Community Experience Distilled

Arquillian Testing Guide

Get familiarized with the Arquillian framework and its capabilities to carry out integration and functional testing on a Java virtual machine

John D. Ament

[PACKT] open source*
PUBLISHING
community experience distilled

Arquillian Testing Guide

Get familiarized with the Arquillian framework and its capabilities to carry out integration and functional testing on a Java virtual machine

John D. Ament

[PACKT] PUBLISHING open source community experience distilled

BIRMINGHAM - MUMBAI

Arquillian Testing Guide

First published: April 2013

Production Reference: 1050413

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78216-070-0

www.packtpub.com

Cover Image by Al Viciedo (viciedophotography@gmail.com)

Credits

Author
John D. Ament

Reviewers
Hantsy Bai
Jakub Narloch

Acquisition Editor
James Jones

Commissioning Editor
Llewellyn F. Rozario

Lead Technical Editor
Neeshma Ramakrishnan

Technical Editors
Kirti Pujari
Nitee Shetty

Project Coordinator
Anugya Khurana

Proofreader
Sandra Hopper

Indexer
Monica Ajmera

Graphics
Aditi Gajjar

Production Coordinator
Conidon Miranda

Cover Work
Conidon Miranda

About the Author

John D. Ament was born to a beautician and a taxi driver in the warm summer of 1983 in Ridgewood, New York. At the age of six his family moved to the northern suburbs of New Jersey. After graduating from the Rutgers University and working a few short-term IT jobs, he moved to the southern side of New Jersey just outside of Philadelphia, where he has been located for the last seven years.

In 2008, he started participating in the open source community. After working with Spring a bit here and there, he started investigating the Seam framework. After finding use of the framework in a few applications, he started participating more and more with the community.

Eventually, he became the module lead for a couple of components in Seam 3 and started working more and more with open source technologies. This led to more and more community involvement, including participation in the JMS 2.0 Expert Group.

After following through on some test-driven trends, he decided to try out a new framework called Arquillian to help automate some of the testing being performed on a few work-related projects. It surprisingly worked well, to the point of being used to perform all of the automated testing against container deployments of applications he was working on. This drove a lot of passion for the framework and his continued use of the tool today.

First, I would like to thank my mother Joan for all the years of her life she gave me; my father Jack, through the thick and thin he's been there; my sister Jenna; as well as all of my aunts, uncles, and cousins for being there.

I would like to thank my friends, who got to the point of saying "I'm worried" when I wasn't coming around as much.

Also, I want to thank the Arquillian community, in particular Hansty Bai and Jakub Narloch, the reviewers of the book, for all of the questions they received; my coworkers, past and present, for putting up with all of the things I come up with; and the team at Packt Publishing for supporting me.

A portion of the royalties will be donated to the *Johns Hopkins University Pancreas Multidisciplinary Cancer Team*.

About the Reviewers

Hantsy Bai is a self-employed freelancer and provides professional Java EE development and consulting service for customers around the world. In the past years, he has participated in many projects in different industries, including telecommunication, e-commerce, electric government management, office solutions for medium-sized and small enterprises, and so on.

He is also an open source practitioner and always likes to taste the new technologies. Working in a company and always keeping up with the popular technologies was impossible, so he quit and worked as a freelancer 4 years ago. This is when his field of vision became wider.

He has written a lot of blog entries about the popular and non-popular open source frameworks and tools. He is active in several Java communities in China, and also a troublemaker in the JBoss.org discussions and has posted many problems he encountered when he was using JBoss products in his projects. In 2012, he received the JBoss Community Recognition Award.

He likes traveling, climbing mountains, and currently lives in Guangzhou, China.

Jakub Narloch is a software developer with over 4 years of professional experience. He graduated in Computer Science at the Warsaw University of Technology with an MSc degree. Currently, he is working as a Software Engineer for an international company. During his career he has been mostly developing JEE and ASP .NET web applications. Besides that, he has a lot of interest in open source projects and contributed code to Arquillian and to other open source projects such as Spring OXM.

I would especially like to thank the Arquillian team and all contributors, because without their awesome work this book would never have been written.

Table of Contents

Preface **1**

Chapter 1: The Aliens Have Landed! **7**

The progression of testing **8**

What is Arquillian **9**

The Arquillian difference 10

The fundamentals of a test case 12

Testing profiles 16

Categorizing your test cases 17

Enriching your tests 19

Running out of the container 20

Efficient test authoring 21

ShrinkWrap – building your own app 23

Getting the most from Arquillian 25

Arquillian extensions 26

Summary **27**

Chapter 2: The Evolution of Testing **29**

How did we get here? **29**

J2EE, in the beginning 30

Manual testing patterns 30

XDoclet – the component revolution 32

A jump in technology **33**

Mocking 33

Still missing something 34

OpenEJB, the embedded EJB container 34

Servlet containers 35

Spring TestContext 36

Selenium – functional testing 36
QuickTest Professional 37
soapUI 38
Deployment 38
Summary 40

Chapter 3: Container Testing 41

Containers and deployments 41
Deployments 41
Protocols 42
The embedded container 44
Running the containers 44
Weld SE and EE 45
OpenWebBeans 46
OpenEJB 47
Jetty 48
Apache Tomcat 49
Mix and match patterns 50
Apache TomEE 52
GlassFish 3.1 54
JBoss AS 6.x 55
Honorable mentions 55
Embedded container summary 56
Managed containers 57
JBoss containers 58
GlassFish 3.1 59
Tomcat 60
Managed container summary 60
Remote containers 60
Tomcat 61
JBoss 61
GlassFish 62
WebLogic 62
Apache TomEE 63
Cloud deployments 64
Remote testing strategy 65
Remote testing use cases 65
Structuring the remote test case 66
Other remote cases 68
Remote testing tools 68
Deploying to multiple containers 68
The power of remote testing 69
Remote container summary 70
Container comparison 70
Tomcat 71

Apache TomEE 71
JBoss 72
GlassFish 73
Cloud containers 73
WebLogic 74
Other setups – the embedded applications 75
Unofficial containers 75
Summary 75
Chapter 4: Why Did the Test Fail? 77
Running your tests – the downside 77
The frustration 78
Packaging structure 78
The wrong type of archive 79
The container-specific descriptor 80
Wrong package version 82
The hard hat area 82
Sharing resources 82
Hello, are you listening? 83
Watch out for falling bugs 84
Problems starting up 84
The ports are working against you 84
You need a container 86
Consulting the remote server 88
Managing your test runtimes 89
Make it compile 89
Set up your profiles for long term 90
Overlapping profiles 91
The dos and don'ts 91
Embedded container dos and don'ts 91
Managed containers dos and don'ts 92
Remote container dos and don'ts 92
Summary 93
Chapter 5: Enriching the Enterprise Test Case 95
Testing the dependency injection 95
CDI injection – simplicity in action 97
Testing external extensions 102
Building and testing extensions 104
EJB and resource injection 106
Special considerations for EJB deployments 107
Testing persistence 107
Testing Session beans 112
Working with Remote Session Beans 114
Failure scenarios 116
Dependency failures 116

Introduction to Arquillian extensions **118**
Framework extensions 118
Spring 118
Arquillian versus Spring test 123
Summary **124**
Chapter 6: Arquillian Extensions **125**
Extending Arquillian **125**
Testing against a database using JPA **125**
The Persistence Extension 129
Preloading the database 130
Validate and clean up your database 132
Mix and match 134
The transaction extension **135**
Multiple artifacts **136**
Creating EARs 137
JaCoCo and Arquillian – code coverage 140
Deploying dependencies 141
Spock, very groovy test writing **145**
Summary **149**
Chapter 7: Functional Application Testing **151**
Functional testing **151**
Functional testing using Selenium 152
Consider the approach 155
Test Driven Development 156
Simplifying the test case 156
Page delegate pattern 157
Warp drive 158
Phases 161
Grouping requests 161
Spring MVC and Warp 163
Rich application testing with Graphene 168
Captures 172
Selectors and interactions 173
Summary **174**
Chapter 8: Service Testing **175**
Service testing approaches **175**
Web service stubs 176
Application invocation 177
Compare and contrast 178
Using soapUI 178
soapUI and Arquillian 178
soapUI and REST APIs 181

Other REST tests 183
HTTP client testing 184
Testing JMS 185
Testing Seam 2 188
Summary 189
Chapter 9: Arquillian and OSGi 191
OSGi and Arquillian 191
What is OSGi? 192
JBoss OSGi 192
OSGi versus everything else 192
OSGi versus JBoss Modules 193
Working with OSGi in a test case 193
ShrinkWrap Assets 194
OSGi archives 195
Testing a deployed service 196
Life outside of OSGi embedded 197
Testing Java EE using OSGi 197
Multiple deployment scenarios 199
OSGi support in GlassFish 204
Summary 205
Chapter 10: ShrinkWrap in Action 207
What is ShrinkWrap? 207
Creating archives 208
Adding resources 209
Assets 210
Managing libraries and modules 211
More on building EARs 211
ShrinkWrap resolver 212
Importing modules 213
Deployment descriptors 213
Importing descriptors 214
Deploying descriptors 216
A complex use case 216
Summary 218
Index 219

Preface

Automated testing has existed in some form for quite some time. Arquillian is a fairly new framework that has its roots around test-driven development and TCK validation for Java EE applications. Its use has grown into full-on application testing, supporting capabilities such as automatically seeding data to deploying to a large cluster of application servers.

Arquillian is a growing framework for automated testing of Java EE applications. *Arquillian Testing Guide* is meant to focus on how to use Arquillian to build automated test cases for your Java applications. We step through a large suite of the extensions available on top of Arquillian that you can add to your Maven-based projects to extend your testing capabilities.

This book starts with a quick drop into using Arquillian, and then navigates a little bit of testing history. We move on to Arquillian's container support and troubleshooting needs. We talk a lot about test enrichment strategies and capabilities that Arquillian adds to your test setup.

What this book covers

Chapter 1, *The Aliens Have Landed!*, helps us dive head first into Arquillian with some easy to read CDI examples. We go through the structure of an Arquillian enriched test case to understand what you'll see and where.

Chapter 2, *The Evolution of Testing*, shows that Arquillian has very deep roots in older test strategies and frameworks. We talk about some of these, how Arquillian grew from them, and how Arquillian can extend them.

Chapter 3, *Container Testing*, covers one of the key differences with Arquillian, its use of container deployments to execute tests. We go through the suite of these containers and how you may want to use them in your tests.

Chapter 4, *Why Did the Test Fail?*, explains how sometimes trying to figure out why something isn't passing can be difficult, even more when we have a full application server to deploy. This chapter will help you debug those tests.

