Arquillian: An integration testing framework for Java EE

Reference Guide

1.0.0-SNAPSHOT

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

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. It turns out, if these tests execute quickly, they're really the only tests you need.

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 top-level discussions forum in the 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 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

The mission of the Arquillian project is to provide a simple test harness that developers can use to produce a broad range of integration tests for their Java applications (most likely enterprise applications). A test case may be executed within the container, deployed alongside the code under test, or by coordinating with the container, acting as a client to the deployed code.

Arquillian defines two styles of container, remote and embedded. A remote container resides in a separate JVM from the test runner. Its lifecycle may be managed by Arquillian, or Arquillian may bind to a container that is already started. An embedded container resides in the same JVM and is mostly likely managed by Arquillian. Containers can be further classified by their capabilities. Examples include a fully compliant Java EE application server (e.g., GlassFish, JBoss AS, Embedded GlassFish), a Servlet container (e.g., Tomcat, Jetty) and a bean container (e.g., Weld SE). Arquillian ensures that the container used for testing is pluggable, so the developer is not locked into a proprietary testing environment.

Arquillian seeks to minimize the burden on the developer to carry out integration testing by handling all aspects of test execution, including:

- managing the lifecycle of the container (start/stop),
- bundling the test class with dependent classes and resources into a deployable archive,
- enhancing the test class (e.g., resolving @Inject, @EJB and @Resource injections),
- deploying the archive to test (deploy/undeploy) and
- capturing results and failures.

To avoid introducing unnecessary complexity into the developer's build environment, Arquillian integrates transparently with familiar testing frameworks (e.g., JUnit 4, TestNG 5), allowing tests to be launched using existing IDE, Ant and Maven test plugins without any add-ons.

Arquillian makes integration testing a breeze.

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 explainations 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 been 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 to 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 being 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 SVN. You can choose between a *JUnit example* [http://anonsvn.jboss.org/repos/common/arquillian/tags/1.0.0.Alpha1/examples/junit] and a *TestNG example* [http://anonsvn.jboss.org/repos/common/arquillian/tags/1.0.0.Alpha1/examples/testng]. In this tutorial we show you how to use both.

```
svn co http://anonsvn.jboss.org/repos/common/arquillian/trunk/examples/testng/ arquillian-example-testng
svn co http://anonsvn.jboss.org/repos/common/arquillian/trunk/examples/junit/ arquillian-example-junit
```

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 2), for which we use the `jbossas-remote-60` profile.

First, make sure you have a copy of JBoss AS; you can download it from *jboss.org* [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.M2.zip && mv jboss-6.0.0.20100216-M2 $JBOSS_HOME && $JBOSS_HOME/bin/run.sh
```

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

