverter.java` file in the Projects pane and select `Test File`. NetBeans will delegate to the Maven surefire plugin to execute the tests and then display the results in a result window, showing us a pretty green bar!



Successful test report in NetBeans

As you can see, there was no special configuration necessary to execute the tests in either Eclipse or NetBeans.

Target containers

Arquillian's forte is not only in its ease of use, but also in its flexibility. Good integration testing is not just about testing in *any* container, but rather testing in the container *you* are targeting. It's all too easy to kid ourselves by validating components in a specialized testing container, only to realize that the small variations causes the components fail when it comes time to deploy to the application for real. To make tests count, you want to execute them in the real container.

Arquillian supports a variety of target containers out of the box, which will be covered in this chapter. If the container you are using isn't supported, Arquillian makes it very easy to plug in your own implementation.

4.1. Container varieties

You can run the same test case against various containers with Arquillian. The test class does not reference the container directly, which means you don't get locked into a proprietary test environment. It also means you can select the optimal container for development or easily test the compatibility of your application.

Arquillian recognizes three container interaction styles:

1. a *remote* container resides in a separate JVM from the test runner; Arquillian binds to the container to deploy and undeploy the test archive and invokes tests via a remote protocol (typically HTTP)
2. an *embedded* container resides in the same JVM as the test runner; lifecycle managed by Arquillian; tests are executed via a local protocol for containers without a web component (e.g., Embedded EJB) and via a remote protocol for containers that have a web component (e.g., Embedded Java EE)
3. a *managed* container is the same as a remote container, but in addition, its lifecycle (startup/shutdown) is managed by Arquillian and is run as a separate process

Containers can be further classified by their capabilities. There are three common categories:

1. A fully compliant Java EE application server (e.g., GlassFish, JBoss AS, Embedded GlassFish)
2. A Servlet container (e.g., Jetty, Tomcat)
3. A standalone bean container (e.g., Weld SE, Spring)

Arquillian provides SPIs that handle each of the tasks involved in controlling the runtime environment, executing the tests and aggregating the results. So in theory, you can support just about any environment that can be controlled with the set of hooks you are given.

4.2. Container management

While the management of an *embedded* container is straightforward, you may wonder how Arquillian knows where the remote and managed containers are installed. Actually, Arquillian only needs to know the install path of *managed* containers (e.g., jbossas-managed-6). In this case, since Arquillian manages the container process, it must have access to the container's startup script. For managed JBoss AS containers, the install path is read from the environment variable `JBOSS_HOME`.

For *remote* containers (e.g., jbossas-remote-6), Arquillian simply needs to know that the container is running and communicates with it using a remote protocol (e.g., JNDI). For remote JBoss AS containers, the JNDI settings are set in a `jndi.properties` file on the classpath. You also have to set the remote address and HTTP port in the container configuration if they differ from the default values (localhost and 8080 for JBoss AS, respectively).

4.3. Supported containers

The implementations provided so far are shown in the table below. Also listed is the artifactId of the JAR that provides the implementation. To execute your tests against a container, you must include the artifactId that corresponds to that container on the classpath. Use the following Maven profile definition as a template to add support for a container to your Maven build, replacing `%artifactId%` with the artifactId from the table. You then activate the profile when executing the tests just as you did in the [Chapter 3, Getting started](#) chapter.

```
<profile>
  <id>%artifactId%</id>
  <dependencies>
    <dependency>
      <groupId>org.jboss.arquillian.container</groupId>
      <artifactId>%artifactId%</artifactId>
      <version>${arquillian.version}</version>
    </dependency>
  </dependencies>
</profile>
```

Table 4.1. Target containers supported by Arquillian

Container name	Container type	Spec compliance	artifactId
JBoss AS 5	remote	Java EE 5	arquillian-jbossas-remote-5
JBoss AS 5.1	remote	Java EE 5	arquillian-jbossas-remote-5.1

Container name	Container type	Spec compliance	artifactId
<i>JBoss AS 5.1</i>	managed	Java EE 5	arquillian-jbossas-managed-5.1
<i>JBoss AS 6.0</i>	remote	Java EE 6	arquillian-jbossas-remote-6
<i>JBoss AS 6.0</i>	managed	Java EE 6	arquillian-jbossas-managed-6
<i>JBoss AS 6.0</i>	embedded	Java EE 6	arquillian-jbossas-embedded-6
<i>JBoss Reloaded 1.0</i>	embedded	JBoss MC	arquillian-reloaded-embedded-1
<i>GlassFish 3.1</i>	remote	Java EE 6	arquillian-glassfish-remote-3.1
<i>GlassFish 3.1</i>	embedded	Java EE 6	arquillian-glassfish-embedded-3.1
<i>Tomcat 6.0</i>	embedded	Servlet 2.5	arquillian-tomcat-embedded-6
<i>Jetty 6.1</i>	embedded	Servlet 2.5	arquillian-jetty-embedded-6.1
<i>Jetty 7.0</i>	embedded	Servlet ~3.0	arquillian-jetty-embedded-7
<i>Weld SE 1.0</i>	embedded	CDI	arquillian-weld-se-embedded-1
<i>Weld SE 1.1</i>	embedded	CDI	arquillian-weld-se-embedded-1.1
<i>Weld EE 1.1</i>	embedded	CDI	arquillian-weld-ee-embedded-1.1
<i>Apache OpenWebBeans 1.0</i>	embedded	CDI	arquillian-openwebbeans-embedded-1
<i>Apache OpenEJB 3.1</i>	embedded	EJB 3.0	arquillian-openejb-embedded-3.1

Support for other containers is planned, including Weblogic (remote), WebSphere (remote) and Hibernate.

4.4. Container configuration

You can come a long way with default values, but at some point you may need to customize some of the container settings to fit your environment. We're going to have a look at how this can be done

with Arquillian. Arquillian will look for a file named `arquillian.xml` in the root of your classpath. If it exists it will be auto loaded, else default values will be used. So this file is not a requirement.

Lets imagine that we're working for the company `example.com` and in our environment we have two servers; `test.example.com` and `hudson.example.com`. `test.example.com` is the JBoss instance we use for our integration tests and `hudson.example.com` is our continuous integration server that we want to run our integration suite from. By default, Arquillian will use localhost, so we need to tell it to use `test.example.com` to run the tests. The JBoss AS container by default use the Servlet protocol, so we have to override the default configuration.

```
<?xml version="1.0"?>

<arquillian xmlns="http://jboss.com/arquillian"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://jboss.org/schema/arquillian http://jboss.org/schema/arquillian/
arquillian_1_0.xsd">

  <container qualifier="jbossas" default="true">
    <configuration>
      <property name="providerUrl">jnp://test.example.com:1099</property>
    </configuration>
    <protocol type="Servlet 3.0">
      <configuration>
        <property name="host">test.example.com</property>
        <property name="port">8181</property>
      </configuration>
    </protocol>
  </container>
</arquillian>
```

That should do it! Here we use the JBoss AS 6.0 Remote container which default use the Servlet 3.0 protocol implementation. We override the default Servlet configuration to say that the http requests for this container can be executed over `test.example.com:8181`, but we also need to configure the container so it knows where to deploy our archives. We could for example have configured the Servlet protocol to communicate with a Apache server in front of the JBoss AS Server if we wanted to. Each container has different configuration options.



Tip

For a complete overview of all the containers and their configuration options, see [Chapter 12, Complete Container Reference](#)

Test enrichment

When you use a unit testing framework like JUnit or TestNG, your test case lives in a world on its own. That makes integration testing pretty difficult because it means the environment in which the business logic executes must be self-contained within the scope of the test case (whether at the suite, class or method level). The bonus of setting up this environment in the test falls on the developer's shoulders.

With Arquillian, you no longer have to worry about setting up the execution environment because that is all handled for you. The test will either be running in a container or a local CDI environment. But you still need some way to hook your test into this environment.

A key part of in-container integration testing is getting access the container-managed components that you plan to test. Using the Java new operator to instantiate the business class is not suitable in this testing scenario because it leaves out the declaratives services that get applied to the component at runtime. We want the real deal. Arquillian uses test enrichment to give us access to the real deal. The visible result of test enrichment is injection of container resources and beans directly into the test class.

5.1. Injection into the test case

Before Arquillian negotiates the execution of the test, it enriches the test class by satisfying injection points specified declaratively using annotations. There are three injection-based enrichers provided by Arquillian out of the box:

- `@Resource` - Java EE resource injections
- `@EJB` - EJB session bean reference injections
- `@Inject` - CDI injections

The first two enrichers use JNDI to lookup the instance to inject. The CDI injections are handled by treating the test class as a bean capable of receiving standard CDI injections.

The `@Resource` annotation gives you access to any object which is available via JNDI. It follows the standard rules for `@Resource` (as defined in the Section 2.3 of the Common Annotations for the Java Platform specification).

The `@EJB` annotation performs a JNDI lookup for the EJB session bean reference using the following equation in the specified order:

```
"java:global/test.ear/test/" + fieldType.getSimpleName() + "Bean",  
"java:global/test.ear/test/" + fieldType.getSimpleName(),  
"java:global/test/" + fieldType.getSimpleName(),  
"java:global/test/" + fieldType.getSimpleName() + "Bean",
```

```
"java:global/test/" + fieldType.getSimpleName() + "/no-interface",  
"test/" + unqualified interface name + "Bean/local",  
"test/" + unqualified interface name + "Bean/remote",  
"test/" + unqualified interface name + "/no-interface",  
unqualified interface name + "Bean/local",  
unqualified interface name + "Bean/remote",  
unqualified interface name + "/no-interface"
```

If no matching beans were found in those locations the injection will fail.



Warning

At the moment, the lookup for an EJB session reference relies on some common naming convention of EJB beans. In the future the lookup will rely on the standard JNDI naming conventions established in Java EE 6.

In order for CDI injections to work, the test archive defined with `ShrinkWrap` must be a bean archive. That means adding `beans.xml` to the `META-INF` directory. Here's a `@Deployment` method that shows one way to add a `beans.xml` to the archive:

```
@Deployment  
public static JavaArchive createTestArchive() {  
    return ShrinkWrap.create("test.jar", JavaArchive.class)  
        .addClass(NameOfClassUnderTest.class)  
        .addAsManifestResource(new ByteArrayAsset(new byte[0]), Paths.create("beans.xml"))  
}
```

In an application that takes full advantage of CDI, you can likely get by only using injections defined with the `@Inject` annotation. Regardless, the other two types of injection come in handy from time-to-time.

5.2. Active scopes

When running your tests the embedded Weld EE container, Arquillian activates scopes as follows:

- Application scope - Active for all methods in a test class
- Session scope - Active for all methods in a test class
- Request scope - Active for a single test method

Scope control is experimental at this point and may be altered in a future release of Arquillian.

Test execution

This chapter walks through the details of test execution, covering both the remote and local container cases.



Note

Whilst it's not necessary to understand the details of how Arquillian works, it is often useful to have some insight. This chapter gives you an overview of how Arquillian executes your test for you in your chosen container.

6.1. Anatomy of a test

In both JUnit 4 and TestNG 5, a test case is a class which contains at least one test method. The test method is designated using the `@Test` annotation from the respective framework. An Arquillian test case looks just like a regular JUnit or TestNG test case with two declarative enhancements:

- The class contains a static method annotated with `@Deployment` that returns a `JavaArchive`
- The class is annotated with `@RunWith(Arquillian.class)` (JUnit) or extends `Arquillian` (TestNG)

With those two modifications in place, the test is recognized by the Arquillian test runner and will be executed in the target container. It can also use the extra functionality that Arquillian provides—namely container resource injections and the injection of beans.

6.2. ShrinkWrap packaging

When the Arquillian test runner processes a test class, the first thing it does is retrieve the definition of the Java archive from the `@Deployment` method, appends the test class to the archive and packages the archive using ShrinkWrap.

The name of the archive is irrelevant, so the base name "test" is typically chosen (e.g., `test.jar`, `test.war`). Once you have created the shell of the archive, the sky is really the limit of how you can assemble it. You are customizing the layout and contents of the archive to suit the needs of the test. Essentially, you are creating a micro application in which to execute the code under test.

You can add the following artifacts to the test archive:

- Java classes
- A Java package (which adds all the Java classes in the package)
- Classpath resources

- File system resources
- A programmatically-defined file
- Java libraries (JAR files)
- Other Java archives defined by ShrinkWrap

Consult the [ShrinkWrap API](http://docs.jboss.org/shrinkwrap/1.0.0-alpha-11/) [http://docs.jboss.org/shrinkwrap/1.0.0-alpha-11/] to discover all the options you have available for constructing the test archive.

6.3. Test archive deployment

After the Arquillian test runner packages the test archive, it deploys it to the container. For a remote container, this means copying the archive to the hot deployment directory or deploying the archive using the container's remote deployment service. In the case of a local container, such as Weld SE, deploying the archive simply means registering the contents of the archive with the runtime environment.

How does Arquillian support multiple containers? And how are both remote and local cases supported? The answer to this question gets into the extensibility of Arquillian.

Arquillian delegates to an SPI (service provider interface) to handle starting and stopping the server and deploying and undeploying archives. In this case, the SPI is the interface `org.jboss.arquillian.spi.client.DeployableContainer`. If you recall from the getting started section, we included an Arquillian library according to the target container we wanted to use. That library contains an implementation of this interface, thus controlling how Arquillian handles deployment. If you wanted to introduce support for another container in Arquillian, you would simply provide an implementation of this interface.

With the archive deployed, all is left is negotiating execution of the test and capturing the results. As you would expect, once all the methods in the test class have been run, the archive is undeployed.

6.4. Enriching the test class

The last operation that Arquillian performs before executing the individual test methods is "enriching" the test class instance. This means hooking the test class to the container environment by satisfying its injection points. The enrichment is provided by any implementation of the `org.jboss.arquillian.spi.TestEnricher` SPI on the classpath. [Chapter 5, Test enrichment](#) details the injection points that Arquillian supports.