Chapter 5, *Enriching the Enterprise Test Case*, goes through the built-in capabilities and when you can or cannot inject, as enrichment is how Arquillian provides injection support to your test case.

Chapter 6, *Arquillian Extensions*, builds upon the enrichment process; we look at some important extensions to Arquillian and alternative test runners.

Chapter 7, *Functional Application Testing*, focuses on the functional testing capabilities of Arquillian. We review Drone, Warp, and Graphene to show how to build out functional tests.

Chapter 8, *Service Testing*, focuses on service testing, such as EJBs, Soap Web Services, or REST APIs will be covered here.

Chapter 9, *Arquillian and OSGi*, focuses on how to use JBoss OSGi as your container runtime with Arquillian-based tests.

Chapter 10, *ShrinkWrap in Action*, illustrates how using Java code to create Java archives has never been as easy as when ShrinkWrap came to town.

What you need for this book

To run the examples in the book, the following software will be required:

- Maven:
    - Maven 3.0.3 or newer
- IDE: (pick one)
    - Eclipse with m2eclipse plugin
    - Netbeans
    - IntelliJIDEA
- Application servers:
    - Apache TomEE 1.5 or higher (preferably TomEE+)
    - JBoss AS 7.1.1 or higher

Sr no	Software Name	URL
1	Maven	`http://maven.apache.org/`
2	Eclipse	`http://www.eclipse.org`
3	Netbeans	`http://netbeans.org/`
4	IntelliJIDEA	`http://www.jetbrains.com/idea/`
5	TomEE	`http://tomee.apache.org/apache-tomee.html`
6	JBoss AS 7.1.1	`http://www.jboss.org/jbossas`
7	soapUI	`http://www.soapui.org`
8	Selenium	`http://www.seleniumhq.org`

Who this book is for

This book is focused on developers and testers alike. Developers will find this book useful if they have previously worked with JUnit, are familiar with Java EE application development, and are interested in learning more about building more in-depth test cases.

Testers will find this book useful if they have used tools such as Selenium or soapUI to test their applications in an automated fashion and want to try a new tool. You'll learn some more about how to automatically deploy the applications that you test and use some additional tools to test more robustly.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "The `@RunWith` annotation tells JUnit to use the Arquillian runner for running this class."

A block of code is set as follows:

```
@Test
public void testCalculationOfBusinessData() {
  CalculatorData cd = new CalculatorData(1, 3, 5);
  CalculatorService ms = new CalculatorServiceImpl();
  ms.calculateSum(cd);
  assertEquals(1 + 3 + 5, cd.getResult());
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
public class TestUtils {
  public static JavaArchive createBasicJar() {
    return ShrinkWrap.create(JavaArchive.class,"test.jar")
        .addAsManifestResource(EmptyAsset.INSTANCE,"beans.xml")
        .addPackages(false,getCorePackages());
  }
```

Any command-line input or output is written as follows:

```
mvn clean install -Popenwebbeans-embedded-1
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: " If you click on **Edit**, the selected row will be shown".

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
The Aliens Have Landed!

The goal of this book is to help understand how and where to use Arquillian with your enterprise applications. I will be providing an in-depth overview of Arquillian throughout this book, as well as overall strategies for how to add Arquillian to your standards at work.

This chapter is meant to get your feet wet. You will be writing tests with your code, and deploying using embedded Weld, OpenWebBeans, and Glassfish. In later chapters, we will use managed containers and remote containers on a wider scale.

Within this book, I make a few assumptions about you, the reader. First, I am assuming that you are familiar enough with most common open source application servers (Tomcat, JBoss, Glassfish) as well as CDI implementations (Weld, OpenWebBeans), and OpenEJB. In later chapters, I will discuss WebLogic and WebSphere testing. Second, I'm assuming you're already familiar with some of the core tools that Arquillian leverages—Maven and JUnit notably. When you download the source code for this chapter, you will see two versions, one a Maven project and the other an Ant project. This is the only chapter in which I provide Ant support, or at least will from the outset of the book (future source code releases may include Ant builds or even Gradle builds). Finally, I am assuming you are working with your favorite IDE. I don't plan on making anything that is specific to certain IDEs, so you should see everything behave in a cross-platform manner.

In this chapter we will cover

- The progression of testing
- What is Arquillian
- The Arquillian difference
- The fundamentals of a test case
- Testing profiles
- Categorizing your test cases
- Enriching your tests
- Arquillian extensions
- Running out of the container
- Authoring efficient tests
- ShrinkWrap — building your own app

The progression of testing

Way back when testing used to be primarily manual, test cases were created and executed by developers or quality assurance team members. These test cases would comprise of anything from simple unit tests (testing single methods or classes of code) or integration tests (testing multiple components within code) or even functional tests (tests that ensure the system performs as required). As we began to develop differently, whether it was from the agile project methodology or extreme programming methodologies, we needed more robust tools to support our needs.

This led to the advent of automated testing. Instead of a tester working with your application and running tests against it, they could simply press a few buttons, hit a few key strokes, and execute a 100 or 200 test case suite against your application to see the functionality. In some realms, something called a test harness was used. Test harnesses usually included running the compiled application in some kind of a sandbox environment that was probably something like production (this is after all the final binary that would be rolled out) that may or may not have pointed to a database (if it did, and it was smart, it probably pointed to a completely non-discreet database instance) to perform some level of testing. User input would be simulated and a report (possibly in some cryptic format that only few understood) would be generated indicating whether the application did what was expected or not.

Since then, new tools such as JUnit, Selenium, SoapUI to name a few, have been introduced to add more functionality to your test cases. These are meant to drive both unit and functional testing of your application. They are meant to be standard tools, easy to use and reuse, and overall a platform that many developers can work with, and can be a desirable skill set for employers. Standardizing of tools also allows for more integrations to occur; it may be difficult to get leverage to build an integration with your own built tools, with many developers wanting an integration with widely used frameworks A and B.

What is Arquillian

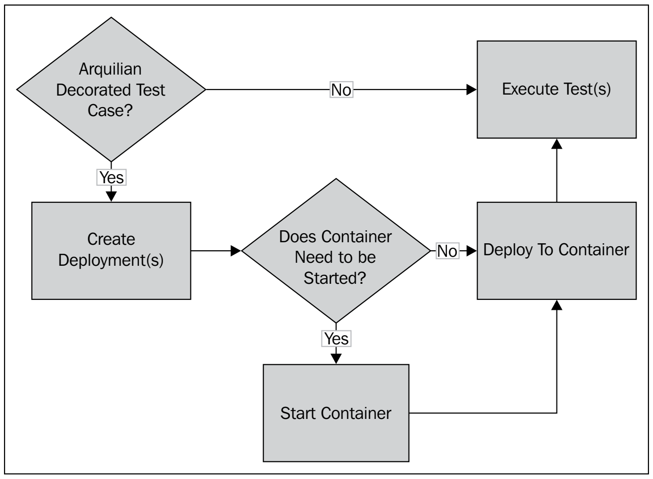
If you haven't heard of Arquillian before (or are very new to it), this may be the section for you. Arquillian is a testing framework for Java that leverages JUnit and TestNG to execute test cases against a Java container. The Arquillian framework is broken up into three major sections: test runners (JUnit or TestNG), containers (Weld, OpenWebBeans, Tomcat, Glassfish, and so on), and test enrichers (integration of your test case into the container that your code is running in). ShrinkWrap is an external dependency for you to use with Arquillian; they are almost sibling projects. ShrinkWrap helps you define your deployments, and your descriptors to be loaded to the Java container you are testing against.

For the sake of this book, the JUnit test container is used throughout. If you'd like to use TestNG with your code, you simply need to replace the JUnit container with the TestNG container, and have your test classes extend the `Arquillian` class found there. The JUnit test cases use a JUnit Runner to start Arquillian. The containers used will vary with each case. In the first few chapters, I introduce you to the basics of Arquillian, understanding how we came to in-container testing. I'll review with you the container options of Arquillian as well as the core Arquillian enrichers. Towards of the end of the book, the bulk of the test cases will focus on testing with JBoss AS 7.1 to show off the robust suite of tools. The version of the container used will likely be shown within the code; however you can usually take the latest 1.0.0.CRX, 1.0.0.BetaX, or 1.0.0.AlphaX of the code to use, in that order. They are typically compiled against the core Arquillian libraries that are current at the time of creation (as of writing, this is Arquillian 1.0.3). These containers will typically cover a range of releases of the application server under test.

The Arquillian difference

Arquillian can be considered a standardized test harness for JVM-based applications. It abstracts the container or application start-up logic away from your unit tests and instead drives a deployment runtime paradigm with your application, allowing you to deploy your program, both via command line and to a Java EE application server.

Arquillian allows you to deploy your application to your targeted runtime to execute test cases. Your targeted runtime can be an embedded application server (or series of libraries), a managed application server (where Arquillian performs the calls necessary to start and stop the JVM), or even a remote application server (which can be local to your machine, remote in your corporate infrastructure, or even the cloud).



Arquillian fits in to certain areas of testing, which can vary based on testing strategies for your application. If you are using Java EE 6, you may want to use an embedded CDI container (such as Weld) to unit test parts of your application. These tests could happen hourly or every time someone commits a code change. You could also use Arquillian to automate your integration test cases, where you use a managed or embedded application server to run your application, or even just parts of your application. You can even use Arquillian to perform automated acceptance testing of your application, using other tools such as Selenium to drive requests through the user interface. This can also be used to smoke test deployments of applications.

Arquillian can be used with a wide variety of containers that support everything from JBoss 4.2 (slightly pre-Java EE 5) through Java EE 6 and can control these containers in what Arquillian considers types – embedded, managed, and remote. Embedded application servers run within the same JVM as your test cases always do. Managed run within a separate JVM and are started and stopped by Arquillian. A managed container will start on the first test that requires a deployment and stop once all tests have been executed; there is no restart. Remote containers are as the name implies, remote JVMs. This could be on the same physical hardware that your test runs on or a remote piece of hardware that the application server runs on. The application server must be running in order to deploy. Note that if there is a problem deploying, such as the managed application server will not start or the remote application server will not start, Arquillian will fail once and assume that deployments will fail afterwards for the remaining test cases. A few important things to note about how to structure your application servers, which will be reviewed in depth as we review containers in *Chapter 3*, *Container Testing*:

- Do not mix your unit test application servers that are used for automated testing and those that you use for manual testing. Whether it's a separate instance required, a distinct domain, profile, whichever your application server vendor supports, avoid mixing them. One of your biggest blockers may be from your manually deployed application interfering with your automated testing application.
- Even though remote application servers can be physically separated from your testing, they typically require the binaries to be locally available. Plan to have a copy of your application server available on your CI server(s) for this purpose.
- Prepare your application for this kind of testing. Whether it's the automatic deployment or undeployment of resources (JMS queues, JDBC connections, users, and so on) or ensuring that the server is up and running (for example, prebuild, kick off a kill, and restart process) make sure this can all happen from your build, either in your CI server or using scripts within your source repository.
- Do not try to reuse application servers across applications. If two test cases are running in parallel, you can run into inconsistent results.

The fundamentals of a test case

As our software has evolved, our strategy for testing it must evolve as well. We have become more dependent on techniques such as dependency injection (or **inversion of control – IoC**). When we take this in to consideration, we realize that our testing has to change. Take a look at the following example:

```
@Test
public void testCalculationOfBusinessData() {
  CalculatorData cd = new CalculatorData(1, 3, 5);
  CalculatorService ms = new CalculatorServiceImpl();
  ms.calculateSum(cd);
  assertEquals(1 + 3 + 5, cd.getResult());
}
```

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

We can assume that the method `calculateSum` takes the `int` values passed in to `MyDataObject` and sums up the values. As a result, when I construct it using `1, 3, 5` the total should come out to `9`. This is a valid test case for our service layer, since our service layer knows what implementations exist out there and how they should be tested. If there was another implementation of `CalculatorService` that multiplied all results by 2, a separate test case or test class would exist which tested that object.

Let's say we look at the business layer that invokes this service object:

```
@Model
public class CalculatorController {
@Inject
private CalculatorService service;
  @Inject
  private CalculatorForm form;
  /**For the injected form, calculates the total of the input**/
public void sum() {
    CalculatorData data = new CalculatorData(form.getX(),form.
    getY(),form.getZ());
    service.calculateSum(data);
    form.setSum(data.getCalculatedResult());
  }
}
```

This example uses JSR-330 annotations to inject references to `CalculatorService`, a service layer object that can perform basic calculator functions and `CalculatorForm`, some sort of UI component that has form input and output that can be read or returned. If we want to test this class, we will immediately run into a problem. Any invocation of the sum method outside of a JSR-330 (dependency injection for Java) container will result in a `NullPointerException`.

So what does Arquillian do to make our test case more legitimate? Let's take a look at the test case and review the anatomy to understand that better:

```
@RunWith(Arquillian.class)
public class CalculatorTest {
  @Deployment
  public static JavaArchive createArchive() {
    return ShrinkWrap.create(JavaArchive.class,"foo.jar")
        .addAsManifestResource(EmptyAsset.INSTANCE,"beans.xml")
        .addPackage(CalculatorData.class.getPackage());
  }

  @Inject CalculatorForm form;
  @Inject CalculatorController controller;

  @Test
  public void testInjectedCalculator() {
    form.setX(1);
    form.setY(3);
    form.setZ(5);
    controller.sum();
    assertEquals(9,form.getSum());
  }
}
```

There are a few pieces that make up this test case, each of which we'll need to review. These are given as follows:

- The `@RunWith` annotation: It tells JUnit to use the Arquillian runner for running this class.
- The `@Deployment` annotation: It tells Arquillian to use the specified archive for deployment purposes and testing purposes.

- The injection points: In this case, CDI injection points represent the objects under test.
- The Test: This is where we process the actual test case. Using the injected objects, we simulate form input by inserting values in X, Y, and Z in the form, then invoke the controller's `sum` method, which would be called from your user interface. We then validate that the resulting sum matches our expectations.

What we gained was leveraging Arquillian to perform the same IoC injection that we would expect to see in our application. In addition, we have the following dependencies within our Maven `pom` file:

```
<dependencyManagement>
<dependencies>
  <dependency>
  <groupId>org.jboss.shrinkwrap.resolver</groupId>
  <artifactId>shrinkwrap-resolver-bom</artifactId>
  <version>2.0.0-alpha-1</version>
  <scope>import</scope>
  <type>pom</type>
    </dependency>
    <dependency>
  <groupId>org.jboss.arquillian</groupId>
  <artifactId>arquillian-bom</artifactId>
  <version>${org.arquillian.bom.version}</version>
  <scope>import</scope>
  <type>pom</type>
    </dependency>
</dependencies>
</dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.10</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.jboss.spec</groupId>
      <artifactId>jboss-javaee-6.0</artifactId>
      <version>1.0.0.Final</version>
      <type>pom</type>
      <scope>provided</scope>
    </dependency>
    <dependency>
```

```
      <groupId>org.jboss.shrinkwrap.resolver</groupId>
      <artifactId>shrinkwrap-resolver-impl-maven</artifactId>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.jboss.arquillian.junit</groupId>
      <artifactId>arquillian-junit-container</artifactId>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.jboss.arquillian.container</groupId>
      <artifactId>arquillian-weld-ee-embedded-1.1</artifactId>
      <version>1.0.0.CR3</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.jboss.weld</groupId>
      <artifactId>weld-core</artifactId>
      <version>1.1.8.Final</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-simple</artifactId>
      <version>1.6.4</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
```

Our dependencies are as follows:

- **Arquillian BOM**: It's an overall list of all artifacts for running Arquillian
- **ShrinkWrap**: It's a separate API for creating deployments
- **JUnit and Arquillian JUnit Container**: Arquillian support for working with JUnit
- **Arquillian Weld EE**: Arquillian containers control what your deployment runtime should be

Testing profiles

One of the benefits of using Arquillian with Maven is that you can control how your tests execute using Maven profiles. With this approach, you can define a profile per container to test against. In some cases, you may want to test your application or library against multiple containers. This allows you to reuse deployments against different libraries. We still remain with the core limitation: you can only have one container on your classpath at a time. Let's suppose we take this same application, but want to run it against Apache OpenWebBeans as well as Weld.

Each Arquillian container has a default Maven profile configuration that can be used for testing. These containers are covered within the Arquillian documentation, found at `https://docs.jboss.org/author/display/ARQ/Container+adapters`.

Steps to try out containers are given as follows:

1. Import the configuration defined by the container definition.
2. Run a mvn clean install `Pconfiguration_name`.
3. You can choose to set a default profile as well if you like. If you don't choose a default profile, you will need to specify one every time you want to run tests.

Running the tests for this project, for both the `weld-ee-embedded-1.1` container and the `openwebbeans-embedded-1` profile should result in the same thing – a working test suite that is valid in both implementations.

At the time of writing, I used Weld 1.1.8.Final and OpenWebBeans 1.1.3.

It is important to point out at this time that these profile names are only useful if your application is designed purely for cross-platform testing and you want to run all test cases against each individual platform. If your application only targets a single platform, you may want to derive test cases that run on that platform as well as any subcomponents of that platform (for example, if you are a WebSphere v8 user, you may want your unit tests against OpenWebBeans and integration against WebSphere; however, if you are a WebLogic user, you would want to use Weld and WebLogic for your testing).

Typically, when it comes to testing, you will use a distinct Maven profile to cover your stages of testing. You should set up a default Maven profile that runs only your basic tests (your unit tests); this will be set as `activeByDefault`. This should include any testing dependencies needed to run only these unit tests. You may optionally choose to only run certain parts of your test suite, which could be distinct packages under `src/test/java` or even standalone projects that are children to your parent that are only run under certain circumstances. I prefer the former approach, since the usage of conditional child projects can become confusing for developers.

Profiles are useful for conditional dependencies, since they do include a dependency and `dependencyManagement` section in their `pom` files. You can also avoid dependency leaking. For example, most applications require the use of the full Java EE 6 APIs, but including these APIs with your Glassfish build will cause your tests to fail. Likewise, deploying to a managed application server may require different APIs than deploying to a remote application server.

Categorizing your test cases

One thing that Arquillian ultimately derives is that names mean everything. There are two naming schemes that you should follow always, they relate to the questions "what" and "how". What components are you testing? What phase of testing are you under? How does this class relate to your test cases? How is this code being deployed?

These questions really pop up due to the nature of Arquillian, and really show off its robustness. Considering some of the main testing phases, unit test, integration test, system test, compatibility test, smoke test, acceptance test, performance test, usability test should all relate to specific packages in your test code. Here's a proposed naming scheme. I will assume that your code starts with `com.mycompany.fooapp` where `com.mycompany` is your company's domain, `fooapp` is the name of your application. You may have packages below this such as `com.mycompany.fooapp.model` or `com.mycompany.fooapp.controller` or even `com.mycompany.fooapp.modulea.controller` all of which represent varying testing scenarios that you may consider.

- `com.mycompany.fooapp.test`: This is the parent package for all test classes. There may or may not be any classes in this package.
- `com.mycompany.fooapp.test.utils`: This is where all of your test utility code goes. This would be where any deployments are actually build, but invoked from your test classes.

- `com.mycompany.fooapp.test.unit`: This is where all unit tests should exist. The packages/classes under test should fall relative under here. For example, `com.mycompany.fooapp.test.unit.controller` should test your controller logic.
- `com.mycompany.fooapp.test.integration`: Likewise, all integration test cases should fall under here.

Following this pattern, you can derive your functional, acceptance, usability, and so on test case names.

Following this pattern, you can easily define Maven profiles within your projects to test out your various layers. Let's suppose you want to define a `unittest` profile where the build includes running all unit tests (usually light weight tests, that maybe use an embedded container with Arquillian), you could do something like this:

```
<profile>
  <activation>
    <activeByDefault>true</activeByDefault>
  </activation>
  <id>unittest</id>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-surefire-plugin</artifactId>
        <configuration>
          <includes>
            <include>**/unit/**</include>
          </includes>
        </configuration>
      </plugin>
    </plugins>
  </build>
</profile>
```

This will tell Maven to only run the tests in the unit test packages by default. This allows your builds to go by quickly and test key components of your code when needed. Since it's active by default, you would get these tests run anytime you kick off `mvn install` or `mvn test` from the command line, or your continuous integration server.

Likewise, you may want to run more tests during your integration testing or system testing phases. These may or may not overlap with one another, but would likely include your unit tests as well. You could use the following, very similar Maven profile to achieve that:

```
<profile>
  <id>integrationtest</id>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-surefire-plugin</artifactId>
        <configuration>
          <includes>
            <include>**/unit/**</include>
            <include>**/integration/**</include>
            <include>**/system/**</include>
            <include>**/regression/**</include>
          </includes>
        </configuration>
      </plugin>
    </plugins>
  </build>
</profile>
```

With the way this profile is configured, any time you invoke mvn test –`integrationtest` you will run all unit, integration, system, and regression test cases. These tests will probably take a while. If you're lucky and your application server supports an embedded mode it probably has a quick start up/shutdown. These tests may be running against a remote container though, so the deployment and execution time may take longer.

Enriching your tests

One of the core principles noted in the demo test case is that we injected a reference to the business object being tested, which itself had injection points as well that were satisfied. There are three core injection points supported by default with Arquillian, though others are supported by extensions as well as some custom injection points defined by Arquillian itself. They are as follows:

- `@Resource`: It defines references to any JNDI entry as per standard naming and injection schemes.

- `@EJB`: As long as you are including a deployed EJB in your archive, you can refer to it as long as your target container includes EJB support. EJB JNDI entry conventions remain intact. It is also worthwhile to note that Local and Remote interfaces matter. When you deploy to a same JVM application server, or a remote application server you must ensure that your injected test cases can see the EJB reference appropriately.
- `@Inject`: This is available as long as you have deployed a valid bean archive (requires `beans.xml` in META-INF or WEB-INF).

Running out of the container

Sometimes you will need to mix your Arquillian tests with non Arquillian tests, or sometimes you have special test cases that require a deployment but should not be run with that deployment (simulating a remote client to that deployment, for example). This may include a set of unit tests that have deployments to the container and some that do not require this. You can run the test outside of the container JVM by using the `@RunAsClient` annotation. This can be done at the class level or at the method level, allowing you to have multiple run-types within a single test class.

This approach would be useful if you want to test a web service client, either SOAP or a REST API, ensuring that you are executing outside of the container JVM for your client. One of the custom injection points that Arquillian supports is a `@ArquillianResource` URL `baseUrl`; this represents the base URL of a remote container deployment that you can use in your client-level tests. The `ArquillianResource` annotation supports an additional value attribute that can be the class of a Servlet deployed in case you have multiple deployments occurring as a part of your test. Once you have the URL for your application deployment, you can build the location of your web services to be able to use your client application for testing purposes. This would also be used if you wanted to functionally test your application via HTTP, perhaps using Selenium as the driver for the application.

The other types of objects available for injection using this annotation are the Deployer used and the InitialContext of the remote application server. The deployer gives you access to the underlying deployment while InitialContext would allow you to look up remote objects – for example, EJBs, JMS Queues/Topics/ConnectionFactories, or any other remotely accessible resource.

Efficient test authoring

Now that we have a test that can run against multiple containers, we need to start adding more test cases to our suite. When we do this, we have to keep two key elements in mind. Don't repeat yourself, and don't bloat your software. Remember that in this chapter, we're sticking with simple deployments to show off a lot of Arquillian's capabilities and will go into more robust containers later on.

In many of our applications, we have a number of components that make up various layers. Some of them are dependent on one another, others are more generic. One of the key things to think about when planning your Arquillian testing is how should your JAR files look. Let's suppose we have an entire data object layer that has dependencies all throughout your application. We must have those classes in every test case. However, we can usually skip to only certain controllers and business beans within their specific test cases. Remember to create utilities to define your object structure, this gives you a single entry point for creating your deployment archive and allows for better extensibility. Here is a prototype class that can be used to start, it supports creating both a JAR file as well as a full web application:

```
public class TestUtils {
  public static JavaArchive createBasicJar() {
    return ShrinkWrap.create(JavaArchive.class,"test.jar")
        .addAsManifestResource(EmptyAsset.INSTANCE,"beans.xml")
        .addPackages(false,getCorePackages());
  }
  public static WebArchive createWebApp() {
    return ShrinkWrap.create(WebArchive.class,"test.war")
        .addAsWebInfResource(EmptyAsset.INSTANCE,"beans.xml")
        .addPackages(false,getCorePackages());
  }
  public static Package[] getCorePackages() {
    return new Package[]{CalculatorData.class.getPackage()};
  }
}
```

What these methods add is a significant reduction in code that is impacted by a package change, or code refactoring in your primary code base. Let's suppose that we want to extend `getCorePackages` to also take in a number of classes, which when added, add the entire package to the deployment:

```
public static Package[] getCorePackages(Class<?>...classes) {
  List<Package> packages = new ArrayList<Package>();
  if(classes != null) {
    for(Class<?> c : classes) {
      packages.add(c.getPackage());
    }
  }
  packages.add(CalculatorData.class.getPackage());
  return packages.toArray(new Package[]{});
}
```

One benefit that we have using this approach is that anyone who was using `getCorePackages()` does not need to change their code, since the argument is not required.

Note that the `ShrinkWrap` API has several `addPackage`/`addPackages` methods. This one used has a first argument, Boolean, whether to re-curse through child packages to find your code. Going back a few pages to some naming conventions I proposed, what would happen if you add the package `com.mycompany.fooapp` to your bundle? All of your application classes, including test classes, would be added to the deployment you are creating. This is probably not what you would have expected, as a result, the recommendation is to not re-curse into child packages and instead just list out each package you want added explicitly.

Another option to consider is to have your test classes delegate their work. Commonly thought of as the façade programming paradigm, you can actually apply this to your test code as well. If you have code that should be tested distinctly but use different deployments you may want to use distinct deployment objects but reuse your test case. This may involve using a controller type test case that simply delegates its test calls to another object that is meant to purely handle the testing logic. Your controller would have methods annotated `@Test` and include your `@Deployment`(s) but would delegate logic to another class, potentially an injected test executor class.

ShrinkWrap – building your own app

One of the more curious things about Arquillian is that your test case is responsible for constructing the application to be deployed. When you're working with Maven it is especially odd, since all of the information to build the application is there, either implicitly based on the project structure or explicitly listed in your `pom` file. There are two realizations around this that are important:

- Your test cases are meant to test anything from a subset of your code to your entire application. Arquillian is flexible enough to handle both extremes.
- Arquillian works great with Maven, but also works with other build tools such as Ant and Gradle.

In order to support the dynamic generation of your application, the ShrinkWrap project exists to help dynamically build Java archive files. There are four primary archive types supported in ShrinkWrap: Java Archive (plain JAR files), Web Archive (WAR files), Enterprise Archive (EAR files), and Resource Adapters (RARs). Your Arquillian test case can declare any number of these archive files to be created for testing purposes.

Another place that ShrinkWrap helps with is the creation of deployment descriptors. These could be `application.xml` files, or `persistence.xml` files, or any of the standard deployment descriptors that you would use in your application. Likewise, it has extensibility built in to allow the creation of new descriptors in a programmatic fashion.

This book assumes that you are using the ShrinkWrap 2.0 APIs; one of the key features added is support for resolving dependency files from reading a Maven `pom` file. Another key thing you need to do is modify your test classpath to include some files from your core application. Here is an example of what to do, from another project I was working on recently:

```
<testResources>
<testResource>
  <directory>src/test/resources</directory>
</testResource>
<testResource>
  <directory>src/main/resources</directory>
</testResource>
<testResource>
  <directory>src/main/webapp/WEB-INF</directory>
  <targetPath>web</targetPath>
</testResource>
</testResources>
```

This will make it easier to reference your main application resources without requiring full paths or file manipulation.

ShrinkWrap also provides ways to build deployment descriptors. This is a programmatic approach to adding the descriptor to your deployment, including programmatically creating the descriptor. Because of the need to test using the same descriptors being built with the production application, I have found it easier to reference to an existing descriptor. However, in some cases it may make more sense to use one customized to your test applications. In this scenario, I would strongly recommend creating a utility method to build the descriptor.

To do this, we will add the following to our Maven `pom.xml` in the `dependencyManagement` section:

```
<dependency>
  <groupId>org.jboss.shrinkwrap.descriptors</groupId>
  <artifactId>shrinkwrap-descriptors-bom</artifactId>
  <version>2.0.0-alpha-4</version>
  <scope>import</scope>
  <type>pom</type>
</dependency>
```

Declare the following dependency:

```
<dependency>
  <groupId>org.jboss.shrinkwrap.descriptors</groupId>
  <artifactId>shrinkwrap-descriptors-impl-javaee</artifactId>
</dependency>
```

Then add the following method to our code:

```
public static StringAsset createWebXml() {
  return new StringAsset(
      Descriptors.create(WebAppDescriptor.class)
      .exportAsString());
}
public static WebArchive createWebApp() {
  return ShrinkWrap.create(WebArchive.class,"test.war")
      .addAsWebInfResource(EmptyAsset.INSTANCE,"beans.xml")
      .addAsWebInfResource(createWebXml(), "web.xml")
      .addPackages(false,getCorePackages());
}
```

This will generate the `web.xml` file expected for your test case. One of the benefits of using the descriptors is that we can actually import the one from our application and make changes as needed.