```
$ cd arquillian-example-testng/
$ mvn test -Pjbossas-remote-60

$ cd ../arquillian-example-junit/
$ mvn test -Pjbossas-remote-60
```

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

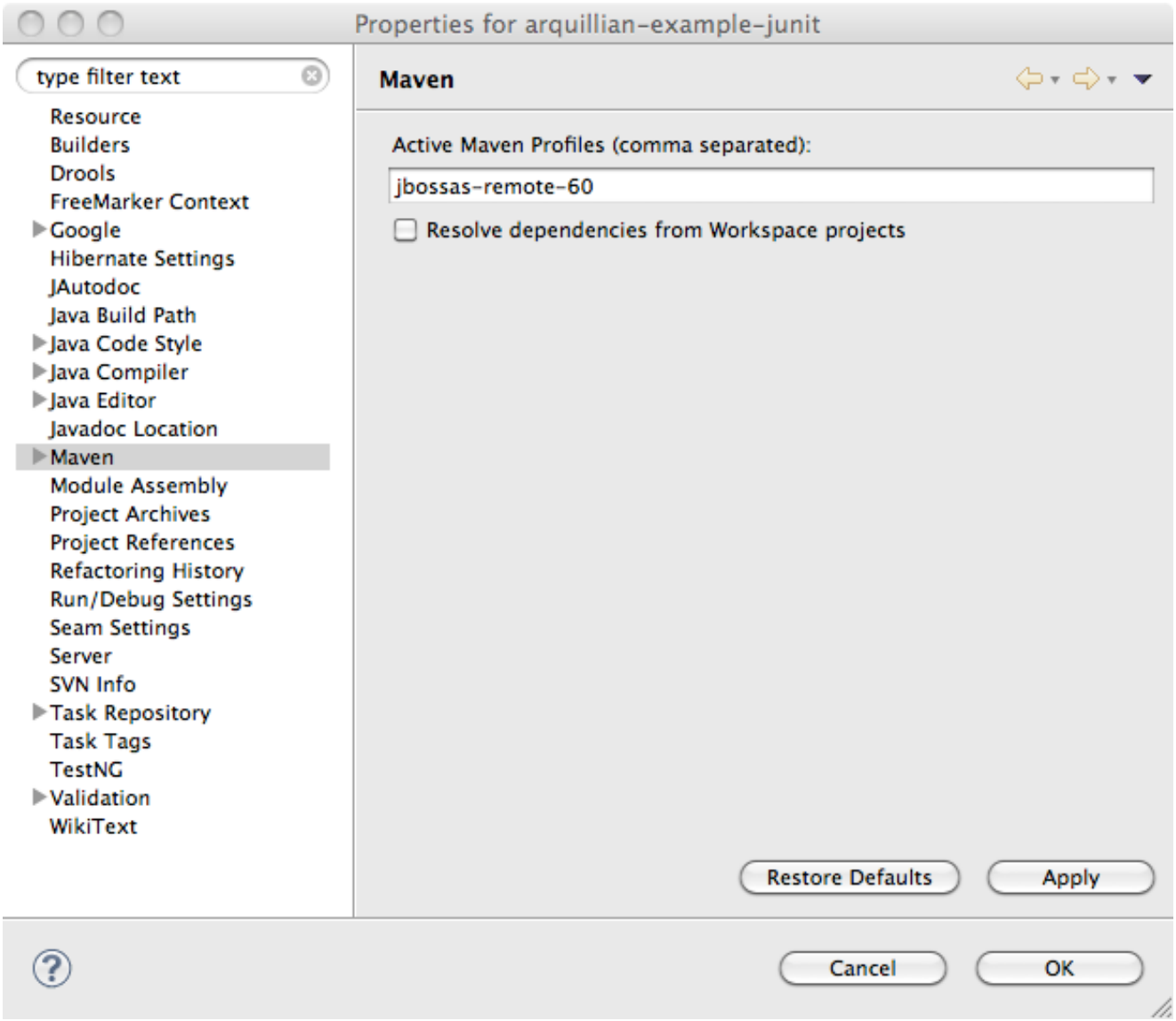
JUnit. If you are writing your tests with TestNG, you need the Eclipse *TestNG plugin* [http://testng.org].

Note

You *must* use the 5.11 version of the TestNG Eclipse plugin, which can be downloaded from *testng.org* [http://testng.org/testng-eclipse-5.11.0.18.zip]. The TestNG update site will give you version 5.12 which is not compatible with any released version of TestNG core.

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].

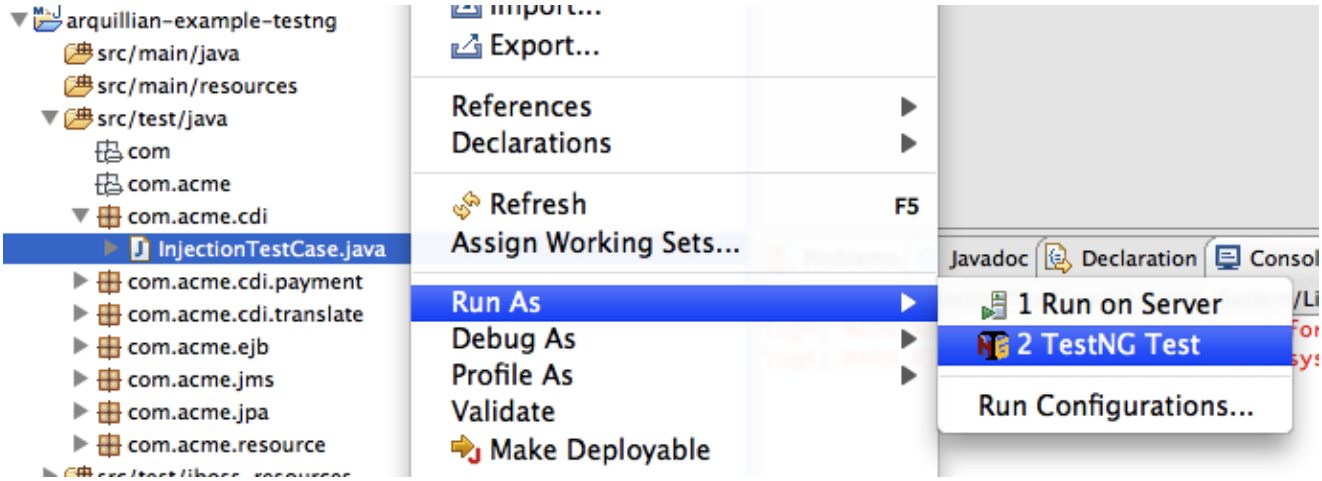
Once the plugins are installed, import your Maven project into the Eclipse workspace. Before executing the test, you need to enable the profile for the target container, as we did on the command line. We'll go ahead and activate the profile globally for the project (we also need the `default` profile, 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-60`; 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.

Asssuming 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.Archives;
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 Archives.create("test.jar", JavaArchive.class)
      .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 an 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.Archives;
import org.jboss.shrinkwrap.api.ArchivePaths;
import org.jboss.shrinkwrap.api.spec.JavaArchive;
import org.jboss.shrinkwrap.impl.base.asset.ByteArrayAsset;
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 Archives.create("test.jar", JavaArchive.class)
      .addClasses(GreetingManager.class, GreetingManagerBean.class)
      .addManifestResource(new ByteArrayAsset(new byte[0]),
          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 Archives.create("test.jar", JavaArchive.class)
      .addClasses(User.class, UserRepository.class, UserRepositoryBean.class)
      .addManifestResource(