6.5. Negotiating test execution

The question at this point is, how does Arquillian negotiate with the container to execute the test when the test framework is being invoked locally? Technically the mechanism is pluggable using another SPI, `org.jboss.arquillian.spi.ContainerMethodExecutor`. Arquillian provides a default implementation for remote servers which uses HTTP communication

and an implementation for local tests, which works through direct execution of the test in the same JVM. Let's have a look at how the remote execution works.

The archive generator bundles and registers (in the `web.xml` descriptor) an `HttpServlet`, `org.jboss.arquillian.protocol.servlet.ServletTestRunner`, that responds to test execution GET requests. The test runner on the client side delegates to the `org.jboss.arquillian.spi.ContainerMethodExecutor` SPI implementation, which originates these test execution requests to transfer control to the container JVM. The name of the test class and the method to be executed are specified in the request query parameters named `className` and `methodName`, respectively.

When the test execution request is received, the servlet delegates to an implementation of the `org.jboss.arquillian.spi.TestRunner` SPI, passing it the name of the test class and the test method. `TestRunner` generates a test suite dynamically from the test class and method name and runs the suite (now within the context of the container).

The `ServletTestRunner` translates the native test result object of JUnit or TestNG into a `org.jboss.arquillian.spi.TestResult` and passes it back to the test executor on the client side by serializing the translated object into the response. The object gets encoded as either html or a serialized object, depending on the value of the `outputMode` request parameter that was passed to the servlet. Once the result has been transferred to the client-side test runner, the testing framework (JUnit or TestNG) wraps up the run of the test as though it had been executed in the same JVM.

Now you should have an understanding for how tests can be executed inside the container, but still be executed using existing IDE, Ant and Maven test plugins without any modification. Perhaps you have even started thinking about ways in which you can enhance or extend Arquillian. But there's still one challenge that remains for developing tests with Arquillian. How do you debug test? We'll look at how to hook a debugger into the test execution process in the next chapter.

6.6. Test run modes

So far, we've focused on testing your application internals, but we also want to test how others (people, or other programs) interact with the application. Typically, you want to make sure that every use case and execution path is fully tested. Third parties can interact with your application in a number of ways, for example web services, remote EJBs or via http. You need to check that you object serialization or networking work for instance.

This is why Arquillian comes with two run modes, `in container` and `as client`. `in container` is to test your application internals and `as client` is to test how your application is used by clients. Lets dive a bit deeper into the differences between the run modes and see how they effect your test execution and packaging.

6.6.1. Mode: in-container

```
@Deployment(testable = true)
```

As we mentioned above, we need to repackage your `@Deployment`, adding some Arquillian support classes, to run in-container. This gives us the ability to communicate with the test, enrich the test and run the test remotely. In this mode, the test executes in the remote container; Arquillian uses this mode by default.

See the [Complete Protocol Reference](#) for an overview of the expected output of the packaging process when you provide a `@Deployment`.

6.6.2. Mode: as-client

```
@Deployment(testable = false)
```

Now this mode is the easy part. As apposed to in-container mode which repackages and overrides the test execution, the as-client mode does as little as possible. It does not repackage your `@Deployment` nor does it forward the test execution to a remote server. Your test case is running in your JVM as expected and you're free to test the container from the outside, as your clients see it. The only thing Arquillian does is to control the lifecycle of your `@Deployment`.

Here is an example calling a Servlet using the `as client` mode.

```
@RunWith(Arquillian.class)
public class LocalRunServletTestCase
{
    @Deployment(testable = false)
    public static WebArchive createDeployment()
    {
        return ShrinkWrap.create("test.war", WebArchive.class)
            .addClass(TestServlet.class);
    }

    @Test
    public void shouldBeAbleToCallServlet(@ArquillianResource(TestServlet.class) URL baseUrl) throws Exception
    {
        // http://localhost:8080/test/
        String body = readAllAndClose(new URL(baseUrl, "/Test").openStream());

        Assert.assertEquals(
            "Verify that the servlet was deployed and returns the expected result",
            "hello",
            body);
    }
}
```

6.6.3. Mode: mixed

```
@Deployment(testable = true)
public static WebArchive create()
{
}

@Test // runs in container
public void shouldBeAbleToRunOnClientSide() throws Exception
{
}

@Test @RunAsClient // runs as client
public void shouldBeAbleToRunOnClientSide() throws Exception
{
}
```

It is also possible to mix the two run modes within the same test class. If you have defined the `Deployment` to be testable, you can specify the `@Test` method to use run mode as `client` by using the `@RunAsClient` annotation. This will allow two method within the same test class to run in different modes. This can be useful if you in a `run as client` mode want to execute against a remote endpoint in your application, for then in the next test method assert on server side state the remote endpoint might have created.



Tip

The effect of the different run modes depend on the `DeployableContainer` used. Both modes might seem to behave the same in some `Embedded` containers, but you should avoid mixing your internal and external tests. One thing is that they should test different aspects of your application and different usecases, another is that you will miss the benefits of switching `DeployableContainers` and run the same tests suite against a remote server if you do.

Debugging remote tests

While Arquillian tests can be easily executing using existing IDE, Ant and Maven test plugins, debugging tests are not as straightforward (but by no means difficult). The extra steps documented in this chapter are only relevant for tests which are not executed in the same JVM as the test runner. These steps do not apply to tests that are run in a local bean container (e.g., Weld SE), which can be debugged just like any other unit test.

We'll assume in this chapter that you are already using Eclipse and you already have the test plugin installed for the testing framework you are using (JUnit or TestNG).

7.1. Debugging in Eclipse

If you set a break point and execute the test in debug mode using a remote container, your break point won't be hit. That's because when you debug an in-container test, you're actually debugging the container. The test runner and the test are executing in different JVMs. Therefore, to setup debugging, you must first attach the IDE debugger to the container, then execute the test in debug mode (i.e., debug as test). That puts the debugger on both sides of the fence, so to speak, and allows the break point to be discovered.

Let's begin by looking at how to attach the IDE debugger to the container. This isn't specific to Arquillian. It's the same setup you would use to debug a deployed application.

7.1.1. Attaching the IDE debugger to the container

There are two ways to attach the IDE debugger to the container. You can either start the container in debug mode from within the IDE, or you can attach the debugger over a socket connection to a standalone container running with JPDA enabled.

The Eclipse Server Tools, a subproject of the Eclipse Web Tools Project (WTP), has support for launching most major application servers, including JBoss AS 5. However, if you are using JBoss AS, you should consider using JBoss Tools instead, which offers tighter integration with JBoss technologies. See either the [Server Tools documentation](http://www.eclipse.org/webtools/server/server.php) [http://www.eclipse.org/webtools/server/server.php] or the [JBoss Tools documentation](http://docs.jboss.org/tools/3.0.1.GA/en/as/html/index.html) [http://docs.jboss.org/tools/3.0.1.GA/en/as/html/index.html] for instructions on how to setup a container and start it in debug mode.

See [this blog entry](http://maverikpro.wordpress.com/2007/11/26/remote-debug-a-web-application-using-eclipse) [http://maverikpro.wordpress.com/2007/11/26/remote-debug-a-web-application-using-eclipse] to learn how to start JBoss AS with JPDA enabled and how to get the Eclipse debugger to connect to the remote process.

7.1.1.1. Starting JBoss AS in debug mode

If you are using JBoss AS, the quickest way to setup debug mode is to add the following line to the end of \$JBOSS_AS_HOME/bin/run.conf (Unix/Linux):

```
JAVA_OPTS="$JAVA_OPTS
```

```
-Xrunjdwp:transport=dt_socket,address=8787,server=y,suspend=n"
```

or before the line `:JAVA_OPTS_SET` in `$JBOSS_AS_HOME/bin/run.conf.bat` (Windows)

```
set                JAVA_OPTS="%JAVA_OPTS%                -  
Xrunjdwp:transport=dt_socket,address=8787,server=y,suspend=n"
```

Keep in mind your container will always run with debug mode enabled after making this change. You might want to consider putting some logic in the `run.conf*` file.

7.1.2. Launching the test in debug mode

Once Eclipse is debugging the container, you can set a breakpoint in the test and debug it just like a unit test. Let's give it a try.

Open an Arquillian test in the Java editor, right click in the editor view, and select `Debug As > TestNG` (or `JUnit`) `Test`. When the IDE hits the breakpoint, it halts the JVM thread of the container rather than the thread that launched the test. You are now debugging remotely.

7.1.3. Stepping into external libraries

If you plan to step into a class in an external library (code outside of your application), you must ensure that the source is properly associated with the library. Below are the steps to follow to associate the source of a library with the debug configuration:

1. Select the `Run > Debug Configurations...` menu from the main menubar
2. Select the name of the test class in the `TestNG` (or `JUnit`) category
3. Select the `Source` tab
4. Click the `Add...` button on the right
5. Select `Java Project`
6. Check the project the contains the class you want to debug
7. Click `OK` on the `Project Selection` window
8. Click `Close` on the `Debug Configurations` window

You'll have to complete those steps for any test class you are debugging, though you only have to do it once (the debug configuration hangs around indefinitely).

**Tip**

These steps may not be necessary if you have a Maven project and the sources for the library are available in the Maven repository.

7.2. Assertions in remote tests

The first time you try Arquillian, you may find that assertions that use the Java assert keyword are not working. Keep in mind that the test is not executing the same JVM as the test runner.

In order for the Java keyword "assert" to work you have to enable assertions (using the -ea flag) in the JVM that is running the container. You may want to consider specifying the package names of your test classes to avoid assertions to be enabled throughout the container's source code.

7.2.1. Enabling assertions in JBoss AS

If you are using JBoss AS, the quickest way to setup debug mode is to add the following line to the end of \$JBOSS_AS_HOME/bin/run.conf (Unix/Linux):

```
JAVA_OPTS="$JAVA_OPTS -ea"
```

or before the line :JAVA_OPTS_SET in \$JBOSS_AS_HOME/bin/run.conf.bat (Windows)

```
set "JAVA_OPTS=%JAVA_OPTS% -ea"
```

Keep in mind your container will always run with assertions enabled after making this change. You might want to consider putting some logic in the run.conf* file.

As an alternative, we recommend using the 'Assert' object that comes with your test framework instead to avoid the whole issue. Also keep in mind that if you use System.out.println statements, the output is going to show up in the log file of the container rather than in the test output.

Build system integration

Just because the Arquillian project uses Maven doesn't mean you have to use it to run your Arquillian tests. Arquillian is designed to have seamless integration with JUnit and TestNG without any necessary test framework configuration. That means you can use any build system that has a JUnit or TestNG task to execute your Arquillian test cases. Since most of this guide focuses on using Arquillian in a Maven build, this chapter is going to be about alternative build systems, namely Gradle and Ant.

8.1. Arquillian's active build ingredient

The secret ingredient required to activate the Arquillian test runner is getting the correct libraries on the classpath. (Often easier said than done). The libraries consist of the Arquillian container integration and the container runtime (for an embedded container) or deployment client (for a remote container).

In general, the steps to incorporate Arquillian into a build, regardless of what build tool you are using, can be summarized as:

1. Activate/configure the JUnit or TestNG task/plugin
2. Add the Arquillian container integration to the test classpath
(e.g., `org.jboss.arquillian.container:arquillian-%VENDOR%-%TYPE%-%VERSION%`)
3. Add the container runtime (embedded) or deployment client (remote) to the classpath
4. Execute the test build task/goal

If you are only running the Arquillian tests on a single container, this setup is exceptionally straightforward. The challenge comes when you want to run the tests on multiple containers. It's really just a matter of putting the correct libraries on the test classpath, though.

For some build systems, isolating multiple classpath definitions is more tricky than others. For instance, in Maven, you only get one test classpath per run (without using really advanced plugin configuration). You can toggle between different test classpath pairings through the use of profiles. Each profile contains the libraries for a single target container (a combination of the libraries itemized in steps 2 and 3 above). You'll see this strategy used in the Arquillian examples.

Other build tools, such as Gradle, can easily define new test tasks that each have their own, unique classpath. This makes it not only possible to separate out the target containers, but also run the tests against each one in the same build execution. We'll see an example of that later in this chapter. Gradle can also emulate the Maven profile strategy through the use of build fragment imports. We'll also show an example of that approach for contrast.

8.2. Integrating Arquillian into a Gradle build

[Gradle](http://gradle.org) [http://gradle.org] is a build tool that allows you to create declarative, maintainable, concise and highly-performing builds. More importantly, in this context, Gradle gives you all the freedom you need instead of imposing a rigid build lifecycle on you. You'll get a glimpse of just how flexible Gradle can be by learning how to integrate Arquillian into a Gradle build.

We'll be contrasting two strategies for running Arquillian tests from Gradle:

1. Container-specific test tasks
2. Test "profiles" via build fragment imports

The first strategy is the recommended one since it gives you the benefit of being able to run your tests on multiple containers in the same build. However, the second approach is less esoteric and will be more familiar to Maven users. Of course, Gradle is so flexible that there are likely other solutions for this problem. We invite you to give us feedback if you find a better way (or another way worth documenting).

Let's get the common build stuff out of the way, then dive into the two strategies listed above.

8.2.1. apply from: common

The simplest Gradle build for a Java project is a sparse one line.

```
apply plugin: JavaPlugin
```

Put this line into a file named build.gradle at the root of the project, which is the standard location of the Gradle build file. (Perhaps after seeing this configuration you'll understand the reference in the section title).