```
public static StringAsset createWebXml() {
  WebAppDescriptor descriptor = Descriptors.
  importAs(WebAppDescriptor.class)
      .fromFile(new File("web/web.xml"));
  descriptor.createWelcomeFileList().welcomeFile("someFile.jsp");
  return new StringAsset(descriptor.exportAsString());
}
```

As a result, this will use your base `web.xml` file but change the welcome file to something else. You can do this with a number of other descriptors, including `persistence.xml`, `beans.xml`, `web-fragment.xml`, and so on.

Getting the most from Arquillian

Arquillian does take some understanding to work with. In order to get the most from it, you have to work with it and invest time.

Your standard rules still apply. Arquillian does not use any special rules when it comes to processing deployments. The rules about deployment descriptors, archive contents, and so on and so forth still apply. You need to keep the rule of thumb – if it deploys to your application server, then you can deploy the same archive via Arquillian; just make sure that you are deploying the same archive.

Note that you can use the Archive's `toString` method to print out the contents when in doubt. This supports a formatter as well, to make it easier to read the contents.

Alternatively, you can export an Archive using `archive.as(ZipExporter.class).exportTo(File)` if you want to manually review the file.

Run as many tests as you can with Arquillian. Due to Arquillian's nature, you're going to start to find inconsistencies in your code if you don't test it with Arquillian. This could include unexpected dependency injection expectations, which Arquillian will process for you. Arquillian, since it executes your code the way the application server would as requests come in, makes your tests more consistent with the real world of how the code works. Testing more in Arquillian, even if it is just using a basic CDI container or OpenEJB container, will allow you to test more effectively. You make the best use of Arquillian when you use it throughout 100 percent of your test cases.

Finally, my last key advice to getting the most from Arquillian is to remember to not over-complicate your build to make use of Arquillian. Arquillian works fine as a part of your standard build. It has no requirement to create distinct projects for each build type or make overly complicated build structures on top of your application. If you are attempting to run Arquillian with your continuous integration server, then you must ensure that different test cases run either as different steps of the build or as separate build jobs.

Arquillian extensions

There are a number of extensions available for Arquillian; they are designed to extend Arquillian functionality to do some domain-specific testing:

- Persistence, using DBUnit and validating that results of interacting with the database. This is covered in-depth in *Chapter 8*, *Service Testing*.
- REST, invoke REST APIs from Arquillian the were deployed as a part of the test case, which will be covered in *Chapter 6*, *Arquillian Extensions*.
- Spring, use a Spring Context and additional Spring libraries with your test cases. Spring will be covered in *Chapter 5*, *Enriching the Enterprise Test Case*, as a part of test case enrichment and Spring MVC support will be reviewed in *Chapter 7*, *Functional Application Testing*, with Warp.
- Drone/Selenium, functionally test your web applications using Arquillian. This is covered in *Chapter 7*, *Functional Application Testing*. In addition to this, when we review Warp, a way to assert on both the client and server side in that chapter.

Summary

My main goal in this chapter was to give you an overview of Arquillian and introduce you to some of its core concepts. I will be starting this book with how Arquillian grows over its predecessor frameworks to give you more application integration.

The book will continue with more strategies around testing, validating code coverage, and expanding what you can test in an automated fashion now that you have access to Arquillian. We will spend a few chapters reviewing how Arquillian works with application servers, including how to target new application servers. There is an entire chapter dedicated to debugging problems with your tests as well.

Finally, we wrap up with a strong focus on extensions and how they can help you test your code more thoroughly in an automated fashion. Next, we'll begin to review how exactly the community came to building Arquillian and what tools we've used in the past. We'll start to see that some of these tools are still useful with Arquillian and how mixing mocking and containers can give us some useful results.

2
The Evolution of Testing

Testing has grown much over time. While the previous chapter gave you a brief introduction to Arquillian as a framework, this chapter will focus on the technologies that have existed for many years to support our testing efforts. In some cases, these are introductions to technologies that Arquillian can extend and others that Arquillian may compete with.

How did we get here?

It's sometimes important to think about the road travelled before we start investigating something new. The goal of this chapter is to review some of the important technologies we've all worked with in the past, how they've grown over time, and how we can leverage them, either from the technologies we've worked with or the ones we are still using, to build new and more robust platforms. These technologies include:

- J2EE
- XDoclet
- Mocking, Mockito, JMock
- OpenEJB
- Servlet containers
- Spring
- Selenium
- soapUI

J2EE, in the beginning

The platform that we all know and love today, **Java 2, Enterprise Edition**, is the basis for everything we deal with today. Originally a group of specifications that were later unified and ultimately recreated as JSRs, they defined the beginning for everything we deal with today – Servlets; EJBs were the beginning points of this group. Now we will define all the levels of components of interactions from a client to a server-level program. Back then (the late 90's), we probably didn't focus much on automated testing. In a new landscape of tight deadlines, large project teams, and many levels of managers on top, we need ways to simplify our processes and deliver higher quality than ever done before.

Manual testing patterns

Possibly around the longest, manual testing is the cornerstone to all test cases. Manual testing has been the type of testing used since the first software was written. You write your code, you build it, and then run it to see if it does what you expected. This approach is used by developers and testers alike to test code after it's built.

Manual testing typically involves a working user interface (UI). For your application this might include a Swing-based UI, a set of REST calls, a SOAP-based web service, or even a socket server. This can have a long lead time before you start being able to test anything. Also, consider if your application is broken down into various levels of programming logic. Your data access objects and entities, for example, are a separate set of JARs from your main code. Since they are separate JARs, they should be considered a separately testable component compared to the main application logic. How would you test these JARs? It's probably not something your quality assurance team would test by themselves; they work on the functionality of the application not the individual technical components.

Manual testing might also include (in the case of Java) creating a `main` method to run and test the execution of a certain code. I consider this case to be the same as if you wrote a JUnit test, but then erased the test completely. This type of testing is hard to reproduce and can end up unnoticed when it breaks. It becomes apparent in this example. Let's suppose we have the `convertToCelsius` method that takes a temperature in Fahrenheit and converts it:

```
public static double convertToCelsius(double farenheit) {
  return (farenheit - 32) * (5/9);
}
```

The manual testing approach has us creating a `main` method that can run and print the output. We can then verify the output as follows:

```
public static void main(String[] args) {
  System.out.println(convertToCelsius(32)); //this should print 0.
}
```

While not a difficult test to run, it does demonstrate the point very clearly. Let's look at the JUnit equivalent:

```
@Test
public void testConvertToCelsius() {
  Assert.assertEquals(0,Converter.convertToCelsius(32));
}
```

In the JUnit version, we have an assertion that the value is `0`. Now, granted, we could have done in the `main` method something more like this to make it clearer:

```
public static void main(String[] args) {
  assert 0 == convertToCelsius(32);
}
```

However, I'll assume that if you didn't start off with JUnit you weren't interested in having an automatic validation. Now there are a lot of reasons you could have ended up with a piece of code like this (and I've actually seen some well-known companies ship software with these sitting around). The first is that your project structure blocks it. JUnit, for example, has an odd license and in some cases it has decided to not leverage JUnit due to its license. You may use Ant as the Ant scripts are not set up to run JUnit. Can you add it? Depends on your role on the team.

There's also the problem of readability with this approach. If you use a JUnit test, it will tell you the expected and the actual values on failure. You can customize it further with more information. Adding `System.out` to your Java code requires some modifications for it to become readable.

There is likely no removing manual testing for functional applications. Manual testing should be considered when performing user acceptance testing, where you need to validate that the application functions the way you would expect end to end. From a quality assurance standpoint, it's the most direct way to make sure an application works the way expected. A developer though shouldn't wait until this point to begin testing if they want to consistently deliver quality software across any tier of application code.

XDoclet – the component revolution

Dating as far back as 2002, Java developers have always wanted a solution for a simplified programming model for EJBs. **XDoclet** was one tool used heavily to help resolve this issue, allowing you to build a single class to support your entire component in EJB. Back then, EJBs were required to have standalone `Home` classes (which were required to get references to the EJBs) as well as the actual EJB implementation. This, as you may guess, was quite a hassle. Back then, we didn't have generics to allow us to simplify and reuse these objects; you needed one for every EJB you were working with. What XDoclet was doing was performing the development technique of "code generation" wherein additional source code was generated based on the XDoclet comments.

Since XDoclet essentially created a POJO for your component, this allowed you to simply construct your XDoclet annotated class (It is worth noting that XDoclet was used prior to Java 5, so its annotations were actually Javadoc comments.) inside a unit test to execute your test cases. Therefore, your unit test case for what would become an EJB could look as simple as this:

```
public class XDocProcessorTest extends TestCase{
  public void testProcessMessage() {
    String msg = "do some work.";
    XDocProcessorBean bean = new XDocProcessorBean();
    bean.processMessage(msg);
  }
}
```

Similar to basic JUnit test cases, you simply need to instantiate your bean and test against it. This provides a simple, straight-to-the-business logic approach to building your code. You can also assure yourself that you are working only with the business logic in your code with this approach to EJB generation. So why am I talking about XDoclet? Since XDoclet simplified the development, it should also simplify the testing right? Unfortunately not, because we still need things such as database connections that are still not available, which limits what we can and cannot test. One major downside was that we needed the EJB libraries on our classpath to test against, since the object structure needed them for consistency.

A jump in technology

If we fast forward a little bit (pretty much skip the rest of J2EE progression) and land in Java EE 5, we run into an interesting paradigm shift. EJB3 came out with this revision of Enterprise Edition, with a lot of cool new features. It was meant to be a simplification release, removing the need for a lot of extra interfaces, or classes implementing methods to do their work. The component programming model instead moved towards annotations (in general, annotations are considered decorations on your code, providing metadata for some runtime to use. In Java, these are `CLASS`, `RUNTIME`, and `SOURCE`. `RUNTIME` allows you to access the annotation after compilation while `CLASS` and `SOURCE` are typically only compile time).

Since we switched to annotations, we no longer needed the EJB container libraries on our classpath to test. The annotation would be ignored, as long as it is not used, by the testing runtime. You can more easily test these objects, as long as they are coded correctly. Setters are available to inject resources that focus on testing methods that do not use the injected members.

Mocking

Around this time, we started experimenting with object mocking: substituting objects for mocks, or fakes of those objects. This is done through method substitution where you take a reference to a mocked object, be it some data object or some component you interact with, you define what the expected results are for calling certain methods, and then define your test case around this built-up mock object.

For the sake of this section, I am referring primarily to Mockito. One of the curious things you'll discover with object mocking is that you are in full control of what your object does at all time. This makes your mocked object behave the way you would expect but still gives your application code everything it needs to operate.

I don't necessarily consider mocking to be something separate from Arquillian. For use inside of unit testing, it may make sense to pair up object mocking with an embedded CDI container to avoid database interaction. This will allow your CDI-based controllers to fluidly interact with a "database tier" that can be verified later on – how many times was the save method called, was the right object passed, and so forth. One antipattern to consider is using mocked objects as CDI-produced objects. First, we'll add the following dependency to our project:

```
<dependency>
  <groupId>org.mockito</groupId>
  <artifactId>mockito-all</artifactId>
  <version>1.9.5</version>
  <scope>test</scope>
</dependency>
```

Next, consider the following producer class:

```
@Produces
public CalculatorForm createMockedForm() {
  CalculatorForm form = mock(CalculatorForm.class);
  verify(form,times(1)).setSum(9);
  return form;
}
```

You'll see that we're able to provide a producer method of the object that creates the mock, which we can then inject and interact with. We must be careful to not include the real producer, since it will generate deployment errors, as this approach assumes that you use a producer method to create your objects. While this is a clever approach to testing, it does require certain behavior within your code. You may want to consider not mocking your injected members and as a result only inject the objects that they integrate with.

Still missing something

One of the important notes to make with each of these frameworks discussed is that we're never really testing the code we wrote in the way it's truly to be executed. While mocking is very powerful, it's still only used in a very controlled scenario. This makes the mocking scenario good for controlled testing (unit testing) but not anything beyond. When we start to move towards integration testing, we will want to ensure that our code is tested against something more concrete, something more like what we're running against.

OpenEJB, the embedded EJB container

OpenEJB was born in 1999, and represented a purely standalone EJB container, not a part of an application server (this was pretty much the norm, since EJB 1 was simply a technology, not a part of an application server). Since then, OpenEJB has remained a standalone EJB container that could be plugged into a number of web containers or application servers to provide EJB support. It is able to handle a bootable container, pointing to resources on the filesystem with application code that can start it from a CLI or main program.

Since OpenEJB is bootable, it allows you to start up the platform within a test case. Since it's using a started container, you can reference EJBs within its context to perform operations. We have finally succeeded in finding a way to test our components in an automated fashion similar to how they are really run.

There's a caveat though. Not every container is the same; there are bugs or misconceptions about various specifications that can throw off the behavior of your application. Also, since OpenEJB is embedded within other application servers, there could be issues with the integration (or if you were very revolutionary, you wrote a command-line program that started OpenEJB in embedded mode). As a result, you do not have a completely perfect environment.

In some cases, this may be enough (for unit testing purposes perhaps). However, fully automated testing in other realms may require more valid testing (right application server, right OS, and so on). There are very similar use cases to OpenEJB now with embedded containers. Jetty can be started automatically using a Maven plugin and then followed up with manual test cases against that Jetty container. JBoss also provides some levels of an embedded container, which are covered in the next chapter.

Servlet containers

If you are like most application developers, your application deploys as a simple war file that includes most, if not all, of your dependencies wrapped together. You may or may not use components such as JNDI or container-managed database connections. You may do some, if not all, of this yourself within your application and simply need a vehicle to automate the deployment and test execution.

It is possible to start most servlet containers automatically from a build. This can be accomplished with a Maven plugin in most cases but with a little bit of work can be done from Ant scripts as well. If you are using a Continuous Integration server, most of them have the ability to automatically deploy after a build. The difference with Arquillian is that the deployment of your code to the application server is managed within the framework. You have the option of building a custom deployment or using ShrinkWrap to import the deployment from a file that was built. In most cases, Arquillian can also start your container for you. With some containers you can even get to the point of downloading a dependency that is your application server from Maven and extracting it prior to a run. One of the clear differences, which will be shown in a few chapters, is Arquillian's support for a combined client and server validation model. Since you are deploying just your application to the server, there is no chance to add in a server-side validation, and instead you must rely on client-side validation only.

Spring TestContext

If you are a Java developer, you have likely heard of SpringFramework, designed to do dependency injection into your applications and overall provide a simplified programming paradigm. While Arquillian provides some Spring integration, Spring also has support for a testing framework. Similar to Arquillian, this is done on top of JUnit, leveraging the `@RunWith` annotation to call Spring. It handles tasks such as bootstrapping your context file(s) to automate tests and handle object injection. This support also existed against JUnit 3 but has since been deprecated.

One of the key differences, even when looking at Spring MVC support, is that the only case supported natively is mocked HTTP requests. If your beans provide configuration to talk to a database, it will work, but there is no concept of a deployment to an application server. Requests are simple mock objects that provide feedback to the Spring IoC container to perform certain actions.

This would be similar to an approach where we used Arquillian to bootstrap a CDI container and used mock objects to stub out certain behaviors. Being a more complete framework, Spring's approach may be a more stable approach to testing. This is where the limitations come in. Since Arquillian is a third-party neutral framework, it can build extensions that work with other tools more easily. Spring Test is obviously tightly coupled with using its own framework to perform work. You obviously wouldn't see an approach to Spring Test that allowed you to test your EJBs; it only tests your Spring beans. As a result, it has a much narrower scope.

The major limitation you'll notice with Spring Test is that you're never actually deploying to an application server. As a result, your tests are never actually against your application running in say Tomcat or WebLogic. The configurability of your test cases are also limited as such. If you use a JNDI connection in production or test but your unit tests are run using an embedded connection (directly specify username, password, JDBC URL) then you do have some minor discrepancies in your tests.

Selenium – functional testing

One of the newer tools out there is **Selenium**, a functional website testing tool. Selenium assumes that you have an already deployed application and are creating test cases to verify its contents. Selenium is platform non-specific; it can test Ruby on Rails as well as JSP and PHP. Selenium test cases are written using an IDE, usually Firefox recording test steps. There are driver APIs available as well that can be used within a unit test (for example, JUnit) to be executed in an automated fashion.

Arquillian provides support for Selenium in the Drone and Warp extensions. Drone provides a cleaner way to inject references to a Selenium WebDriver instance. Warp allows you to build tests that cross both client and server, using tools such as Selenium to drive a client. These will allow you to deploy the application under test and then execute functional test cases via Selenium. As a standalone tool, Selenium may work well for websites. For web applications it may provide an additional level of automation, but you may want to think whether it makes sense or not for you to deploy some or all of your application via Selenium. You may want to do something along the lines of creating a small set of Selenium test cases to smoke test your application after a deployment, to test production, and verify that certain fields are present or that the application started up appropriately. These may be broken out into a separate profile that may include the word "smoke" in the package to indicate that it was run.

If you do want to mix Arquillian and Selenium together (other than receiving an odd-sounding science experiment), you will be able to test components of your application in an automated manner. The components could be anything from a couple of small screens through the entire application. The more of this you do, the more maintenance that needs to be done. Any time a screen changes you must update your tests to ensure that the changes are reflected properly. Similar to a unit test where the expected processing changes, if your user interface changes your test cases become stale and need to be fixed.

QuickTest Professional

Another approach for functional-level testing is a product from HP, **QuickTest Professional** (**QTP**). A widely used product that has been around for some time, it helps to automate the testing of applications. It supports more platforms natively than you would see from Arquillian. Arquillian is based around Java applications but QTP can test a wider variety of applications.

QTP doesn't perform any deployment for you. It's also based around some obsolete technologies, including VBScript (which, from the .NET perspective has largely been replaced by VB.NET for the server and JavaScript on the client side). QTP can do database validation as well as client-based validation but it cannot perform checks against data in memory. More so, it cannot leverage your APIs for reading a NoSQL database. One advantage QTP has is that you can test your thick client applications with it. Arquillian is limited to server-based code, or at least is conceptually bound to server-based code.

soapUI

Similar to the case of Selenium, **soapUI** provides a toolset for creating tests and then executing them. Similar to Arquillian, these cases can be integrated with your testing structure; JUnit and Maven are also supported. There is even a dedicated Maven plugin for executing soapUI and loadUI. While these tools are more directed at functional testing, the soapUI suite is directed at application integration, whereas Selenium is directed towards web application programming. soapUI would be used to test your JMS queues, SOAP or REST web services, or other integration points.

Similar limitations exist when using soapUI. There is no deployment protocol; this is the key thing that Arquillian seems to have leveraged against all competition. soapUI assumes that the application you are testing against is a running application and not something freshly deployed.

One common feature with these tools is that if you could deploy to a container, perhaps using a manual Maven deployment (via Tomcat plugin, or WebLogic plugin, or Jetty), you could then run these cases. This, however, forces the use of Maven.

Deployment

The one key piece missing from all of the mentioned tools is a common deployment platform. In OpenEJB, you could write code in your tests to start up the OpenEJB container, deploy your EJBs, and then perform operations on them. It will work, but if you ever change from OpenEJB to something else you'll need to rewrite the code to work with the new technology in your tests. If we want to ensure that everything is working fine from before to after, we know that code has to change. If we want to deploy to Jetty we have to make certain assumptions about our code and the directory structure included. Tomcat embedded can also be started programmatically, but will need some integration code as well. If you want to do both Tomcat and OpenEJB you will need a lot of integration code.

Arquillian solves this deployment problem through deployment configuration. Based on the classpath entries, deployments will be created and run against the application server. Since it's based on classpath entries, you do have to ensure that separate containers are physically segregated (two containers cannot appear on the classpath). Arquillian also supports a configuration file, `arquillian.xml`, that contains some container-specific configuration for your application. This may involve pointing to an environment variable for a location of an application server, or some username/password configuration.

Arquillian expects test cases to declare a deployment, a method that creates some kind of Java Archive file that contains classes or resource files that can be deployed to the application server. These deployment methods have a specific annotation as well as a particular method signature:

```
@Deployment
public static Archive<?> createJavaArchive()
```

We use static methods because this is run in the `@BeforeClass` section of a JUnit test. Deployments can have the following attributes:

- `name` (String): This attribute is a unique name for the deployment. It is primarily used when there are multiple deployments in your test case.
- `managed` (Boolean): This attribute when true, says that Arquillian will do the deployment. You can also define cases where Arquillian does not do the deployment.
- `order` (int): This attribute is also used when there are multiple deployments and it controls in what order the deployments will occur.
- `testable` (true): This attribute indicates whether the deployment can be wrapped in the Arquillian protocol or not.

In order to work with named deployments, you can use the `@OperateOnDeployment` annotation to determine which test cases run with which deployment archive. You can also use `@TargetsContainer` on a deployment to run that specific deployment on a specific container.

As it stands, Arquillian is the first testing framework that allows deployment of code to occur during your testing phase as well as to support injection methodologies into your components. This allows Arquillian to be a much more robust tool to support many kinds of testing.

Many of the Arquillian extensions leverage this point in driving home how they work. In the case of Selenium, Arquillian provides the needed deployment capabilities. Similar usage could occur with Arquillian and soapUI/loadUI. In other cases, Arquillian simply uses the container deployment model to load your resources to an application server (such as Tomcat or Jetty plugins for Arquillian) and then deploys your application to them, separating your code from the container. The Spring extension makes use of annotations similar to the Spring annotations, but also combines them with a deployment phase. Another key point is that Arquillian does not replace the tools you may already be used to, but simply extends them or simplifies their usage to make them more robust. Obviously being a Java tool though, Arquillian does limit itself to only the Java realm.

Summary

While not covering every single framework around, this chapter was meant to give a little bit of history as well as a few highlights about testing frameworks. This history is meant to give you a background on where Arquillian came from and the need to create a stable and reusable testing infrastructure that can be extended to support many different test cases.

As shown here, your application can be in many shapes or sizes but still use Arquillian to test your services or components. As we will begin to explore, the range of what Arquillian can work with is quite wide and grows more as contributors help add functionality. An important point to raise is that you must remember to use the right tool in your arsenal. Arquillian will help with your deployment and automated testing, but if you just want to use automated tests you should consider your strategy overall.

A part of what makes Arquillian powerful is that it allows your objects to be integrated with your container. Your test case becomes part of your deployment with the container (if needed) and integrates with the container. This process of enrichment will be the focus of the upcoming chapter.

3
Container Testing

Arquillian is all about testing your code inside a container. There are three primary ways that Arquillian can interact with a container – embedded, managed, and remote. This chapter is broken up based on the three types of containers supported in Arquillian. The first are **embedded containers,** which typically run on the same JVM as your test case. Next are **managed containers,** which Arquillian will start up for you during the test process and shut down after the tests are run but run in a different JVM. Finally, there are **remote containers,** which are assumed to be running prior to the test and will simply have deployments sent and tests executed.

Containers and deployments

Containers represent what your application will run on. This is the code that represents what you would typically run on top of the application server. The deployments though are what represent your code as standard Java Archives – JAR, WAR, EAR, RAR—that can be deployed to an application server to run code.

At the start of this, I want to remind you to consult with the Arquillian Reference Guide for the latest container setup. The list of containers can be found at `https://docs.jboss.org/author/display/ARQ/Container+adapters`.

Deployments

This section is meant to be a brief crash course in using ShrinkWrap to create archives. The ShrinkWrap API is designed via Factory and Builder patterns. There is a utility class – `ShrinkWrap` – that acts as the entrance for the factory of archives. Calling `ShrinkWrap.create(JavaArchive.class);` will generate a new Java Archive (representing a JAR file).

Arquillian uses a static method defined in your test case to create the archive. The method signature roughly is `public static Archive<?> createTestArchive()`.

This method must be annotated `@Deployment` – this will tell Arquillian to find the archive and execute it to create the test content. Each of these archive types also supports different kinds of content. `JavaArchive` support classes embedded, whereas `EnterpriseArchive` and `WebArchive` support child archives (for example, add a `JavaArchive` to your `WebArchive` or add a `WebArchive` to your `EnterpriseArchive`).

As mentioned in *Chapter 1, The Aliens Have Landed!*, when you work with ShrinkWrap, you have to manually build your archives. For a very simple archive that contains a few classes and maybe some manifest entries, this is simple. However, for a more complex archive where you essentially want to reuse your built archive for testing purposes, you should consider `ZipImporter`. The syntax is straightforward: `ShrinkWrap.createFromZipFile(JavaArchive.class, new File("target/myarchive.jar"))`. This does require you to have access to the file, but will import the contents of the file.

Protocols

Arquillian uses protocols to capture results of the running test cases. The choice of the protocol does not impact how your deployment runs, other than the use of `web-fragment.xml` files being deployed with your code (in case you deploy a WAR file instead of a JAR file).

There are the four primary protocols:

- **Local**: This assumes a test case is running in the same JVM and as a result only works with embedded containers. This is also the default protocol if you are working with an embedded container (most of the time).
- **Servlet 2.5**: A protocol that leverages Servlet 2.5 functionality to pass the test results back to the caller. This is the default protocol for containers that are pre-Java EE 6. There are certain deployment requirements when using Servlet 2.5, typically including that you create a WAR deployment.
- **Servlet 3.0**: A protocol that leverages Servlet 3.0 functionality to pass the test results back to the caller. This is the default protocol for containers in the Java EE 6 realm (including Tomcat 7, when running Managed or Remote Containers).
- **JMX**: This is only supported on JBoss AS 7. It allows your test to run on the application server but uses JMX to execute and report back.

When you are working with embedded containers, Arquillian supports a local API for test execution. This makes sense when you are working in the same JVM. Arquillian is just there and can issue commands to make a test run. However, in remote containers both "managed" and "remote" are included. You have kicked the build off from your local Maven instance but the test case itself ends up running on the container. As you also may have noticed, certain protocols behave differently from others. When reviewing the JBoss AS 7 containers, I noted that by default they use JMX to run test cases. In order to ensure everything works right for CDI, you will need an HTTP request driving it instead.

This occurs because the source of the request happens differently. Some of the other protocols supported include Servlet 2.5/3.0 as well as local. JMX is a protocol that was built for JBoss AS 7. As a result, it is meant to test within the JBoss AS 7 stack, where Arquillian is used quite heavily. The JMX protocol as a result does not generate an HTTP request. If you are used to using CDI's `@RequestScoped` then this will be active in a HTTP request but not in a new JMX command. This is similar to OpenWebBeans versus Weld in the embedded containers. In Weld EE, the example works correctly in both Dependent and RequestScoped. In Weld SE and OpenWebBeans, the example only works in Dependent. Weld EE simply starts up the HTTP request for you.

Arquillian's protocol framework is extensible. That is how the additional protocols were created – Servlet 2.5, Servlet 3.0, and JMX. In addition to this, the OSGi protocol (part of the Arquillian OSGi suite) is implemented by extending the JMX protocol to also work with OSGi.

Depending on how your application is deployed, you may or may not be able to support additional protocols. Servlet 3.0 is the default for Java EE 6 containers (save JBoss AS 7) and Servlet 2.5 is the default for Java EE 5 containers. If you are using an embedded container, you will likely use the local protocol. You may not have the ability to substitute another protocol. Weld itself does not handle JMX, and does not have a native Servlet container so it cannot support a servlet-based protocol. If you were, however, using Jetty as your container, depending on which version you use, you can use different protocols. Jetty 8, since it works with Java EE 6/Servlet 3.0, can use Servlet 2.5, Servlet 3.0, and Local (since it's embedded) as its protocols. The older versions of Jetty can use Servlet 2.5 and Local.

Protocols aren't something you will typically deal with when working on your day-to-day projects. Understanding them though may help you understand why certain test cases pass on one container but not another. It may also give you some ideas on how to mix them up with your containers to get more realistic results.

If you do want to create your own protocol though, you would simply need to register it as an extension and implement the interface protocol. Your extension should then be registered in `META-INF/services/org.jboss.arquillian.core.spi.LoadableExtension`.

The embedded container

Embedded containers come in a number of varieties, ranging from simple containers such as Weld, OpenEJB, and OpenWebBeans to full-blown application servers such as GlassFish. Within the embedded containers we will review how to work with them effectively.

Running the containers

Each of the containers has its own configuration contained within `arquillian.xml`. When you download the sample code, you will notice that there is not much in here. This is because typically little is required to run with `arquillian.xml`.

Since we haven't dug into it much yet, it's important to understand the use of this file. This file should be located at the root of your test classpath, usually as a resource (when using Maven) under `src/test/resources`. Each of the containers supports some level of configuration, which should be placed into this file. The basic skeleton of this file includes:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<arquillian xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns="http://jboss.org/schema/arquillian"
   xsi:schemaLocation="http://jboss.org/schema/arquillian
   http://jboss.org/schema/arquillian/arquillian_1_0.xsd">
</arquillian>
```

From here, you can add containers to your XML document based on qualifiers. One of the important things to note is that the examples accompanying this chapter use package-level filtering to run test cases. Since the containers listed here are either CDI-or servlet-based, we will validate that the CDI components or servlets are running as expected. Another use for this file is to manage extension configuration. This will be discussed in more depth over the next few chapters of the book.

Weld SE and EE

These two containers are very similar, but cover different use cases. The core difference is that the SE container does not include EE APIs, whereas the EE container supports mocked EE APIs. You will need to add the EE APIs to your classpath for test execution. The easiest ones to use are the JBoss-provided APIs in a Maven `pom`. Here's the typical import statement:

```
<dependency>
<groupId>org.jboss.spec</groupId>
  <artifactId>jboss-javaee-6.0</artifactId>
    <version>1.0.0.Final</version>
  <type>pom</type>
  <scope>provided</scope>
</dependency>
```

In this example, I'm using them as `provided`. If you don't need them for compilation purposes you can switch the scope to `test`. You would use this dependency anytime you need Java EE 6 APIs, so feel free to replace this with your standard Java EE API dependency.

Even though you may be working with the EE container, you should not expect transactions to be present and only EJBs that are exposed via no-interface views can be resolved. Everything acts like a ManagedBean in the container, with CDI support for injection of your resources where appropriate.

Weld SE is the right container for you if you are working on a standalone CDI application. Weld EE is the right container for you if you are working on an application being deployed to an application server.

The standalone Weld containers are what you would want to use if you are running unit tests. They will not contain transactions or anything from a full Java EE container. One other important aspect to remember is that you are dealing with SE objects. There is no "RequestScope" that is active. As a result, any object bound to the request scope will not work for injection. To work around this, you can make your objects dependent instead of RequestScoped.

Let's say that we create the following CDI bean:

```
public class HelloWorld {
  private final String text = "Hello, World!";
  public String getText() {
    return text;
  }
}
```

And then add the following test case for it:

```
@RunWith(Arquillian.class)
public class HelloCDITest {
  @Deployment
  public static Archive<?> createTestArchive() {
    return ShrinkWrap.create(JavaArchive.class).
    addClass(HelloWorld.class)
        .addAsManifestResource(EmptyAsset.INSTANCE, "beans.xml");
  }

  @Inject
  BeanManager beanManager;

  @Inject
  HelloWorld helloWorld;

  @Test
  public void testCdiBootstrap() {
    assertNotNull(beanManager);
    assertFalse(beanManager.getBeans(BeanManager.class)
    .isEmpty());
    assertEquals("Hello, World!", helloWorld.getText());
  }
}
```

When running this test case, in either Weld SE or Weld EE you'll see that the result is what you expected – fully passing tests. From the sample code, you can either use `mvn clean install -Pweld-se-embedded-11` or `mvn clean install -Pweld-ee-embedded-11`.

OpenWebBeans

Everything stated for the Weld containers also applies to the OpenWebBeans container. It will behave like the SE container outside of a servlet container or other application server. The capabilities of the containers are very similar and you should see similar test results when running on OWB instead of Weld.

Later in this chapter, I describe a few extensions using Weld. A lot of this applies to OWB as well; however, for brevity they are not deeply discussed in the book. If you look at the example code though, you will see examples using both.

For this example, you can reuse the existing CDI bean for your OpenWebBeans test. It will work just fine. Just run the following command to execute the OWB version of the test cases:

```
mvn clean install -Popenwebbeans-embedded-1
```

OpenEJB

The situation with OpenEJB is very similar to Weld and OpenWebBeans. This is an embedded container, however, it does support the necessary transaction state expected in an EJB runtime.

Assuming you have appropriate services deployed, it is just as easy to define an EJB test case as it is to define a CDI test case. In fact, the source code example simply takes the CDI bean we've been testing and changes it to become an EJB.

So far we've spoken about CDI test cases. It's important that you remember this when you work on building your test harnesses using Arquillian. If your code is using CDI but you don't want to test against CDI (at least in this profile), you need to ensure that you have CDI APIs available to compile against, otherwise you'll get the ominous `"package javax.enterprise.inject does not exist"` compilation error. Similarly, we'll be adding servlets to our test cases later on. We want to test them against Jetty and Tomcat, but in order to ensure they compile, the servlet API must always be on the classpath to pass the compilation. We don't want to dynamically change source files for each profile, only the test cases that execute. This is a factor when trying to debug. We'll review more issues like this in *Chapter 4, Why Did the Test Fail?*

We'll review it more in depth later on as well, but you can add CDI support to your EJB container easily. To build this profile, take the existing OpenEJB profile and give it a new name (copy it). Add the same dependency management section from OpenWebBeans and paste it under this. Then, add the OpenWebBeans dependencies as well as the `openwebbeans-openejb` dependency. Since OpenWebBeans is the controlling side of this container, it will be the deployment we use for Arquillian. Switch the OpenEJB deployment with the OpenWebBeans deployment.