        "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.Archives;
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 Archives.create("test.jar", JavaArchive.class)
      .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.Alpha1</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>4.6</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>5.10</version>
  <classifier>jdk15</classifier>
  <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 convertToFarenheit(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 Archives.create("test.jar", JavaArchive.class)
    .addClasses(TemperatureConverter.class)
    .addManifestResource(
        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 Archives.create("test.jar", JavaArchive.class)
      .addClasses(TemperatureConverter.class)
      .addManifestResource(
        new ByteArrayAsset("<beans/>".getBytes()),
        ArchivePaths.create("beans.xml"));
  }

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

  @Test
  public void testConvertToFarenheit() {
    Assert.assertEquals(converter.convertToFarenheit(0d), 32d);
    Assert.assertEquals(converter.convertToFarenheit(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 Archives.create("test.jar", JavaArchive.class)
      .addClasses(TemperatureConverter.class)
      .addManifestResource(
        new ByteArrayAsset("<beans/>".getBytes()),
        ArchivePaths.create("beans.xml"));
  }

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

  @Test
  public void testConvertToFarenheit() {
    Assert.assertEquals(converter.convertToFarenheit(0d), 32d);
    Assert.assertEquals(converter.convertToFarenheit(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/], 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-60</id>
    <dependencies>
      <dependency>
        <groupId>org.jboss.arquillian.container</groupId>
        <artifactId>arquillian-jbossas-remote-60</artifactId>
        <version>${arquillian.version}</version>
      </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-60` profile activated:

```
mvn test -Pjbossas-remote-60
```

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

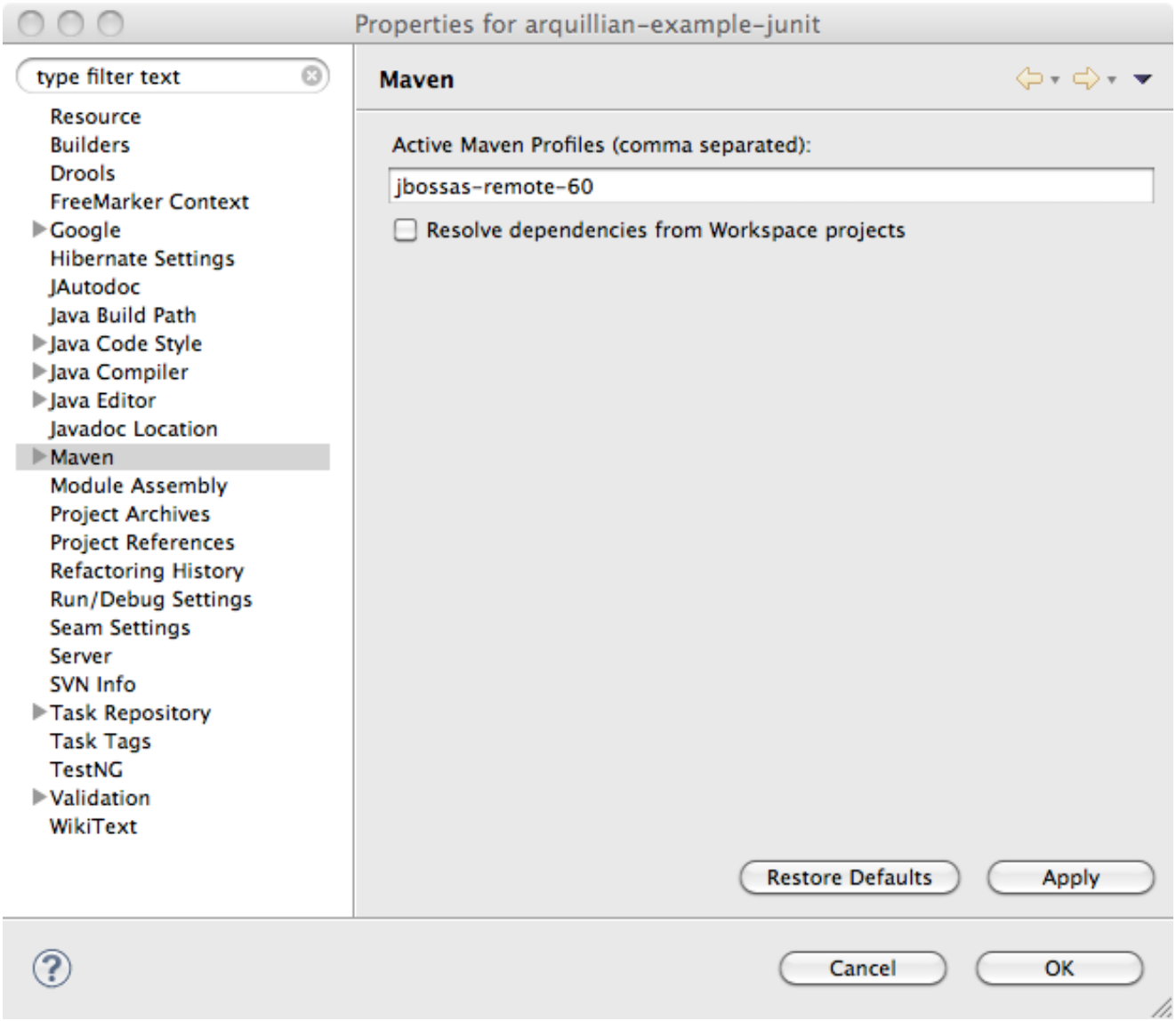
Before running an Arquillian test in Eclipse, you must have the plugin for the unit testing framework you are using installed. Eclipse ships with the JUnit plugin, so you are already setup if you selected JUnit. If you are writing your tests with TestNG, you need the Eclipse *TestNG plugin* [http://testng.org].

Note

You *must* use the 5.11 version of the TestNG Eclipse plugin, which can be downloaded from *testng.org* [http://testng.org/testng-eclipse-5.11.0.18.zip]. The TestNG update site will give you version 5.12 which is not compatible with any released version of TestNG core.

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].

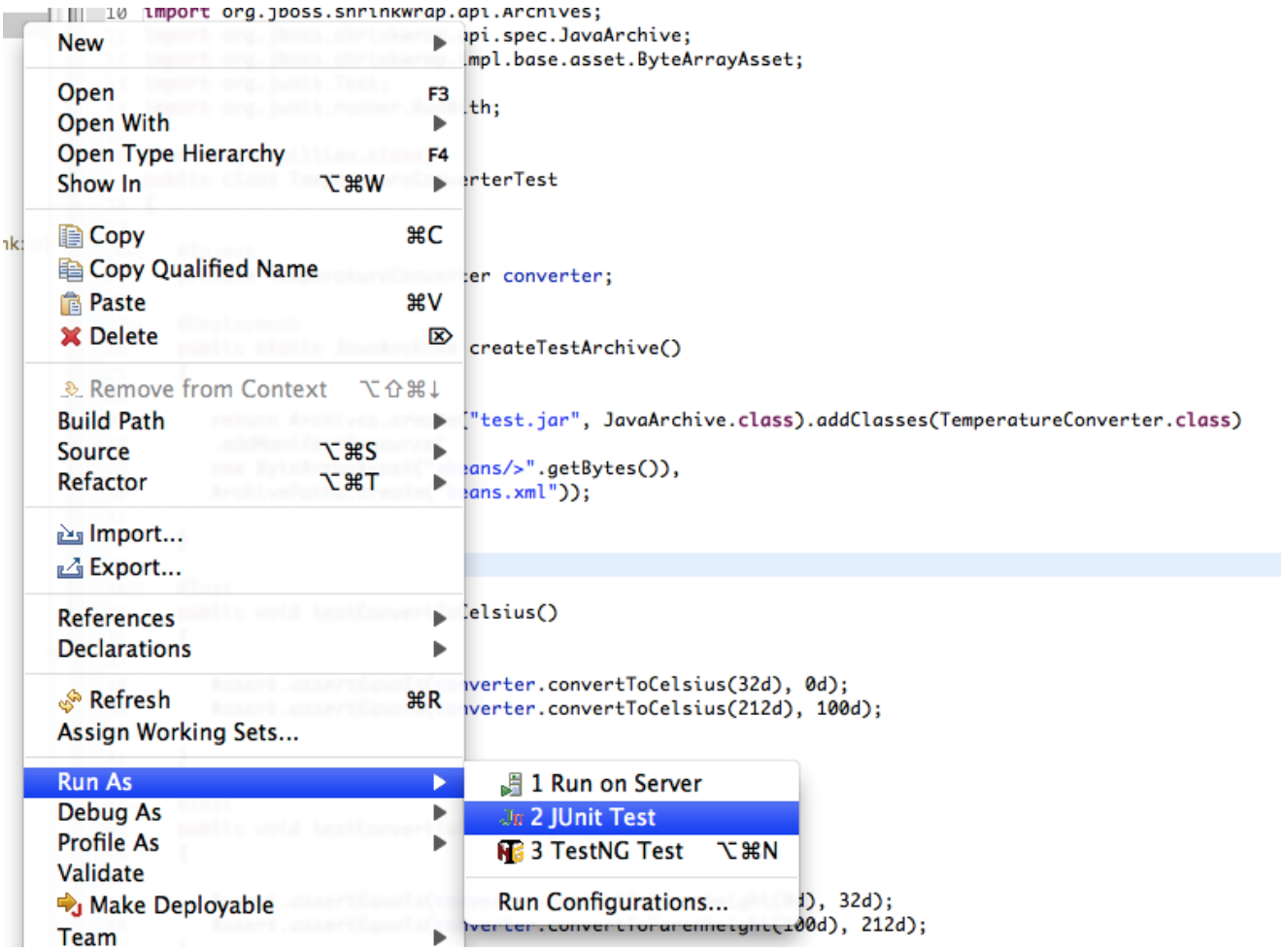
Once the plugins are installed, import your Maven project into the Eclipse workspace. Before executing the test, you need to enable the profile for the target container, as you did in the previous section. We'll go ahead and activate the profile globally for the project. Right