Next we'll add the Maven Central and JBoss Community repository definitions, so that we can pull down dependent libraries. The latter repository hosts the Arquillian artifacts.

```
apply plugin: JavaPlugin

repositories {
    mavenCentral()
    mavenRepo urls: 'http://repository.jboss.org/nexus/content/groups/public'
}
```

If your SCM (e.g., SVN, Git) is already ignoring the target directory, you may want to move the Gradle build output underneath this folder, rather than allowing Gradle to use its default build directory, build. Let's add that configuration to the common build logic as well:

```

apply plugin: JavaPlugin

buildDir = 'target/gradle-build'

repositories {
    mavenCentral()
    mavenRepo urls: 'http://repository.jboss.org/nexus/content/groups/public'
}

```



Warning

If you are using Gradle alongside Maven, you shouldn't set the buildDir to target since Gradle organizes compiled classes different than Maven does, possibly leading to conflicts (Though, the behavior of Gradle can also be customized).

We also recommend that you centralize version numbers at the top of your build to make upgrading your dependency easy. This list will grow as you add other containers, but we'll seed the list for the examples below:

```

apply plugin: JavaPlugin

buildDir = 'target/gradle-build'

libraryVersions = [
    junit: '4.8.1', arquillian: '1.0.0.Alpha4', jbossJavaeeSpec: '1.0.0.Beta7', weld: '1.0.1-Final',
    slf4j: '1.5.8', log4j: '1.2.14', jbossas: '6.0.0.Final', glassfish: '3.0.1-b20', cdi: '1.0-SP1'
]

...

```

We also need to add the unit test library (JUnit or TestNG) and the corresponding Arquillian integration:

```

dependencies {
    testCompile group: 'junit', name: 'junit', version: libraryVersions.junit
    testCompile group: 'org.jboss.arquillian', name: 'arquillian-junit', version:
    libraryVersions.arquillian
}

```

In this example, we'll assume the project is compiling against APIs that are provided by the target container runtime, so we need to add a dependency configuration (aka scope) to include libraries on the compile classpath but excluded from the runtime classpath. In the future, Gradle will include support for such a scope. Until then, we'll define one ourselves in the configurations closure.

```
configurations {  
    compileOnly  
}
```

We also need to add the dependencies associated with that configuration to the compile classpaths using the sourceSets closure:

```
sourceSets {  
    main {  
        compileClasspath = configurations.compile + configurations.compileOnly  
    }  
    test {  
        compileClasspath = compileClasspath + configurations.compileOnly  
    }  
}
```

Here's the Gradle build all together now:

```
apply plugin: JavaPlugin  
  
buildDir = 'target/gradle-build'  
  
libraryVersions = [  
    junit: '4.8.1', arquillian: '1.0.0.Alpha3', jbossJavaeeSpec: '1.0.0.Beta7', weld: '1.0.1-Final',  
    slf4j: '1.5.8', log4j: '1.2.14', jbossas: '6.0.0.Final', glassfish: '3.0.1-b20', cdi: '1.0-SP1'  
]  
  
repositories {  
    mavenCentral()  
    mavenRepo url: 'http://repository.jboss.org/nexus/content/groups/public'  
}  
  
configurations {  
    compileOnly  
}
```

```
sourceSets {
    main {
        compileClasspath = configurations.compile + configurations.compileOnly
    }
    test {
        compileClasspath = compileClasspath + configurations.compileOnly
    }
}
```

Now that the foundation of a build is in place (or you've added these elements to your existing Gradle build), we are ready to configuring the container-specific test tasks. In the first approach, we'll create a unique dependency configuration and task for each container.

8.2.2. Strategy #1: Container-specific test tasks

Each project in Gradle is made up of one or more tasks. A task represents some atomic piece of work which a build performs. Examples include compiling classes, *executing tests*, creating a JAR, publishing an artifact to a repository. We are interested in the executing tests task. But it's not necessarily just a single test task. Gradle allows you to define any number of test tasks, each having its own classpath configuration. We'll use this to configure test executions for each container.

Let's assume that we want to run the tests against the following three Arquillian-supported containers:

- Weld EE Embedded 1.1
- Remote JBoss AS 6
- Embedded GlassFish 3

We'll need three components for each container:

1. Dependency configuration (scope)
2. Runtime dependencies
3. Custom test task

We'll start with the Weld EE Embedded container. Starting from the Gradle build defined in the previous section, we first define a configuration for the test runtime dependencies.

```
configurations {
    compileOnly
    weldEmbeddedTestRuntime { extendsFrom testRuntime }
}
```

Next we add the dependencies for compiling against the Java EE API and running Arquillian tests in the Weld EE Embedded container:

```
dependencies {
    compileOnly group: 'javax.enterprise', name: 'cdi-api', version: libraryVersions.cdi

    testCompile group: 'junit', name: 'junit', version: libraryVersions.junit
        testCompile group: 'org.jboss.arquillian', name: 'arquillian-junit', version:
libraryVersions.arquillian

    // temporarily downgrade the weld-ee-embedded-1.1 container
    weldEmbeddedTestRuntime group: 'org.jboss.arquillian.container', name: 'arquillian-weld-ee-
embedded-1.1', version: '1.0.0.Alpha3'
    weldEmbeddedTestRuntime group: 'org.jboss.spec', name: 'jboss-javaee-6.0', version:
libraryVersions.jbossJavaeeSpec
    weldEmbeddedTestRuntime group: 'org.jboss.weld', name: 'weld-core', version:
libraryVersions.weld
    weldEmbeddedTestRuntime group: 'org.slf4j', name: 'slf4j-log4j12', version: libraryVersions.slf4j
    weldEmbeddedTestRuntime group: 'log4j', name: 'log4j', version: libraryVersions.log4j
}
```

Finally, we define the test task:

```
task weldEmbeddedTest(type: Test) {
    testClassesDir = sourceSets.test.classesDir
    classpath = sourceSets.test.classes + sourceSets.main.classes +
configurations.weldEmbeddedTestRuntime
}
```

This task will execute in the lifecycle setup by the Java plugin in place of the normal test task. You run it as follows:

```
gradle weldEmbeddedTest
```

Or, more simply:

```
gradle wET
```

Now we just repeat this setup for the other containers.



Tip

Since you are creating custom test tasks, you likely want to configure the default test task to either exclude Arquillian tests or to use a default container, perhaps Weld EE Embedded in this case.

Here's the full build file with the tasks for our three target containers:

```
apply plugin: JavaPlugin

buildDir = 'target/gradle-build'

libraryVersions = [
    junit: '4.8.1', arquillian: '1.0.0.Alpha4', jbossJavaeeSpec: '1.0.0.Beta7', weld: '1.0.1-Final',
    slf4j: '1.5.8', log4j: '1.2.14', jbossas: '6.0.0.Final', glassfish: '3.0.1-b20', cdi: '1.0-SP1'
]

repositories {
    mavenCentral()
    mavenRepo url: 'http://repository.jboss.org/nexus/content/groups/public'
    mavenRepo url: 'http://repository.jboss.org/nexus/content/repositories/deprecated'
}

configurations {
    compileOnly
    weldEmbeddedTestRuntime { extendsFrom testRuntime }
    jbossasRemoteTestRuntime { extendsFrom testRuntime, compileOnly }
    glassfishEmbeddedTestRuntime { extendsFrom testRuntime }
}

dependencies {
    compileOnly group: 'javax.enterprise', name: 'cdi-api', version: libraryVersions.cdi

    testCompile group: 'junit', name: 'junit', version: libraryVersions.junit
        testCompile group: 'org.jboss.arquillian', name: 'arquillian-junit', version:
libraryVersions.arquillian

    // temporarily downgrade the weld-ee-embedded-1.1 container
    weldEmbeddedTestRuntime group: 'org.jboss.arquillian.container', name: 'arquillian-weld-ee-
embedded-1.1', version: '1.0.0.Alpha3'
    weldEmbeddedTestRuntime group: 'org.jboss.spec', name: 'jboss-javaee-6.0', version:
libraryVersions.jbossJavaeeSpec
}
```

```

        weldEmbeddedTestRuntime group: 'org.jboss.weld', name: 'weld-core', version:
libraryVersions.weld
        weldEmbeddedTestRuntime group: 'org.slf4j', name: 'slf4j-log4j12', version: libraryVersions.slf4j
        weldEmbeddedTestRuntime group: 'log4j', name: 'log4j', version: libraryVersions.log4j

        jbossasRemoteTestRuntime group: 'org.jboss.arquillian.container', name: 'arquillian-jbossas-
remote-6', version: libraryVersions.arquillian
        jbossasRemoteTestRuntime group: 'org.jboss.jbossas', name: 'jboss-as-server', classifier:
'client', version: libraryVersions.jbossas, transitive: false
        jbossasRemoteTestRuntime group: 'org.jboss.jbossas', name: 'jboss-as-profileservice',
classifier: 'client', version: libraryVersions.jbossas

        glassfishEmbeddedTestRuntime group: 'org.jboss.arquillian.container', name: 'arquillian-
glassfish-embedded-3', version: libraryVersions.arquillian
        glassfishEmbeddedTestRuntime group: 'org.glassfish.extras', name: 'glassfish-embedded-all',
version: libraryVersions.glassfish
    }

    sourceSets {
        main {
            compileClasspath = configurations.compile + configurations.compileOnly
        }
        test {
            compileClasspath = compileClasspath + configurations.compileOnly
        }
    }

    task weldEmbeddedTest(type: Test) {
        testClassesDir = sourceSets.test.classesDir
        classpath = sourceSets.test.classes + sourceSets.main.classes +
configurations.weldEmbeddedTestRuntime
    }

    task jbossasRemoteTest(type: Test) {
        testClassesDir = sourceSets.test.classesDir
        classpath = sourceSets.test.classes + sourceSets.main.classes + files('src/test/resources-
jbossas') + configurations.jbossasRemoteTestRuntime
    }

    task glassfishEmbeddedTest(type: Test) {
        testClassesDir = sourceSets.test.classesDir
        classpath = sourceSets.test.classes + sourceSets.main.classes +
configurations.glassfishEmbeddedTestRuntime
    }

```

```
}
```

**Note**

Notice we've added an extra resources directory for remote JBoss AS 6 to include the required jndi.properties file. That's a special configuration for the remote JBoss AS containers, though won't be required after Arquillian 1.0.0.Alpha4.

It's now possible to run the Arquillian tests against each of the three containers in sequence using this Gradle command (make sure a JBoss AS is started in the background):

```
gradle weldEmbeddedTest jbossasRemoteTest glassfishEmbeddedTest
```

Pretty cool, huh?

Now let's look at another way to solve this problem.

8.2.3. Strategy #2: Test profiles

Another way to approach integrating Arquillian into a Gradle build is to emulate the behavior of Maven profiles. In this case, we won't be adding any extra tasks, rather overriding the Java plugin configuration and provided tasks.

**Note**

A Maven profile effectively overrides portions of the build configuration and is activated using a command option (or some other profile activation setting).

Once again, let's assume that we want to run the tests against the following three Arquillian-supported containers:

- Weld EE Embedded 1.1
- Remote JBoss AS 6
- Embedded GlassFish 3

All we need to do is customize the test runtime classpath for each container. First, let's setup the common compile-time dependencies in the main build file:

```
apply plugin: JavaPlugin
```

```
buildDir = 'target/gradle-build'
```

```
libraryVersions = [
    junit: '4.8.1', arquillian: '1.0.0.Alpha3', jbossJavaeeSpec: '1.0.0.Beta7', weld: '1.0.1-Final',
    slf4j: '1.5.8', log4j: '1.2.14', jbossas: '6.0.0.Final', glassfish: '3.0.1-b20', cdi: '1.0-SP1'
]

repositories {
    mavenCentral()
    mavenRepo urls: 'http://repository.jboss.org/nexus/content/groups/public'
}

configurations {
    compileOnly
}

dependencies {
    group: 'org.jboss.spec', name: 'jboss-javaee-6.0', version: libraryVersions.jbossJavaeeSpec
}

sourceSets {
    main {
        compileClasspath = configurations.compile + configurations.compileOnly
    }
    test {
        compileClasspath = compileClasspath + configurations.compileOnly
    }
}
```

We then need to create a partial Gradle build file for each container that contains the container-specific dependencies and configuration. Let's start with Weld EE Embedded.