click on the project and select Properties. Select the Maven property sheet and in the first form field, enter `jbossas-remote-60`; you also need to tell Maven to not resolve depedencies 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.

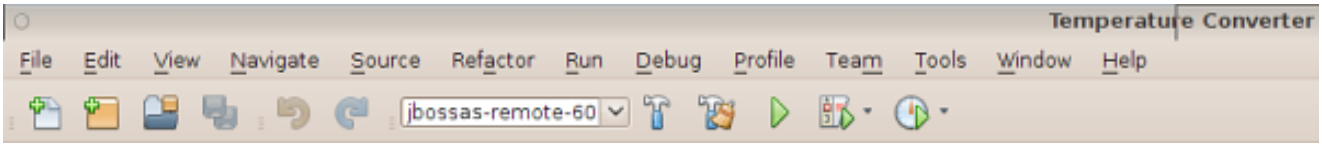


Running the the JUnit test in Eclipse

3.5. Setting up and running the test in NetBeans

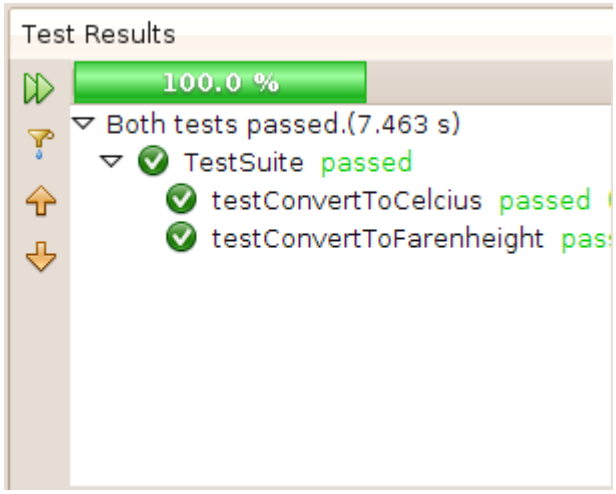
Things get even simpler when using NetBeans 6.8 or better. NetBeans ships with native Maven 2 support and, rather than including a test plugin for each unit testing framework, it has a generic test plugin which delegates to the Maven surefire plugin to execute the tests.

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-60` 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 windown, 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

There are two styles of containers that you can target in Arquillian:

1. remote — resides in a separate JVM from the test runner; its lifecycle may be managed by Arquillian, or Arquillian may bind to a container that is already started
2. embedded — resides in the same JVM as the test runner; its lifecycle is likely managed by Arquillian

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

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. 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 5.1	remote	Java EE 5	arquillian-jbossas-remote-51
JBoss AS 6.0 M2	remote	Java EE 6	arquillian-jbossas-remote-60
Embedded GlassFish V3	embedded	Java EE 6	arquillian-glassfish-embedded-30
Weld SE	embedded	CDI	arquillian-weld-embedded
Apache OpenEJB	embedded	EJB 3.0	arquillian-openejb

Support for other containers is planned, including GlassFish V3(remote), Tomcat and Jetty. We don't have plans to provide implementations for Spring or Guice at this time, but we welcome a contribution from the community, because it's certainly possible.

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 onus 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/" + unqualified interface name + "Bean"
```

```
"test/" + unqualified interface name + "Bean/local"
```

```
"test/" + unqualified interface name + "Bean/remote"
```