Create a file named `weld-ee-embedded-profile.gradle` and populate it with the following contents:

```
dependencies {
    // temporarily downgrade the weld-ee-embedded-1.1 container
    testRuntime group: 'org.jboss.arquillian.container', name: 'arquillian-weld-ee-embedded-1.1',
    version: '1.0.0.Alpha3'
    testRuntime group: 'org.jboss.spec', name: 'jboss-javaee-6.0', version:
    libraryVersions.jbossJavaeeSpec
    testRuntime group: 'org.jboss.weld', name: 'weld-core', version: libraryVersions.weld
    testRuntime group: 'org.slf4j', name: 'slf4j-log4j12', version: libraryVersions.slf4j
    testRuntime group: 'log4j', name: 'log4j', version: libraryVersions.log4j
}
```

Here's the partial build file for Remote JBoss AS, named `jbossas-remote-profile.gradle`:

```
dependencies {
    testRuntime group: 'javax.enterprise', name: 'cdi-api', version: libraryVersions.cdi
    testRuntime group: 'org.jboss.arquillian.container', name: 'arquillian-jbossas-remote-6', version:
libraryVersions.arquillian
    testRuntime group: 'org.jboss.jbossas', name: 'jboss-as-server', classifier: 'client', version:
libraryVersions.jbossas, transitive: false
    testRuntime group: 'org.jboss.jbossas', name: 'jboss-as-profileservice', classifier: 'client', version:
libraryVersions.jbossas
}

test {
    classpath = sourceSets.test.classes + sourceSets.main.classes + files('src/test/resources-
jbossas') + configurations.testRuntime
}
```

And finally the one for Embedded GlassFish, named `glassfish-embedded-profile.gradle`:

```
dependencies {
    testRuntime group: 'org.jboss.arquillian.container', name: 'arquillian-glassfish-embedded-3',
version: libraryVersions.arquillian
    testRuntime group: 'org.glassfish.extras', name: 'glassfish-embedded-all', version:
libraryVersions.glassfish
}
```

Now we need to import the appropriate partial Gradle build into the main build. The file will be selected based on the value of the project property named `profile`.

```
apply plugin: JavaPlugin

buildDir = 'target/gradle-build'

libraryVersions = [
    junit: '4.8.1', arquillian: '1.0.0.Alpha4', jbossJavaeeSpec: '1.0.0.Beta7', weld: '1.0.1-Final',
    slf4j: '1.5.8', log4j: '1.2.14', jbossas: '6.0.0.Final', glassfish: '3.0.1-b20', cdi: '1.0-SP1'
]

apply from: profile + '-profile.gradle'

repositories {
```

```
mavenCentral()
mavenRepo urls: 'http://repository.jboss.org/nexus/content/groups/public'
}

configurations {
    compileOnly
}

dependencies {
    compileOnly group: 'javax.enterprise', name: 'cdi-api', version: libraryVersions.cdi

    testCompile group: 'junit', name: 'junit', version: libraryVersions.junit
        testCompile group: 'org.jboss.arquillian', name: 'arquillian-junit', version:
libraryVersions.arquillian
}

sourceSets {
    main {
        compileClasspath = configurations.compile + configurations.compileOnly
    }
    test {
        compileClasspath = compileClasspath + configurations.compileOnly
    }
}
```

Tests are run in the Weld EE Embedded runtime using this command:

```
gradle test -Pprofile=weld-ee-embedded
```

That's pretty much the same experience you get when you use Maven (and a whole heck of a lot simpler).

While the configuration is much simpler using the profiles strategy, there are two things to keep in mind:

1. It crosses over into more than one build file
2. You cannot run the tests in each container in a single build execution

If you have a better idea of how to integrate an Arquillian test suite into a Gradle build, we'd love to hear it on the [Arquillian discussion forums](http://community.jboss.org/en/arquillian) [http://community.jboss.org/en/arquillian].

8.3. Integrating Arquillian into an Ant (+Ivy) build

See the [CDI subproject](http://github.com/mojavelinux/arquillian-showcase/tree/master/cdi/) [http://github.com/mojavelinux/arquillian-showcase/tree/master/cdi/] of the Arquillian showcase for an Ant+Ivy build example until this section is written.

Advanced use cases

This chapter walks through some more advanced features and use cases you can have Arquillian do for you.

9.1. Descriptor deployment

We have previously seen Arquillian deploy ShrinkWrap Archives, but some times you need to deploy other items like a JMS Queue or a DataSource for your test to run. This can be done by using a ShrinkWrap sub project called ShrinkWrap Descriptors. Just like you would deploy a Archive you can deploy a `Descriptor`.

```
@Deployment(order = 1)
public static Descriptor createDep1()
{
    return Descriptors.create(DataSourceDescriptor.class);
}

@Deployment(order = 2)
public static WebArchive createDep2() {}

@Test
public void testDataBase() {}
```

9.2. Resource injection

When dealing with multiple different environments and hidden dynamic container configuration you very soon come to a point where you need access to the backing containers ip/port/context information. This is especially useful when doing remote end point testing. So instead of trying to setup all containers on the same ip/port/context, or hard code this in your test, Arquillian provides something we call `@ArquillianResource` injection. Via this injection point we can expose multiple internal object.

When you need to get a hold of the HTTP context your `Deployment` defined, you can use `@ArquillianResource` on a field or method argument of type `URL`.

```
@ArquillianResource
private URL baseUrl;

@ArquillianResource(MyServlet.class)
private URL baseUrlServerURL;
```

```
@Test
private void shouldDoX(@ArquillianResource(MyServlet.class) URL baseUrl)
{
}
```

9.3. Multiple Deployments

Sometimes a single `Deployment` is not enough, and you need to specify more than one to get your test done. Maybe you want to test communication between two different web applications? Arquillian supports this as well. Simple just add more `@Deployment` methods to the test class and you're done. You can use the `@Deployment.order` if they need to be deployed in a specific order. When dealing with multiple in-container deployments you need to specify which `Deployment` context the individual test methods should run in. You do this by adding a name to the deployment by using the `@Deployment.name` and refer to that name on the test method by adding `@OperateOnDeployment("deploymentName")`.

```
@Deployment(name = "dep1", order = 1)
public static WebArchive createDep1() {}

@Deployment(name = "dep2", order = 2)
public static WebArchive createDep2() {}

@Test @OperateOnDeployment("dep1")
public void testRunningInDep1() {}

@Test @OperateOnDeployment("dep2")
public void testRunningInDep2() {}
```

9.4. Multiple Containers

There are times when you need to involve multiple containers in the same test case, if you for instance want to test clustering. The first step you need to take is to add a `group` with multiple containers to your Arquillian configuration.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<arquillian xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xsi:schemaLocation="http://jboss.org/schema/arquillian http://jboss.org/schema/
arquillian/arquillian_1_0.xsd">
  <group qualifier="tomcat-cluster">
    <container qualifier="container-1" default="true">
      <configuration>
```

```

    <property name="tomcatHome">target/tomcat-embedded-6-standby</property>
    <property name="workDir">work</property>
    <property name="bindHttpPort">8880</property>
    <property name="unpackArchive">true</property>
  </configuration>
  <dependencies>
    <dependency>org.jboss.arquillian.container:arquillian-tomcat-embedded-6:1.0.0-
    SNAPSHOT</dependency>
    <dependency>org.apache.tomcat:catalina:6.0.29</dependency>
    <dependency>org.apache.tomcat:coyote:6.0.29</dependency>
    <dependency>org.apache.tomcat:jasper:6.0.29</dependency>
  </dependencies>
</container>
<container qualifier="container-2">
  <configuration>
    <property name="tomcatHome">target/tomcat-embedded-6-active-1</property>
    <property name="workDir">work</property>
    <property name="bindHttpPort">8881</property>
    <property name="unpackArchive">true</property>
  </configuration>
  <dependencies>
    <dependency>org.jboss.arquillian.container:arquillian-tomcat-embedded-6:1.0.0-
    SNAPSHOT</dependency>
    <dependency>org.apache.tomcat:catalina:6.0.29</dependency>
    <dependency>org.apache.tomcat:coyote:6.0.29</dependency>
    <dependency>org.apache.tomcat:jasper:6.0.29</dependency>
  </dependencies>
</container>
</group>
</arquillian>

```

So what we have done here is to say we have two containers that Arquillian will control, container-1 and container-2. Arquillian will now instead of starting up one container, which is normal, start up two. In your test class you can target different deployments against the different containers using the `@TargetsContainer("containerName")` annotation on your `Deployment` methods.

```

@Deployment(name = "dep1") @TargetsContainer("container-1")
public static WebArchive createDep1() {}

@Deployment(name = "dep2") @TargetsContainer("container-2")
public static WebArchive createDep2() {}

@Test @OperateOnDeployment("dep1")

```

```
public void testRunningInDep1() {}

@Test @OperateOnDeployment("dep2")
public void testRunningInDep2() {}
```

We now have a single test class that will be executed in two different containers. `testRunningInDep1` will operate in the context of the `dep1` deployment which is deployed on the container named `container-1` and `testRunningInDep2` will operate in the context of deployment `dep2` which is deployed on container `container-2`. As the test moves along, each method is executed inside the individual containers.



Note

We also define the containers dependencies as part of the Arquillian xml. In some cases, like when running against multiple containers of the same type and the container has no client side state, this might not be needed. But for the sake of the example we define them in the configuration. In this case you should not have any of these dependencies on your application classpath.



Warning

Defining dependencies in arquillian xml is at the moment considered a experimental feature.

9.5. Protocol selection

A protocol is how Arquillian talks and executes the tests inside the container. For ease of development and configuration a container defines a default protocol that will be used if no other is specified. You can override this default behavior by defining the `@OverProtocol` annotation on your `@Deployment` method.

```
@Deployment @OverProtocol("MyCustomProtocol")
public static WebArchive createDep1() {}

@Test
public void testExecutedUsingCustomProtocol() {}
```

When `testExecutedUsingCustomProtocol` is executed, instead of using the containers default defined protocol, Arquillian will use `MyCustomProtocol` to communicate with the container. Since this is defined on `Deployment` level, you can have different test methods operate on different deployments and there for be executed using different protocols. This can be useful when for

instance a protocols packaging requirements hinder how you define your archive, or you simply can't communicate with the container using the default protocol due to e.g. fire wall settings.

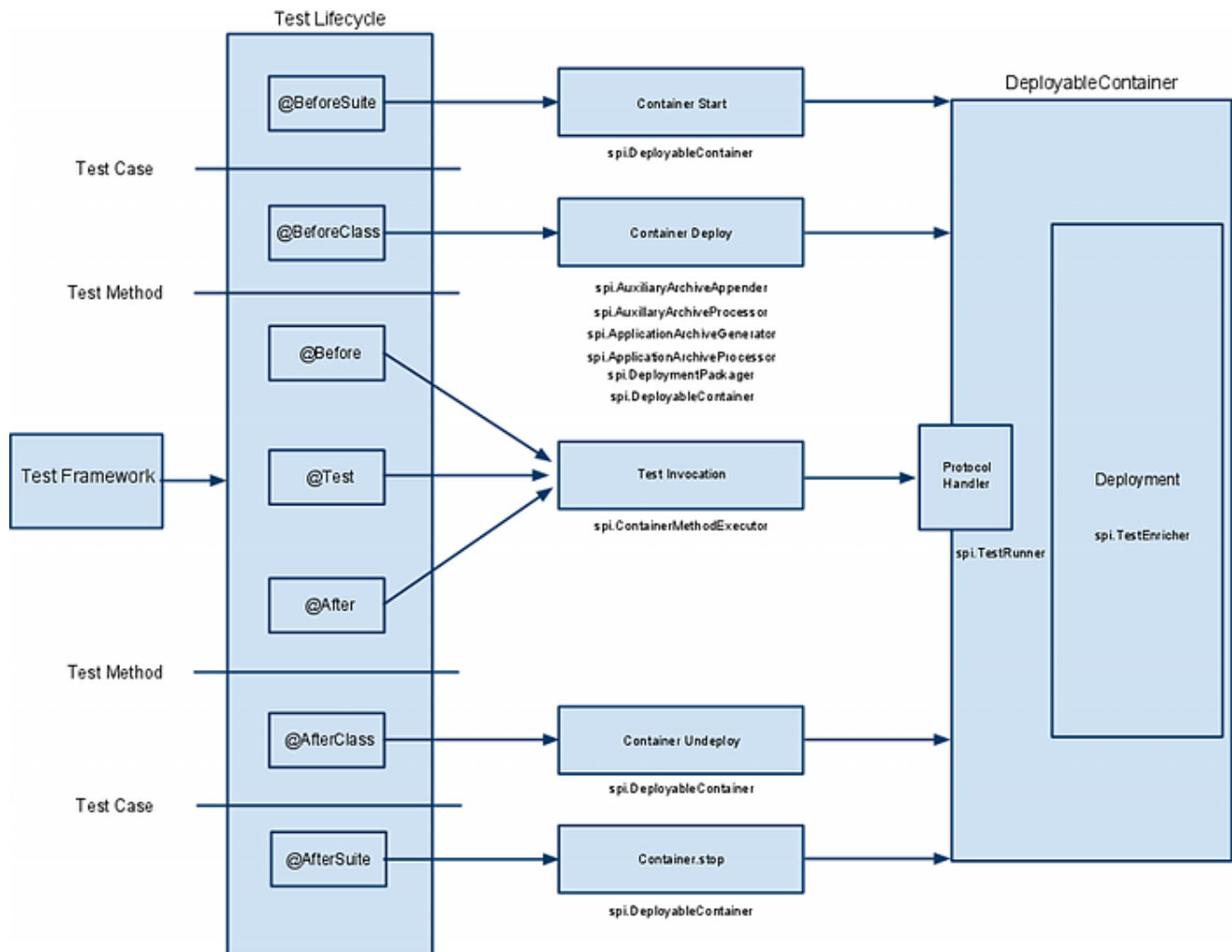


Warning

Arquillian only support Servlet 2.5 and Servlet 3.0 at this time. EJB 3.0 and 3.1 are in the plans. But your might implement your own Protocol. See the *[Complete Protocol Reference](#)* for what is currently supported.

Extending Arquillian

Arquillian is designed to be very extensible. This is accomplished through the use of Service Provider Interfaces (SPIs). The following diagram shows how the various SPIs in Arquillian tie into the test execution.



Arquillian test execution and SPI overview

Complete Extension/Framework Reference

11.1. Performance

The performance extension to Arquillian is a simple way of checking that the code you want to test performs within the range you want it to. It's can also automatically catch any performance regressions that might be added to your applications. - and as Arquillian itself, its very easy to use.

11.1.1. Code example

```
// include other arquillian imports here...
import org.jboss.arquillian.performance.annotation.Performance;
import org.jboss.arquillian.performance.annotation.PerformanceTest;

@PerformanceTest(resultsThreshold=2)
@RunWith(Arquillian.class)
public class WorkHardCdiTestCase
{
    @Deployment
    public static JavaArchive createDeployment() {
        return ShrinkWrap.create(JavaArchive.class, "test.jar")
            .addPackage( WorkHard.class.getPackage())
            .addAsManifestResource(
                EmptyAsset.INSTANCE,
                ArchivePaths.create("beans.xml"));
    }

    @Inject HardWorker worker;

    @Test
    @Performance(time=20)
    public void doHardWork() throws Exception
    {
        Assert.assertEquals(21, worker.workingHard(), 0d);
    }
}
```

As you can see the only two additions needed are `@Performance` and `@PerformanceTest`. They do different things and can be used separately or combined.

`@Performance` require one argument, `time` (a double) which set the required maximum time that the test is allowed to spend in milliseconds. If the test exceeds that time it will fail with an exception explaining the cause.

`@PerformanceTest` will cause every testrun of that test to be saved and every new run will compare results with previous runs. If the new testrun exceeds the previous runs with a defined threshold an exception will be thrown. The threshold can be set with the parameter `resultsThreshold`. It is by default set to 1d.

How threshold is calculated: `resultsThreshold * newTime < oldTime`.

11.1.2. Maven setup example

The only extra dependency needed is to add `arquillian-performance` to your `pom.xml`. Take a look at the [Chapter 3, Getting started](#) to see how you set up arquillian using maven.

```
<dependency>
  <groupId>org.jboss.arquillian.extension</groupId>
  <artifactId>arquillian-performance</artifactId>
  <version>${arquillian.version}</version>
  <scope>test</scope>
</dependency>
```

11.2. JSFUnit

The JSFUnit integration to Arquillian is a simpler way of using JSFUnit.

- You no longer need to manually post processor your WebArchives with JSFUnit dependencies
- You can easily test single pages
- Both in-container and client mode support
- Use JUnit 4.8.1 or TestNG 5.12.1



Warning

JSFUnit integration requires a Java EE 6 compliant server. The packaging is based on web-fragments from Servlet 3.0.

11.2.1. Code example

```
// imports here...
@RunWith(Arquillian.class)
public class JSFUnitTestCase
{
    @Deployment
    public static WebArchive createDeployment()
    {
        return ShrinkWrap.create(WebArchive.class, "test.war")
            .addClasses(
                RequestScopeBean.class,
                ScopeAwareBean.class)
            .setWebXML("jsf/jsf-web.xml")
            .addResource("jsf/index.xhtml", "index.xhtml")
            .addWebResource(EmptyAsset.INSTANCE, ArchivePaths.create("beans.xml"));
    }

    @Test
    public void shouldExecutePage() throws Exception
    {
        JSFSession jsfSession = new JSFSession("/index.jsf");

        Assert.assertTrue(Environment.is12Compatible());
        Assert.assertTrue(Environment.is20Compatible());
        Assert.assertEquals(2, Environment.getJSFMajorVersion());
        Assert.assertEquals(0, Environment.getJSFMinorVersion());

        JSFServerSession server = jsfSession.getJSFServerSession();

        Assert.assertEquals("request", server.getManagedBeanValue("#{requestBean.scope}"));
    }
}
```

11.2.2. Maven setup example

The only dependencies needed is to add `org.jboss.arquillian.framework:arquillian-framework-jsfunit` and `org.jboss.jsfunit:jboss-jsfunit-core` to your `pom.xml`. The rest is handled by Arquillian in the background. Take a look at the [Chapter 3, Getting started](#) to see how you set up arquillian using maven.

```
<dependency>
  <groupId>org.jboss.arquillian.framework</groupId>
  <artifactId>arquillian-framework-jsfunit</artifactId>
  <version>1.0.0-SNAPSHOT</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.jboss.jsfunit</groupId>
  <artifactId>jboss-jsfunit-core</artifactId>
  <version>1.3.0.Final</version>
  <scope>test</scope>
</dependency>
```



Warning

To use JSFUnit with Arquillian, JSFUnit 1.3.0.Final is required.

11.3. Drone

The Arquillian Drone extension for Arquillian provides a simple way of including functional tests for your web based application. Arquillian Drone manages the life cycle of web testing tool, which is either Arquillian Ajocado, Selenium or WebDriver. Arquillian Drone automatically manages life cycle of objects required for interaction between browser and deployed application.

11.3.1. Commented Example

Following example illustrates how Arquillian Drone can be used with Arquillian Ajocado. This example is a part of Arquillian Drone test classes, so you are free to experiment with it. Arquillian Ajocado is a Selenium on steroids, because it provides type safe API over classic `DefaultSelenium` object, has extended support for handling AJAX based UI and adds pretty fast JQuery locators to you browser, so your test are executed faster. If you are not experienced with Arquillian Ajocado, you can still use `DefaultSelenium` or `WebDriver` specific browser, such as `FirefoxDriver`. The beauty of Arquillian Drone is that it supports all of them and their usage is pretty much the same.

```
package org.jboss.arquillian.drone.example;

import static org.jboss.arquillian.ajocado.Ajocado.elementPresent;
import static org.jboss.arquillian.ajocado.Ajocado.waitModel;
import static org.jboss.arquillian.ajocado.guard.request.RequestTypeGuardFactory.waitHttp;
```

```

import static org.jboss.arquillian.ajocado.locator.LocatorFactory.id;
import static org.jboss.arquillian.ajocado.locator.LocatorFactory.xp;

import java.net.URL;

import org.jboss.arquillian.ajocado.framework.AjaxSelenium;
import org.jboss.arquillian.ajocado.locator.IdLocator;
import org.jboss.arquillian.ajocado.locator.XpathLocator;
import org.jboss.arquillian.api.Run;
import org.jboss.arquillian.drone.annotation.ContextPath;
import org.jboss.arquillian.drone.annotation.Drone;
import org.jboss.arquillian.junit.Arquillian;
import org.junit.Assert;
import org.junit.Test;
import org.junit.runner.RunWith;

/**
 * Tests Arquillian Drone extension against Weld Login example. *
 * Uses Ajocado driver bound to Firefox browser.
 */
@RunWith(Arquillian.class)
public class AjocadoTestCase extends AbstractTestCase
{
    // load ajocado driver
    @Drone
    AjaxSelenium driver;

    // Load context path to the test
    @ContextPath
    URL contextPath;

    protected XpathLocator LOGGED_IN = xp("//li[contains(text(),'Welcome')]");
    protected XpathLocator LOGGED_OUT = xp("//li[contains(text(),'Goodbye')]");

    protected IdLocator USERNAME_FIELD = id("loginForm:username");
    protected IdLocator PASSWORD_FIELD = id("loginForm:password");

    protected IdLocator LOGIN_BUTTON = id("loginForm:login");
    protected IdLocator LOGOUT_BUTTON = id("loginForm:logout");

    @Deployment(testable=false)
    public static WebArchive createDeployment()
    {

```

```

    return ShrinkWrap.create(WebArchive.class, "weld-login.war")
        .addClasses(Credentials.class, LoggedIn.class, Login.class, User.class, Users.class)
        .addAsWebInfResource(new File("src/test/webapp/WEB-INF/beans.xml"))
        .addAsWebInfResource(new File("src/test/webapp/WEB-INF/faces-config.xml"))
        .addAsWebInfResource(new File("src/test/resources/import.sql"))
        .addAsWebResource(new File("src/test/webapp/index.html"))
        .addAsWebResource(new File("src/test/webapp/home.xhtml"))
        .addAsWebResource(new File("src/test/webapp/template.xhtml"))
        .addAsWebResource(new File("src/test/webapp/users.xhtml"))
        .addAsResource(new File("src/test/resources/META-INF/
persistence.xml"), ArchivePaths.create("META-INF/persistence.xml"))
        .setWebXML(new File("src/test/webapp/WEB-INF/web.xml"));
}

@Test
public void testLoginAndLogout()
{
    driver.open(contextPath);
    waitModel.until(elementPresent.locator(USERNAME_FIELD));
    Assert.assertFalse("User should not be logged
in!", driver.isElementPresent(LOGOUT_BUTTON));
    driver.type(USERNAME_FIELD, "demo");
    driver.type(PASSWORD_FIELD, "demo");

    waitHttp(driver).click(LOGIN_BUTTON);
    Assert.assertTrue("User should be logged in!", driver.isElementPresent(LOGGED_IN));

    waitHttp(driver).click(LOGOUT_BUTTON);
    Assert.assertTrue("User should not be logged in!", driver.isElementPresent(LOGGED_OUT));
}
}

```

As you can see, execution does not differ from common Arquillian test much. The only requirement is actually running Arquillian in client mode, which is enforced by marking deployment as `utestable = false` or alternatively by `@RunAsClient` annotation. The other annotations present in the test are used to inject web test framework instance (`@Drone`) and context path (`@ContextPath`) for deployed archive into your test. Their life cycle is completely managed by Arquillian Drone, as described in [Section 11.3.3, “Life cycle of @Drone objects”](#). The instance is used in test method to traverse UI of application via Firefox browser, fill user credentials and signing up and out. Test is based on JUnit, but Arquillian Drone, as well as the rest of Arquillian supports TestNG as well.

Table 11.1. Supported frameworks and their tested versions

Framework name and implementation class	Tested version	Additional information
Arquillian Ajocado - AjaxSelenium	1.0.0.Alpha1	Requires Selenium Server running
Selenium - DefaultSelenium	2.0b2	Requires Selenium Server running
Selenium - HtmlUnitDriver, FirefoxDriver	2.0b2	Selenium Server is not required

This combination matrix is tested and known to work. However, we expect that all `WebDriver` interface based browsers will work. Arquillian Drone does not force you to use a specific version of web framework test implementation, so feel free to experiment with it.

11.3.2. Maven setup example

Arquillian Drone requires a few test dependencies which are marked as provided to let you choose their versions. Add following code into your Maven dependencies to enable Arquillian Drone functionality in your test cases.

```

<!-- Arquillian Drone dependency -->
<dependency>
  <groupId>org.jboss.arquillian.extension</groupId>
  <artifactId>arquillian-drone</artifactId>
  <version>${arquillian.version}</version>
  <scope>test</scope>
</dependency>

<!-- Arquillian Ajocado dependencies -->
<dependency>
  <groupId>org.jboss.arquillian.ajocado</groupId>
  <artifactId>arquillian-ajocado-api</artifactId>
  <version>${version.ajocado}</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.jboss.arquillian.ajocado</groupId>
  <artifactId>arquillian-ajocado-impl</artifactId>
  <version>${version.ajocado}</version>
  <scope>test</scope>
</dependency>

<!-- Selenium (including WebDriver in 2.x versions) -->

```

```
<dependency>
  <groupId>org.seleniumhq.selenium</groupId>
  <artifactId>selenium-remote-control</artifactId>
  <version>${version.selenium}</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.seleniumhq.selenium</groupId>
  <artifactId>selenium-server</artifactId>
  <version>${version.selenium}</version>
  <scope>test</scope>
</dependency>

<!-- required to run Selenium Server, needed if you want Arquillian Drone to start Selenium Server
for you -->
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-simple</artifactId> <!-- choose different underlying implementation if you want --
>
  <version>${version.slf4j}</version> <!-- up to you, tested with 1.5.10 -->
  <scope>test</scope>
</dependency>
```

11.3.3. Life cycle of @Drone objects

Arquillian Drone does not allow you to control life cycle of web testing framework objects, but it provides two different scenarios which should be sufficient for most usages required by developers. These are

1. Class based life cycle
2. Method based life cycle

For class based life cycle, configuration for the instance is created before a test class is run. This configuration is used to properly initialize an instance of the tool. The instance is injected into the field and hold until the last test in the test class is finished, then it is disposed. You can think of `@BeforeClass` and `@AfterClass` equivalents. On the other hand, for method based life cycle, an instance is configured and created before Arquillian enters test method and it is disposed after method finishes. You can think of `@Before` and `@After` equivalents.

It is import to know that you can combines multiple instances in one tests and you can have them in different scopes. You can as well combine different framework types. Following example shows class based life cycle instance `foo` of type `AjaxSelenium` (Arquillian Ajocado) combined with method based life cycle `bar` of type `DefaultSelenium` (Selenium).


```

@RunWith(Arquillian.class)
@RunAs(AS_CLIENT)
public class EnrichedClass
{
    @Drone AjaxSelenium foo;

    public void testRatherWithSelenium(@Drone DefaultSelenium bar)
    {
        ...
    }
}

```

11.3.4. Keeping multiple @Drone instances of the same type

With Arquillian Drone, it is possible to keep more than one instance of a web test framework tool of the same type and determine which instance to use in a type safe way. Arquillian Drone uses concept of `@Qualifier`, which may be known to you from CDI. `@Qualifier` is a meta-annotations which allows you to annotate annotation you create to tell instances apart. By default, if no `@Qualifier` annotation is present, Arquillian Drone uses `@Default`. Following code defines new qualifying annotation

```

package org.jboss.arquillian.drone.factory;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

import org.jboss.arquillian.drone.spi.Qualifier;

@Retention(RetentionPolicy.RUNTIME)
@Target({ ElementType.FIELD, ElementType.PARAMETER })
@Qualifier
public @interface Different
{
}

```

Once you have defined a qualifier, you can use it in you tests, for example in following way, having two distinct class based life cycle instances of `DefaultSelenium`.