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 a 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 Archives.create("test.jar", JavaArchive.class)
    .addClass(NameOfClassUnderTest.class)
    .addManifestResource(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 SE 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 choosen (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 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-6/api/] 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 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.DeployableContainer`. If you recall from the getting started section, we included an Arquillian library according to the target container we wanted to us. 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 be 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? Technially 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 transfered 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.

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 to 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] or the *JBoss Tools documentation* [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] 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)

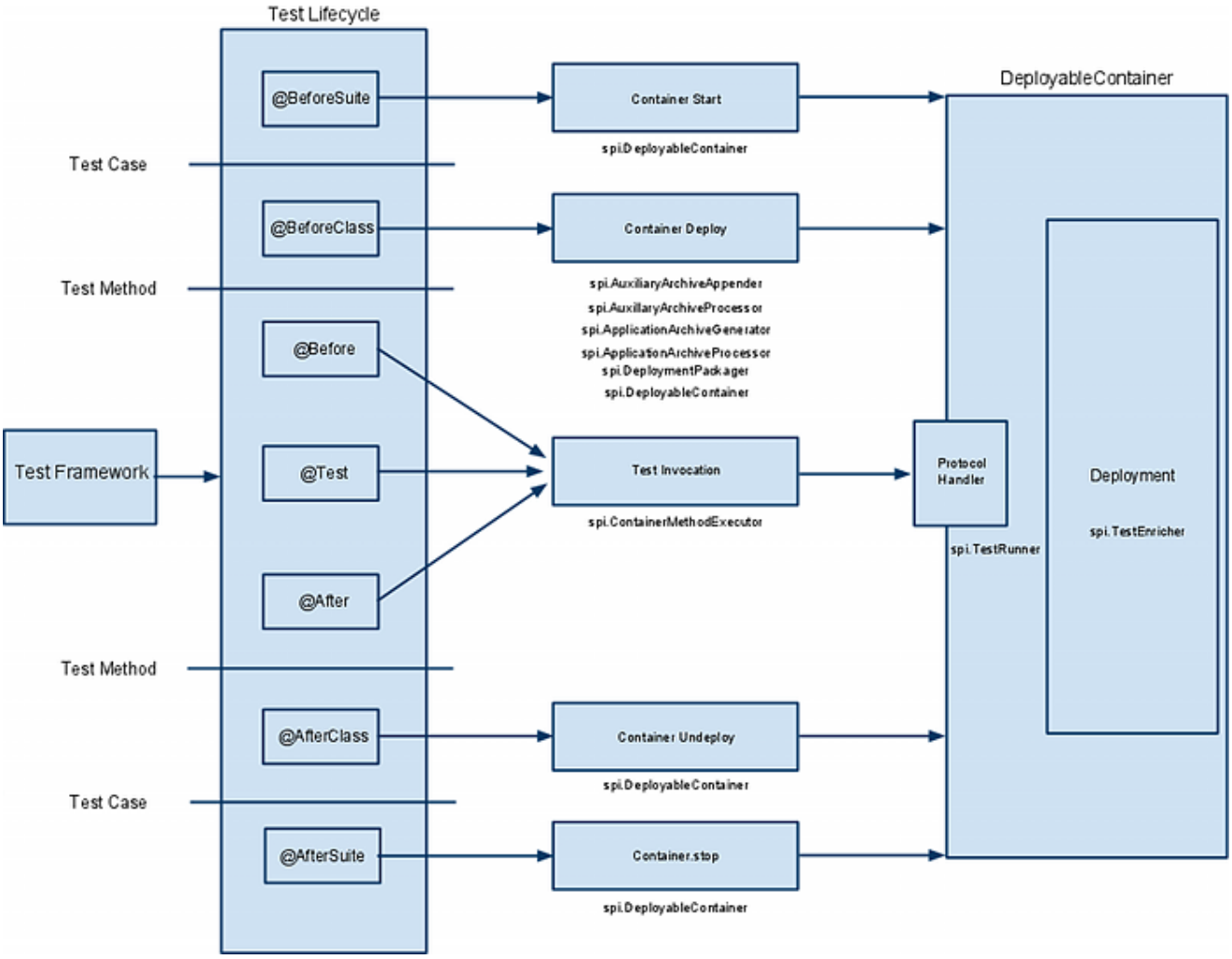
```
set "JAVA_OPTS=%JAVA_OPTS% -ea"
```

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.

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