```
@RunWith(Arquillian.class)
@RunWith(AS_CLIENT)
public class EnrichedClass
{
    @Drone DefaultSelenium foo;
    @Drone @Different DefaultSelenium bar;

    public void testWithBothFooAndBar()
    {
        ...
    }
}
```

11.3.5. Configuring @Drone instances

@Drone instances are automatically configured from arquillian.xml descriptor file, with possibility of overriding arquillian.xml configuration by system properties. Every type of @Drone instance has its own configuration namespace, although namespace overlap in areas where it makes sense, such as sharing a part of configuration between Selenium and Selenium server. System properties always take precedence.

If you are using @Qualifiers, you can use them to modify configuration, such as create a special configuration for a method based life cycle browser.

Table 11.2. Arquillian Ajocado configuration

Property name	Default value	Description
contextRoot	http://localhost:8080	Web Server URL
contextPath	(empty)	Application URL of your deployed application
browser	*firefox	Browser type, following Selenium conventions
resourcesDirectory	target/test-classes	Directory where additional resources are stored
buildDirectory	target	Directory where application is built
seleniumHost	localhost	Name of the machine where Selenium server is running
seleniumPort	14444	Port on machine where Selenium server is running
seleniumMaximize	false	Maximize browser window

Property name	Default value	Description
seleniumDebug	false	Produce debug output in browser console
seleniumNetworkTrafficEnabled	false	Capture network traffic in browser console
seleniumSpeed	0	Delay in ms before each command is sent
seleniumTimeoutDefault	30000	Default timeout in ms
seleniumTimeoutGui	5000	Timeout of GUI wait in ms
seleniumTimeoutAjax	15000	Timeout of AJAX wait in ms
seleniumTimeoutModel	30000	Timeout of Model wait in ms

Arquillian Ajocado uses `ajocado` namespace. This means, you can define properties either in `arquillian.xml`

```
<extension qualifier="ajocado">
  <configuration>
    <property name="seleniumHost">myhost.org</property>
  </configuration>
</extension>
```

Or you can convert property name to name of system property, using following formula `arquillian.` + `(namespace)` + `.` + `(property name converted to dotted lowercase)`. For instance, `seleniumNetworkTrafficEnabled` will be converted to `arquillian.ajocado.selenium.network.traffic.enabled` System property name.

Table 11.3. Selenium configuration

Property name	Default value	Description
serverPort	14444	Port on machine where Selenium server is running
serverHost	localhost	Name of the machine where Selenium server is running
url	http://localhost:8080	Web Server URL
timeout	60000	Default timeout in ms
speed	0	Delay in ms before each command is sent
browser	*firefox	Browser type, following Selenium conventions

Selenium uses `selenium` namespace.

Table 11.4. WebDriver configuration

Property name	Default value	Description
implementationClass	org.openqa.selenium.htmlunit.HtmlUnitDriver	Determines which browser instance is created for WebDriver testing

WebDriver uses `webdriver` namespace.

Table 11.5. Selenium Server configuration

Property name	Default value	Description
port	14444	Port on machine where to start Selenium Server
host	localhost	Name of the machine where to start Selenium Server
output	target/selenium-server-output.log	Name of file where to redirect Selenium Server logger
enable	false	Enable Arquillian to start Selenium Server

Selenium Server uses `selenium-server` namespace.



Warning

Please note that non-letter characters are converted to dots, so for instance to enable Selenium via System property, you have to set `arquillian.selenium.server.enable` to `true`.

Selenium Server has different life cycle than `@Drone` instance, it is created and started before test suite and disposed after test suite. If you have your own Selenium Server instance running, you simply omit its configuration, however specifying it is the simplest way how to start it and have it managed by Arquillian.

If you are wondering how to define configuration for `@Qualifier @Drone` instance, it's very easy. Only modification you have to do is to change namespace to include - (`@Qualifier` annotation name converted to lowercase). Please note, that for System properties are all non-letter characters converted to dots. For instance, if you qualified Arquillian Ajocado instance with `@MyExtraBrowser`, its namespace will become `ajocado-myextrabrowser`.

The namespace resolution is a bit more complex. Arquillian Drone will search for configuration in following order:

1. Search for the exact match of namespace (e.g. `ajocado-myextrabrowser`) in `arquillian.xml`, if found, step 2 is not performed

2. Search for a match of base namespace, without qualifier (e.g. `ajocado`) in `arquillian.xml`

Then System property overrides are applied in the same fashion.

11.3.6. Arquillian Drone SPI

The big advantage of Arquillian Drone extension is its flexibility. We provide you reasonable defaults, but if they are not sufficient or if they do not fulfill your needs, you can change them. You can change behaviour of existing implementation or implement support for your own testing framework. See `JavaDoc/sources` for more details, here is an enumeration of classes you should focus on:

`org.jboss.arquillian.drone.spi.Configurator<T, C>`

Provides a way how to configure configurations of type `C` for `@Drone` object of type `T`

`org.jboss.arquillian.drone.spi.Instantiator<T, C>`

Provides a way how to instantiate `@Drone` object of type `T` with configuration `C`

`org.jboss.arquillian.drone.spi.Destructor<T>`

Provides a way how to dispose `@Drone` object of type `T`

`org.jboss.arquillian.drone.spi.DroneConfiguration`

This is effectively a marker for configuration of type `C`

The import note is that implementation of `Configurator`, `Instantiator` and `Destructor` are searched on the class path and they are sorted according to precedence they declare. Arquillian Ajocado default implementation has precedence of 0, so if your implementation has bigger precedence and instantiates type `T` with configuration `C`, Arquillian Drone will use it. This provides you the ultimate way how to change behavior if desired. Of course, you can provide support for your own framework in the very same way, so in your test you can use `@Drone` annotation to manage instance of arbitrary web testing framework.

Arquillian Drone SPI extensions are searched via descriptions in `META-INF/services` on class path. For instance, to override `DefaultSelenium` instantiator, create file `META-INF/services/org.jboss.arquillian.drone.spi.Instantiator` with following content:

```
fully.qualified.name.of.my.implementation.Foo
```

Your class `Foo` must implement `Instantiator<DefaultSelenium, SeleniumConfiguration>` interface.

Complete Container Reference

12.1. JBoss AS 5 - Remote

A DeployableContainer implementation that can connect and run against a remote(different JVM, different machine) running JBoss AS 5 instance. This implementation has no lifecycle support, so it can not be started or stopped.



Warning

This container needs a jndi.properties file on classpath to be able to connect to the remote running instance.

Table 12.1. Container Injection Support Matrix

@EJB	@EJB interface)	(no- @Inject (CDI)	@Inject (MC)	@PersistenceContext @PersistenceUnit

12.1.1. Configuration

Default Protocol: [Servlet 2.5](#)

Table 12.2. Container Configuration Options

Name	Type	Default	Description
profileName	String	default	ProfileService profileKey. Used to load the correct profile into the DeploymentManager.
providerUrl	String	jnp://localhost:1099	The JNDI connection URL.
urlPkgPrefix	String	org.jboss.naming:org.jnp.interfaces	The JNDI package prefix.
contextFactory	String	org.jnp.interfaces.NamingContextFactory	The JNDI context factory class name.

Example of Maven profile setup

```
<profile>
  <id>jbossas-remote-5</id>
  <dependencies>
    <dependency>
      <groupId>org.jboss.arquillian.container</groupId>
      <artifactId>arquillian-jbossas-remote-5</artifactId>
```

```

    <version>1.0.0-SNAPSHOT</version>
  </dependency>
  <dependency>
    <groupId>org.jboss.jbossas</groupId>
    <artifactId>jboss-as-client</artifactId>
    <version>5.0.1.GA</version>
    <type>pom</type>
  </dependency>
</dependencies>
</profile>

```

12.2. JBoss AS 5.1 - Remote


A DeployableContainer implementation that can connect and run against a remote(different JVM, different machine) running JBoss AS 5.1 instance. This implementation has no lifecycle support, so it can not be started or stopped.



Warning

This container needs a jndi.properties file on classpath to be able to connect to the remote running instance.

Table 12.3. Container Injection Support Matrix

@EJB	@EJB (no- interface)	@Inject (CDI)	@Inject (MC)	@PersistenceContext @PersistenceUnit
				

12.2.1. Configuration

Default Protocol: [Servlet 2.5](#)

Table 12.4. Container Configuration Options

Name	Type	Default	Description
profileName	String	default	ProfileService profileKey. Used to load the correct profile into the DeploymentManager.
providerUrl	String	jnp://localhost:1099	The JNDI connection URL.
urlPkgPrefix	String	org.jboss.naming:org.jnp.interfaces	The JNDI package prefix.
contextFactory	String	org.jnp.interfaces.NamingContextFactory	The JNDI context factory class name.

Example of Maven profile setup

```


<profile>
  <id>jbossas-remote-5.1</id>
  <dependencies>
    <dependency>
      <groupId>org.jboss.arquillian.container</groupId>
      <artifactId>arquillian-jbossas-remote-5.1</artifactId>
      <version>1.0.0-SNAPSHOT</version>
    </dependency>
    <dependency>
      <groupId>org.jboss.jbossas</groupId>
      <artifactId>jboss-as-client</artifactId>
      <version>5.1.0.GA</version>
      <type>pom</type>
    </dependency>
  </dependencies>
</profile>

```

12.3. JBoss AS 5.1 - Managed

A DeployableContainer implementation that can run and connect to a remote(different JVM, same machine) JBoss AS 5.1 instance. This implementation has lifecycle support, so the container will be started and stopped as part of the test run.

Table 12.5. Container Injection Support Matrix

@EJB	@EJB (no- interface)	@Inject (CDI)	@Inject (MC)	@PersistenceContext @PersistenceUnit
				

12.3.1. Configuration

Default Protocol: [Servlet 2.5](#)

Table 12.6. Container Configuration Options

Name	Type	Default	Description
profileName	String	default	ProfileService profileKey. Used to load the correct profile into the DeploymentManager.
bindAddress	String	localhost	The Address the server should bind to.

Name	Type	Default	Description
httpPort	int	8080	Used by the ServerManager to communicate with the server.
rmiPort	int	1099	Used by the ServerManager to communicate with the server.
jbossHome	String	\$JBoss_HOME	The JBoss configuration to start.
javaHome	String	\$JAVA_HOME	The Java runtime to use to start the server.
javaVmArguments	String	-Xmx512m XX:MaxPermSize=128m	JVM arguments used to start the server.
useRmiPortForAliveCheck	boolean	false	If the ServerManager should use the RMI port when checking if the server is up.
startupTimeoutInSeconds	int	120	Time to wait before throwing Exception on server startup.
shutdownTimeoutInSeconds	int	45	Time to wait before throwing Exception on server shutdown.

Example of Maven profile setup

```

<profile>
  <id>jbossas-managed-5.1</id>
  <dependencies>
    <dependency>
      <groupId>org.jboss.arquillian.container</groupId>
      <artifactId>arquillian-jbossas-managed-5.1</artifactId>
      <version>1.0.0-SNAPSHOT</version>
    </dependency>
    <dependency>
      <groupId>org.jboss.jbossas</groupId>
      <artifactId>jboss-server-manager</artifactId>
      <version>1.0.3.GA</version>
    </dependency>
    <dependency>
      <groupId>org.jboss.jbossas</groupId>
      <artifactId>jboss-as-client</artifactId>
      <version>5.1.0.GA</version>
      <type>pom</type>
    </dependency>
  </dependencies>
</profile>

```

12.4. JBoss AS 6.0 - Remote

A DeployableContainer implementation that can connect and run against a remote(different JVM, different machine) running JBoss AS 6.0 instance. This implementation has no lifecycle support, so it can not be started or stopped.



Warning

This container needs a jndi.properties file on classpath to be able to connect to the remote running instance.

Table 12.7. Container Injection Support Matrix

@EJB	@EJB (no- interface)	@Inject (CDI)	@Inject (MC)	@PersistenceContext @PersistenceUnit

12.4.1. Configuration

Default Protocol: [Servlet 3.0](#)

Table 12.8. Container Configuration Options

Name	Type	Default	Description
profileName	String	default	ProfileService profileKey. Used to load the correct profile into the DeploymentManager.
providerUrl	String	jnp://localhost:1099	The JNDI connection URL.
urlPkgPrefix	String	org.jboss.naming:org.jnp.interfaces	The JNDI package prefix.
contextFactory	String	org.jnp.interfaces.NamingContextFactory	JNDI Context factory class name.

Example of Maven profile setup

```
<profile>
  <id>jbossas-remote-6</id>
  <dependencies>
    <dependency>
      <groupId>org.jboss.arquillian.container</groupId>
      <artifactId>arquillian-jbossas-remote-6</artifactId>
      <version>1.0.0-SNAPSHOT</version>
    </dependency>
    <dependency>
```

```




<groupId>org.jboss.jbossas</groupId>
<artifactId>jboss-as-client</artifactId>
<version>6.0.0.Final</version>
<type>pom</type>
</dependency>
</dependencies>
</profile>

```

12.5. JBoss AS 6.0 - Managed

A DeployableContainer implementation that can run and connect to a remote(different JVM, same machine) JBoss AS 6.0 instance. This implementation has lifecycle support, so the container will be started and stopped as part of the test run.

Table 12.9. Container Injection Support Matrix

@EJB	@EJB interface)	(no-	@Inject (CDI)	@Inject (MC)	@PersistenceContext @PersistenceUnit
					

12.5.1. Configuration

Default Protocol: [Servlet 3.0](#)

Table 12.10. Container Configuration Options

Name	Type	Default	Description
profileName	String	default	ProfileService profileKey. Used to load the correct profile into the DeploymentManager.
bindAddress	String	localhost	The Address the server should bind to.
httpPort	int	8080	Used by Servlet Protocol to connect to the server.
rmiPort	int	1099	Used by the ServerManager to communicate with the server.
jbossHome	String	\$JBOSS_HOME	The JBoss configuration to start.
javaHome	String	\$JAVA_HOME	The Java runtime to use to start the server.
javaVmArguments	String	-Xmx512m -XX:MaxPermSize=128m	JVM arguments used to start the server.

Name	Type	Default	Description
useRmiPortForAliveCheck	boolean	false	If the ServerManager should use the RMI port when checking if the server is up.
startupTimeoutInSeconds	int	120	Time to wait before throwing Exception on server startup.
shutdownTimeoutInSeconds	int	45	Time to wait before throwing Exception on server shutdown.




Example of Maven profile setup

```
<profile>
  <id>jbossas-managed-6</id>
  <dependencies>
    <dependency>
      <groupId>org.jboss.arquillian.container</groupId>
      <artifactId>arquillian-jbossas-managed-6</artifactId>
      <version>1.0.0-SNAPSHOT</version>
    </dependency>
    <dependency>
      <groupId>org.jboss.jbossas</groupId>
      <artifactId>jboss-server-manager</artifactId>
      <version>1.0.3.GA</version>
    </dependency>
    <dependency>
      <groupId>org.jboss.jbossas</groupId>
      <artifactId>jboss-as-client</artifactId>
      <version>6.0.0.Final</version>
      <type>pom</type>
    </dependency>
  </dependencies>
</profile>
```

12.6. JBoss AS 6.0 - Embedded

A DeployableContainer implementation that can run and connect to a embedded(same JVM) JBoss AS 6.0 instance. This implementation has lifecycle support, so the container will be started and stopped as part of the test run.

Table 12.11. Container Injection Support Matrix

@EJB	@EJB (no- interface)	@Inject (CDI)	@Inject (MC)	@PersistenceContext @PersistenceUnit
				

12.6.1. Configuration

Default Protocol: [Servlet 3.0](#)

Table 12.12. Container Configuration Options

Name	Type	Default	Description
profileName	String	default	ProfileService profileKey. Used to load the correct profile into the DeploymentManager.
bindAddress	String	localhost	The Address the server should bind to.

Example of Maven profile setup

```
<profile>
  <id>jbossas-embedded-6</id>
  <dependencies>
    <dependency>
      <groupId>org.jboss.arquillian.container</groupId>
      <artifactId>arquillian-jbossas-embedded-6</artifactId>
      <version>1.0.0-SNAPSHOT</version>
    </dependency>
    <dependency>
      <groupId>org.jboss.jbossas</groupId>
      <artifactId>jboss-as-depchain</artifactId>
      <version>6.0.0.Final</version>
      <type>pom</type>
    </dependency>
  </dependencies>
  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>org.jboss.jbossas</groupId>
        <artifactId>jboss-as-depchain</artifactId>
        <version>6.0.0.Final</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>
</profile>
```

```

</dependencyManagement>
<build>
  <plugins>
    <plugin>
      <artifactId>maven-surefire-plugin</artifactId>
      <configuration>
        <additionalClasspathElements>
          <additionalClasspathElement>${env.JBOSS_HOME}/client/jboss-
client.jar</additionalClasspathElement>
          <!--
            Because jbossweb.sar contains shared web.xml, which must be
            visible from same CL as TomcatDeployer.class.getClassLoader
          -->
          <additionalClasspathElement>${env.JBOSS_HOME}/server/default/deploy/
jbossweb.sar</additionalClasspathElement>
        </additionalClasspathElements>

        <redirectTestOutputToFile>true</redirectTestOutputToFile>
        <trimStackTrace>false</trimStackTrace>
        <printSummary>true</printSummary>
        <forkMode>once</forkMode>


        <!--
          MaxPermSize Required to bump the space for selective data like
          classes, methods, etc. EMB-41. Endorsed required for things like
          WS support (EMB-61)
        -->
        <argLine>-Xmx512m -XX:MaxPermSize=256m -Djava.net.preferIPv4Stack=true -
Djava.util.logging.manager=org.jboss.logmanager.LogManager
Djava.endorsed.dirs=${env.JBOSS_HOME}/lib/endorsed
Djboss.home=${env.JBOSS_HOME} -Djboss.boot.server.log.dir=${env.JBOSS_HOME}</
argLine>
      </configuration>
    </plugin>
  </plugins>
</build>
</profile>

```

12.7. JBoss Reloaded 1.0 - Embedded

A DeployableContainer implementation that can run and connect to a embedded(same JVM) JBoss Reloaded(MicroContainer + VirtualDeploymentFramework) instance. This implementation has lifecycle support, so the container will be started and stopped as part of the test run.

Table 12.13. Container Injection Support Matrix

@EJB	@EJB (no- interface)	@Inject (CDI)	@Inject (MC)	@PersistenceContext @PersistenceUnit
				

12.7.1. Configuration

Default Protocol:




Table 12.14. Container Configuration Options

Name	Type	Default	Description

12.8. GlassFish 3.1 - Embedded

A DeployableContainer implementation that can run and connect to a embedded(same JVM) GlassFish 3.1 instance. This implementation has lifecycle support, so the container will be started and stopped as part of the test run.

Table 12.15. Container Injection Support Matrix

@EJB	@EJB (no- interface)	@Inject (CDI)	@Inject (MC)	@PersistenceContext @PersistenceUnit
				

12.8.1. Configuration

Default Protocol: [Servlet 3.0](#)

Table 12.16. Container Configuration Options

Name	Type	Default	Description
bindHttpPort	int	8181	The HTTP port the server should bind to.
instanceRoot	String		The instanceRoot to use for booting the server. If it does not exist, a default structure will be created.
installRoot	String		The installRoot to use for booting the server. If it does not exist, a default structure will be created.
configurationXml	String		The relative or absolute path to the domain.xml file that will be used to

Name	Type	Default	Description
			configure the instance. If absent, the default domain.xml configuration will be used.
configurationReadOnly	Boolean	false	If true deployment changes are not written to the configuration and persisted.
sunResourcesXml	String		The relative or absolute path to the sun-resources.xml file that will be used to add resources to the instance using the add-resources asadmin command.





Example of Maven profile setup

```
<profile>
  <id>glassfish-embedded-3.1</id>
  <dependencies>
    <dependency>
      <groupId>org.jboss.arquillian.container</groupId>
      <artifactId>arquillian-glassfish-embedded-3</artifactId>
      <version>1.0.0-SNAPSHOT</version>
    </dependency>
    <dependency>
      <groupId>org.glassfish.extras</groupId>
      <artifactId>glassfish-embedded-all</artifactId>
      <version>3.1</version>
    </dependency>
  </dependencies>
</profile>
```

12.9. GlassFish 3.1 - Remote

A DeployableContainer implementation that connects to a remote GlassFish 3.1 instance and deploys the test archive using the admin REST api.

Table 12.17. Container Injection Support Matrix

@EJB	@EJB interface)	(no- @Inject (CDI)	@Inject (MC)	@PersistenceContext @PersistenceUnit
				

12.9.1. Configuration

Default Protocol: [Servlet 3.0](#)

Table 12.18. Container Configuration Options

Name	Type	Default	Description
remoteServerAdminPort	int	4848	The administrative port the client should connect to.
remoteServerAddress	String	localhost	The administrative address the client should connect to.
remoteServerAdminHttps	boolean	false	Use SSL for communicating with the admin server.
remoteServerHttps	boolean	false	Use SSL to communicate with application.
remoteServerHttpPort	int	8080	The HTTP port of the remote server.



Example of Maven profile setup

```
<profile>
  <id>glassfish-remote-3.1</id>
  <dependencies>
    <dependency>
      <groupId>org.jboss.arquillian.container</groupId>
      <artifactId>arquillian-glassfish-remote-3</artifactId>
      <version>1.0.0-SNAPSHOT</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</profile>
```

12.10. Tomcat 6.0 - Embedded

A DeployableContainer implementation that manages the complete lifecycle of an embedded (same JVM) Tomcat 6 Servlet Container. (Keep in mind that only select EE APIs are available in Tomcat 6, such as JNDI and Servlet 2.5). Test archives are adapted to Tomcat's StandardContext API by ShrinkWrap and deployed programmatically.

Table 12.19. Container Injection Support Matrix

@Reso	@EJB	@EJB (no- interface)	(no- @Inject (CDI)	@Inject (MC)	@PersistenceContext @PersistenceUnit
					



Warning

CDI support requires use of Weld Servlet and associated configuration. The WAR will have to be unpacked as well in order for Weld to locate the classes. See the following configuration example.

```
<?xml version="1.0" encoding="UTF-8"?>
<arquillian xmlns="http://jboss.com/arquillian"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:tomcat6="urn:arq:org.jboss.arquillian.container.tomcat.embedded_6">

  <tomcat6:container>
    <!-- unpackArchive must be true if using the Weld Servlet module -->
    <tomcat6:unpackArchive>true</tomcat6:unpackArchive>
  </tomcat6:container>

</arquillian>
```

Running an in-container test on Tomcat 6 currently requires that you add the Arquillian Protocol Servlet to the test archive's web.xml, a temporary measure until [ARQ-217](#) is resolved. The listing below shows a minimum web.xml containing the required Servlet mapping:

```
<?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>ServletTestRunner</servlet-name>
  <servlet-class>org.jboss.arquillian.protocol.servlet_3.ServletTestRunner</servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>ServletTestRunner</servlet-name>
  <url-pattern>/ArquillianServletRunner</url-pattern>
</servlet-mapping>

</web-app>
```

If you forget to add this Servlet mapping for a test using the in-container run mode, you will get a failure with the message "Kept getting 404s" because Arquillian can't communicate with the deployed application.

12.10.1. Configuration

Default Protocol: [Servlet 2.5](#)

Table 12.20. Container Configuration Options

Name	Type	Default	Description
bindHttpPort	int	9090	The HTTP port the server should bind to.
bindAddress	String	localhost	The host the server should be run on.
tomcatHome	String		Optional location of a Tomcat installation to link against.
serverName	String		Optional name of the server
appBase	String		Optional relative or absolute path to the directory where applications are deployed (e.g., webapps).
workDir	String		Optional relative or absolute path to the directory where applications are expanded and session serialization data is stored (e.g., work).
unpackArchive	boolean	true	Specify if the deployment should be deployed exploded or compressed.

Example of Maven profile setup

```
<profile>
  <id>tomcat-embedded</id>
  <dependencies>
    <dependency>
      <groupId>org.jboss.arquillian.container</groupId>
      <artifactId>arquillian-tomcat-embedded-6</artifactId>
      <version>1.0.0-SNAPSHOT</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.apache.tomcat</groupId>
      <artifactId>catalina</artifactId>
      <version>6.0.29</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.apache.tomcat</groupId>
      <artifactId>coyote</artifactId>
      <version>6.0.29</version>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>org.apache.tomcat</groupId>
      <artifactId>jasper</artifactId>
      <version>6.0.29</version>
      <scope>provided</scope>
    </dependency>
    <!-- Weld servlet, EL and JSP required for testing CDI injections -->
    <dependency>
      <groupId>org.jboss.weld.servlet</groupId>
      <artifactId>weld-servlet</artifactId>
      <version>1.0.1-Final</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.glassfish.web</groupId>
      <artifactId>el-impl</artifactId>
      <version>2.2</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>javax.servlet.jsp</groupId>
      <artifactId>jsp-api</artifactId>
```

```



    <version>2.2</version>
    <scope>test</scope>
  </dependency>
</dependencies>
</profile>

```

12.11. Jetty 6.1 - Embedded

A DeployableContainer implementation that can run and connect to a embedded (same JVM) Jetty 6.1 Servlet Container. The minimum recommended version is Jetty 6.1.12, though you can use an earlier 6.1 version if you aren't using JNDI resources. Only select EE APIs are available, such as JNDI and Servlet 2.5. This implementation has lifecycle support, so the container will be started and stopped as part of the test run.

Table 12.21. Container Injection Support Matrix

@Reso	@EJB	@EJB (no- interface)	(no- @Inject (CDI)	@Inject (MC)	@PersistenceContext @PersistenceUnit
					



Warning

CDI support requires use of Weld Servlet.

12.11.1. Configuration

Default Protocol: [Servlet 2.5](#)

Table 12.22. Container Configuration Options

Name	Type	Default	Description
bindHttpPort	int	9090	The HTTP port the server should bind to.
bindAddress	String	localhost	The host the server should be run on.
jettyPlus	boolean	true	Activates the Jetty plus configuration to support JNDI resources (requires jetty-plus and jetty-naming artifacts on the classpath).
configurationClasses	String	null	Specify your own Jetty configuration classes as a comma separated list.

Example of Maven profile setup

```
<profile>
  <id>jetty-embedded</id>
  <dependencies>
    <dependency>
      <groupId>org.jboss.arquillian.container</groupId>
      <artifactId>arquillian-jetty-embedded-6.1</artifactId>
      <version>1.0.0-SNAPSHOT</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.mortbay.jetty</groupId>
      <artifactId>jetty</artifactId>
      <version>6.1.12</version>
      <scope>test</scope>
    </dependency>
    <!-- plus and naming requires for using JNDI -->
    <dependency>
      <groupId>org.mortbay.jetty</groupId>
      <artifactId>jetty-plus</artifactId>
      <version>6.1.12</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.mortbay.jetty</groupId>
      <artifactId>jetty-naming</artifactId>
      <version>6.1.12</version>
      <scope>test</scope>
    </dependency>
    <!-- Weld servlet, EL and JSP required for testing CDI injections -->
    <dependency>
      <groupId>org.jboss.weld.servlet</groupId>
      <artifactId>weld-servlet</artifactId>
      <version>1.0.1-Final</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.glassfish.web</groupId>
      <artifactId>el-impl</artifactId>
      <version>2.2</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>javax.servlet.jsp</groupId>
```

```



<artifactId>jsp-api</artifactId>
<version>2.2</version>
<scope>test</scope>
</dependency>
</dependencies>
</profile>

```

12.12. Jetty 7.0 - Embedded

A DeployableContainer implementation that can run and connect to a embedded (same JVM) Jetty 7 Servlet Container. Only select EE APIs are available, such as JNDI and parts of Servlet (support for web-fragment.xml is the important bit). This implementation has lifecycle support, so the container will be started and stopped as part of the test run.

Table 12.23. Container Injection Support Matrix

@Reso	@EJB	@EJB (no- interface)	(no- @Inject (CDI)	@Inject (MC)	@PersistenceContext @PersistenceUnit
					



Warning

CDI support requires use of Weld Servlet.

12.12.1. Configuration

Default Protocol: [Servlet 3.0](#)

Table 12.24. Container Configuration Options

Name	Type	Default	Description
bindHttpPort	int	9090	The HTTP port the server should bind to.
bindAddress	String	localhost	The host the server should be run on.
jettyPlus	boolean	true	Activates the Jetty plus configuration to support JNDI resources (requires jetty-plus and jetty-naming artifacts on the classpath).
configurationClasses	String	null	Specify your own Jetty configuration classes as a comma separated list.




Example of Maven profile setup


```
<profile>
  <id>jetty-embedded</id>
  <dependencies>
    <dependency>
      <groupId>org.jboss.arquillian.container</groupId>
      <artifactId>arquillian-jetty-embedded-7</artifactId>
      <version>1.0.0-SNAPSHOT</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.eclipse.jetty</groupId>
      <artifactId>jetty-webapp</artifactId>
      <version>7.0.2.v20100331</version>
      <scope>test</scope>
    </dependency>
    <!-- plus and naming requires for using JNDI -->
    <dependency>
      <groupId>org.eclipse.jetty</groupId>
      <artifactId>jetty-plus</artifactId>
      <version>7.0.2.v20100331</version>
      <scope>test</scope>
    </dependency>
    <!-- Weld servlet, EL and JSP required for testing CDI injections -->
    <dependency>
      <groupId>org.jboss.weld.servlet</groupId>
      <artifactId>weld-servlet</artifactId>
      <version>1.0.1-Final</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.glassfish.web</groupId>
      <artifactId>el-impl</artifactId>
      <version>2.2</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>javax.servlet.jsp</groupId>
      <artifactId>jsp-api</artifactId>
      <version>2.2</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</profile>
```

12.13. Weld SE 1.0 - Embedded

A DeployableContainer implementation that can run and connect to a embedded(same JVM) Weld(CDI reference implementation) SE edition. No EE APIs are available. This implementation has lifecycle support, so the container will be started and stopped as part of the test run.

Table 12.25. Container Injection Support Matrix

@EJB	@EJB (no- interface)	(no- @Inject (CDI)	@Inject (MC)	@PersistenceContext @PersistenceUnit
				



Warning

Local EJBs only, which get treated as managed beans. Transactions, security and EJB context injection are not applied.

12.13.1. Configuration

Default Protocol: [Local](#)

Table 12.26. Container Configuration Options

Name	Type	Default	Description

Example of Maven profile setup

```
<profile>
  <id>weld-se-embedded-1</id>
  <dependencies>
    <dependency>
      <groupId>org.jboss.arquillian.container</groupId>
      <artifactId>arquillian-weld-se-embedded-1</artifactId>
      <version>1.0.0-SNAPSHOT</version>
    </dependency>
    <dependency>
      <groupId>org.jboss.weld</groupId>
      <artifactId>weld-core</artifactId>
    </dependency>
    <dependency>
      <groupId>org.jboss.weld</groupId>
      <artifactId>weld-api</artifactId>
    </dependency>
  </dependencies>
</profile>
```

```




<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-simple</artifactId>
</dependency>
</dependencies>
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.jboss.weld</groupId>
      <artifactId>weld-core-bom</artifactId>
      <version>1.0.1-SP1</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
</profile>

```

12.14. Weld SE 1.1 - Embedded

A DeployableContainer implementation that can run and connect to a embedded(same JVM) Weld(CDI reference implementation) SE edition. No EE APIs are available. This implementation has lifecycle support, so the container will be started and stopped as part of the test run.

Table 12.27. Container Injection Support Matrix

@EJB	@EJB (no- interface)	@Inject (CDI)	@Inject (MC)	@PersistenceContext @PersistenceUnit
				



Warning

Local EJBs only, which get treated as managed beans. Transactions, security and EJB context injection are not applied.

12.14.1. Configuration

Default Protocol: [Local](#)

Table 12.28. Container Configuration Options

Name	Type	Default	Description




Example of Maven profile setup

```
<profile>
  <id>weld-se-embedded-11</id>
  <dependencies>
    <dependency>
      <groupId>org.jboss.arquillian.container</groupId>
      <artifactId>arquillian-weld-se-embedded-1</artifactId>
      <version>1.0.0-SNAPSHOT</version>
    </dependency>
    <dependency>
      <groupId>org.jboss.weld</groupId>
      <artifactId>weld-core</artifactId>
    </dependency>
    <dependency>
      <groupId>org.jboss.weld</groupId>
      <artifactId>weld-api</artifactId>
    </dependency>
    <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-simple</artifactId>
    </dependency>
  </dependencies>
  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>org.jboss.weld</groupId>
        <artifactId>weld-core-bom</artifactId>
        <version>1.1.0.Final</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>
</profile>
```

12.15. Weld EE 1.1 - Embedded

A DeployableContainer implementation that can run and connect to a embedded(same JVM) Weld(CDI reference implementation) EE version. Mock EE APIs are available. This implementation has lifecycle support, so the container will be started and stopped as part of the test run.

Table 12.29. Container Injection Support Matrix

@EJB	@EJB (no- interface)	@Inject (CDI)	@Inject (MC)	@PersistenceContext @PersistenceUnit
				

**Warning**

Local EJBs only, which get treated as managed beans. Transactions, security and EJB context injection are not applied.

12.15.1. Configuration

Default Protocol: [Local](#)

Table 12.30. Container Configuration Options

Name	Type	Default	Description
enableConversationScope	boolean	false	Activate ConversationScope between @Test methods. Use this to simulate Weld Servlet HTTP Conversation scope support.

Example of Maven profile setup

```
<profile>
  <id>weld-ee-embedded-1.1</id>
  <dependencies>
    <dependency>
      <groupId>org.jboss.arquillian.container</groupId>
      <artifactId>arquillian-weld-ee-embedded-1.1</artifactId>
      <version>1.0.0-SNAPSHOT</version>
    </dependency>
    <dependency>
      <groupId>org.jboss.weld</groupId>
      <artifactId>weld-core</artifactId>
    </dependency>
    <dependency>
      <groupId>org.jboss.weld</groupId>
      <artifactId>weld-api</artifactId>
    </dependency>
    <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-simple</artifactId>
    </dependency>
  </dependencies>
</profile>
```

```
</dependency>
</dependencies>
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.jboss.weld</groupId>
      <artifactId>weld-core-bom</artifactId>
      <version>1.1.0.Final</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
</profile>
```

To run Weld EE Embedded you also need the Java EE APIs. These APIs might be provided to you by other dependencies like `org.jboss.jbossas:jboss-as-client`, `org.jboss.spec:jboss-javaee-6.0` or `org.glassfish.extras:glassfish-embedded-all`.




```
<dependency>
  <groupId>javax.el</groupId>
  <artifactId>el-api</artifactId>
  <version>2.2</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.glassfish.web</groupId>
  <artifactId>el-impl</artifactId>
  <version>2.1.2-b04</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.jboss.spec.javax.servlet</groupId>
  <artifactId>jboss-servlet-api_3.0_spec</artifactId>
  <version>1.0.0.Beta2</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>javax.validation</groupId>
  <artifactId>validation-api</artifactId>
  <version>1.0.0.GA</version>
  <scope>test</scope>
</dependency>
```

```
<dependency>
  <groupId>javax.transaction</groupId>
  <artifactId>jta</artifactId>
  <version>1.1</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>javax.persistence</groupId>
  <artifactId>persistence-api</artifactId>
  <version>1.0</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.jboss.ejb3</groupId>
  <artifactId>jboss-ejb3-api</artifactId>
  <version>3.1.0</version>
  <scope>test</scope>
</dependency>
```

12.16. Apache OpenWebBeans 1.0 - Embedded

A DeployableContainer implementation that can run and connect to a embedded(same JVM) WeldApache OpenWebBeans(CDI) instance. No EE APIs are available. This implementation has lifecycle support, so the container will be started and stopped as part of the test run.

Table 12.31. Container Injection Support Matrix

@EJB	@EJB interface)	(no- @Inject (CDI)	@Inject (MC)	@PersistenceContext @PersistenceUnit
				



Warning

Local EJBs only, which get treated as managed beans. Transactions, security and EJB context injection are not applied.

12.16.1. Configuration

Default Protocol: [Local](#)

Table 12.32. Container Configuration Options

Name	Type	Default	Description

Example of Maven profile setup

```
<profile>
  <id>openwebbeans-embedded-1</id>
  <dependencies>
    <dependency>
      <groupId>org.jboss.arquillian.container</groupId>
      <artifactId>arquillian-openwebbeans-embedded-1</artifactId>
      <version>1.0.0-SNAPSHOT</version>
    </dependency>
    <dependency>
      <groupId>org.apache.openwebbeans</groupId>
      <artifactId>openwebbeans-spi</artifactId>
    </dependency>
    <dependency>
      <groupId>org.apache.openwebbeans</groupId>
      <artifactId>openwebbeans-impl</artifactId>
    </dependency>
    <dependency>
      <groupId>org.apache.geronimo.specs</groupId>
      <artifactId>geronimo-el_2.2_spec</artifactId>
    </dependency>
    <dependency>
      <groupId>org.apache.geronimo.specs</groupId>
      <artifactId>geronimo-jta_1.1_spec</artifactId>
    </dependency>
    <dependency>
      <groupId>org.apache.geronimo.specs</groupId>
      <artifactId>geronimo-validation_1.0_spec</artifactId>
    </dependency>
    <dependency>
      <groupId>org.apache.geronimo.specs</groupId>
      <artifactId>geronimo-interceptor_1.1_spec</artifactId>
    </dependency>
    <dependency>
      <groupId>org.apache.geronimo.specs</groupId>
      <artifactId>geronimo-jcdi_1.0_spec</artifactId>
    </dependency>
    <dependency>
      <groupId>org.apache.geronimo.specs</groupId>
      <artifactId>geronimo-atinject_1.0_spec</artifactId>
    </dependency>
    <dependency>
      <groupId>org.apache.geronimo.specs</groupId>
```



```


    <artifactId>geronimo-servlet_2.5_spec</artifactId>
  </dependency>
</dependencies>
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.apache.openwebbeans</groupId>
      <artifactId>openwebbeans</artifactId>
      <version>1.0.0</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
</profile>

```

12.17. Apache OpenEJB 3.1 - Embedded

A DeployableContainer implementation that can run and connect to a embedded(same JVM) Apache OpenEJB instance. EJB 3.0 APIs are available, but no JMS. This implementation has lifecycle support, so the container will be started and stopped as part of the test run.

Table 12.33. Container Injection Support Matrix

@EJB	@EJB (no- interface)	@Inject (CDI)	@Inject (MC)	@PersistenceContext @PersistenceUnit
				

12.17.1. Configuration

Default Protocol: [Local](#)

Table 12.34. Container Configuration Options

Name	Type	Default	Description
openEjbXml	String		Specify the OpenEJB XML configuration file.
jndiProperties	String		Specify the OpenEJB properties configuration file.

Example of Maven profile setup

```
<profile>
```

```
<id>openejb-embedded-3.1</id>
<dependencies>
  <dependency>
    <groupId>org.jboss.arquillian.container</groupId>
    <artifactId>arquillian-openejb-3.1</artifactId>
    <version>1.0.0-SNAPSHOT</version>
  </dependency>
  <dependency>
    <groupId>org.apache.openejb</groupId>
    <artifactId>openejb-core</artifactId>
    <version>3.1.4</version>
  </dependency>
</dependencies>
</profile>
```

Complete Protocol Reference

13.1. Local

The Local Protocol implementation is used by most EE5 compliant containers. It does nothing to the deployment. The Local Protocol is also used when executing in run mode as `client`.

Table 13.1. Packaging rules

@Deployment	Output	Action
JavaArchive	JavaArchive	Does nothing.
WebArchive	WebArchive	Does nothing.
EnterpriseArchive	EnterpriseArchive	Does nothing.

13.1.1. Configuration

Table 13.2. Protocol Configuration Options

Name	Type	Default	Description

13.2. Servlet 2.5

The Servlet 2.5 Protocol implementation is used by most EE5 compliant containers. It will attempt to add a war to the deployment.

Table 13.3. Packaging rules

@Deployment	Output	Action
JavaArchive	EnterpriseArchive	Create a new <code>EnterpriseArchive</code> , add <code>@Deployment</code> and <code>ServletProtocol</code> as module, the other <code>Auxiliary Archives</code> as libraries.
WebArchive	WebArchive	If a <code>web.xml</code> is found, a <code>Servlet</code> will be added, else a <code>web.xml</code> will be added. The <code>Servlet WebArchive</code> will be merged with the <code>Deployment</code> and the <code>Auxiliary Archives</code> added as libraries.
EnterpriseArchive	EnterpriseArchive	Same as <code>JavaArchive</code> , but using the <code>@Deployment</code> defined <code>EnterpriseArchive</code> instead of creating a new.

13.2.1. Configuration

Table 13.4. Protocol Configuration Options

Name	Type	Default	Description
host	String	none	Used to override the Deployments default hostname.
port	String	none	Used to override the Deployments default http port.
contextRoot	int	none	Used to override the Deployments default contextRoot.

13.3. Servlet 3.0

The Servlet 3.0 Protocol implementation is used by most EE6 compliant containers. It will attempt to add a web-fragment to the deployment.

Table 13.5. Packaging rules

@Deployment	Output	Action
JavaArchive	WebArchive	Creates a new WebArchive, adds @Deployment and Auxiliary Archives as libraries.
WebArchive	WebArchive	Adds @Deployment and Auxiliary Archives as libraries.
EnterpriseArchive	EnterpriseArchive	If a single WebArchive is found, the same as for WebArchive is done. If no WebArchives are found a new one is creates, adds @Deployment and Auxiliary Archives as libraries. If multiple WebArchives are found, a exception is thrown.

13.3.1. Configuration

Table 13.6. Protocol Configuration Options

Name	Type	Default	Description
host	String	none	Used to override the Deployments default hostname.
port	String	none	Used to override the Deployments default http port.
contextRoot	int	none	Used to override the Deployments default contextRoot.