Since OpenEJB supports EJBs, we want to test it out using an EJB. Therefore, assume the following EJB:

```
@Stateless
@LocalBean
public class HelloLocalBean {
  private final String text = "Hello, World!";

  public String getText() {
    return text;
  }
}
```

We can test it using the OpenEJB container based on the following test case:

```
@RunWith(Arquillian.class)
public class HelloLocalBeanTest {
  @Deployment
  public static Archive<?> createTestArchive() {
    return ShrinkWrap.create(JavaArchive.class)
        .addClass(HelloLocalBean.class)
        .addAsManifestResource(EmptyAsset.INSTANCE,
         "ejb-jar.xml");
  }
  @EJB
  HelloLocalBean helloWorld;
  @Test
  public void testEJBBootstrap() {
    assertNotNull(helloWorld);
    assertEquals("Hello, World!", helloWorld.getText());
  }
}
```

We can then run this test case (from the sample code) using:

```
mvn clean install –Popenejb-embedded-3.1
```

The EJB test case should have worked in the CDI containers as well. This was not demonstrated simply to ensure that the EJB example was run within an EJB container.

Jetty

Jetty 6.1 and Jetty 7 are supported by Arquillian (as well as Jetty 8). Since Jetty migrated from Mortbay to Eclipse, the package names changed and that is the key difference between the suites. These act completely as servlet containers only. Using the Arquillian Jetty container is a lot like using the `mvn jetty:run` command from the command line. The main difference is that a distinct WAR file is generated, based on your deployment method usage, for your test cases.

As a result of these only being servlet containers, unless you mix in additional container functionality (such as Weld or OpenEJB) it is not advisable to test component levels here. It may be more practical to use these containers to simply test your web tier, perhaps a web service implementation on top of Jetty (RESTEasy or Jersey, for example) rather than testing out your application logic. You do gain a few things such as simplified deployment processes by leveraging the `arquillian.xml` file for some level of configuration.

Jetty, Weld, OpenEJB and OpenWebBeans are only available as embedded deployments. They cannot be run as managed or remote containers.

Apache Tomcat

Tomcat is a tried and true pure servlet container. Both Tomcat 6 and Tomcat 7 support embedded modes where the ZIP for the application server is downloaded and extracted into a temporary location, then booted from the JVM.

Since Tomcat is just a servlet container, its behavior and usage will be very similar to Jetty. Since this is a servlet container, you should only target WAR files for deployment here. You cannot deploy a JAR file directly to Tomcat; Arquillian will create a wrapper WAR file for you.

Since Tomcat is simply a servlet container, the target tests should be around servlet testing. You can extend it further to test things such as JAX-RS or JAX-WS runtimes, as long as they are embedded into your WAR file. This approach will be reviewed in depth in a later chapter on service and remote testing.

If we want, we can unit test a servlet using Tomcat. Assume the following servlet (can you guess what it does?):

```
@WebServlet(name="HelloServlet",urlPatterns={"/hello"})
public class HelloServlet extends HttpServlet {
  @Override
  protected void doGet(HttpServletRequest req,
  HttpServletResponse resp)
      throws ServletException, IOException {
    PrintWriter out = resp.getWriter();
    out.println("Hello, World!");
  }
}
```

When we want to create a WAR for this servlet attempting to test it using Tomcat embedded, we create the following test case. It opens an HTTP connection and reads the response from the servlet back:

```
@RunWith(Arquillian.class)
public class HelloServletTest {
  @Deployment
  public static Archive<?> createTestArchive() {
    WebArchive wa = ShrinkWrap.create(WebArchive.class,"test.war")
        .addClass(HelloServlet.class)
        .addAsWebInfResource("web.xml")
    return wa;
  }

  @Test
  public void testGetText() throws Exception {
    URL url = new URL("http://localhost:9090/test/hello");
    InputStream is = url.openStream();
    BufferedReader br = new BufferedReader(new
    InputStreamReader(is));
    String result = br.readLine();
    String expected = "Hello, World!";
    assertEquals(expected,result);
  }
}
```

For the benefit of Tomcat, this is all you need. If you want you can also use this test case with Jetty, however using it with Jetty will need the `jetty-env.xml` and `jetty-web.xml` (both are listed in the test case) files. You can run your servlet test case in the sample code by running:

```
mvn clean install –Ptomcat-embedded
```

Mix and match patterns

Did you know that you can mix containers together to generate more robust test cases?

Let's say you are building a simple web application; by default it runs on WebLogic and uses CDI. You can build a lightweight test harness for the application by embedding Weld in your test package and deploying that to a Jetty container. You can do something very similar. If you want to test EJB and CDI together, you can deploy Weld with your OpenEJB container to create test cases that work with EJBs and CDI, but don't require a full container deployment.

To do this, all you need to do is add your CDI runtime to your test classpath. Assuming that you are using Maven and the ShrinkWrap Maven Resolver, the following will work for you:

```
MavenDependencyResolver resolver =
  DependencyResolvers.use(MavenDependencyResolver.class)
  .loadMetadataFromPom("pom.xml");

return ShrinkWrap
  .create(WebArchive.class, "test.war")
  .addClasses(listYourApplicationClasses())
  .addAsLibraries(
  resolver.artifact("org.jboss.weld:weld-core-bom")
  .resolveAsFiles());
```

This can apply to any of the partial containers that support deployment to occur. For example, you cannot create a WAR file that targets the CDI containers and expect web requests to be answers. This isn't bootstrapped in the CDI container. You can however enable CDI requests in a servlet container (Tomcat, Jetty) as long as you deploy a sufficiently configured `web.xml` that refers to the components necessary to activate the CDI runtime.

To activate Weld in Tomcat, you will need the following two files:

- The `META-INF/context.xml` file needs to have necessary configuration for an object factory given as follows:

  ```
  <Context>
     <Resource name="BeanManager"
        auth="Container"
        type="javax.enterprise.inject.spi.BeanManager"
        factory="org.jboss.weld.resources.
        ManagerObjectFactory"/>
  </Context>
  ```

- The `WEB-INF/web.xml` file must contain the JNDI location for the type:

  ```
  <resource-env-ref>
     <resource-env-ref-name>BeanManager
     </resource-env-ref-name>
     <resource-env-ref-type>
        javax.enterprise.inject.spi.BeanManager
     </resource-env-ref-type>
  </resource-env-ref>
  ```

Using these two together, along with having the CDI runtime embedded in your WAR file, will activate CDI capabilities in your deployed Tomcat application (and test cases). In addition, you should refer to the Weld Servlet Listener, also in your `web.xml`:

```
<listener>
  <listener-class>org.jboss.weld.environment.servlet.
  Listener</listener-class>
</listener>
```

This will ensure that your application starts properly with CDI capabilities native. Of course, the user of either of these requires you to have deployed a WAR file (of course, you could always deploy `web-fragment.xml` if you are running Tomcat 7). When you do this step though, you should be careful to remember that you are now running Tomcat as your container and Weld simply becomes an add-on to your archive. You must make sure you deploy a WAR file to Tomcat as a result.

This generally also works with Jetty integration. Making changes to your `web.xml` is generally safe and can be considered cross-platform compliant.

Once you have created your mixed container environment, you can then start to mix your unit tests together. For example, create a Maven profile that runs both OpenWebBeans and OpenEJB and performs both the CDI and EJB test cases.

Apache TomEE

For those unaware, TomEE is a newer project from the combined teams of OpenEJB and Tomcat designed to provide a lightweight Java EE compliant application server. It targets the Java EE 6 web profile; however, it also provides a plus profile that includes JAX-RS, JAX-WS, JMS, and Connector Architecture. Both types of containers are usable within Arquillian. Support for Apache TomEE is provided by the OpenEJB team. This is one of the few cases where an Arquillian adapter is created by a team other than the JBoss group.

Using the embedded container, it will actually boot TomEE within the test case. Once you have Arquillian configured (using JUnit or TestNG), you can add the Arquillian dependency for TomEE to your build:

```
<dependency>
  <groupId>org.apache.openejb</groupId>
  <artifactId>arquillian-tomee-embedded</artifactId>
  <version>${tomee.version}</version>
  <scope>provided</scope>
</dependency>
```

You'll use the provided scope, rather than test scope, if you need to use TomEE's APIs for compilation purposes. This may work better for you if the only goal you have is an overall clean `pom.xml`.

As a result, TomEE integration works a little different than standard Arquillian containers. While the `arquillian.xml` file is still used for standard Arquillian support, it is not recommended for configuring your container. Instead, you should use system properties configured in your Surefire plugin in Maven:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-surefire-plugin</artifactId>
  <configuration>
    <systemPropertyVariables>
      <tomee.httpPort>0</tomee.httpPort>
      <tomee.stopPort>0</tomee.stopPort>
    </systemPropertyVariables>
  </configuration>
</plugin>
```

The code snippet we just saw would be equivalent to the following one being in your `arquillian.xml`:

```
<container qualifier="tomee">
    <configuration>
        <property name="httpPort">0</property>
        <property name="stopPort">0</property>
    </configuration>
</container>
```

Since this is a standard Java EE container (web profile), you can deploy EAR, WAR, and JAR files to the container, so you can have any of them as your deployment object. You'll note that for the sake of the code examples I have set the port to 9090; this is because the configuration for ports is container specific. Later in this chapter when we go over remote container testing, I will show how to get a direct connection to the deployment when running with remote test cases.

Configuration information for TomEE is documented at `http://tomee.apache.org/arquillian-available-adapters.html` including the Maven profiles.

One thing to point out is that you will likely receive deployment exceptions while working with TomEE. This can happen when you have a failed deployment that doesn't get fully cleaned up. As a result, you may need to manually delete files. Your tests will still work. The main cause of this is if you do not provide the system arguments for TomEE. You can run the TomEE version of the tests by running the following:

```
mvn clean install –Ptomee-embedded-1.5
```

Another important piece of information to point out is that because of the development for TomEE being provided by Apache, an Arquillian container TCK has been founded to be able to validate that all containers operate as expected.

GlassFish 3.1

The GlassFish embedded container is a "onejar" dependency that will include the entire application server; all packages typically deployed, and all APIs. Since it includes all APIs, you need to ensure that you aren't cross compiling with another set of APIs (make sure the JBoss APIs aren't being compiled against). Running tests against the GlassFish embedded container will download your container for you completely (it's about 45 MB) during the test compile phase of your build.

The only special configuration needed is to set the HTTP listening port. This is to support the servlet test cases. This is meant to simply keep it consistent with the other servlet containers.

```
<container qualifier="glassfish">
   <configuration>
     <property name="bindHttpPort">9090</property>
   </configuration>
</container>
```

Now the standard servlet request will work fine. You'll see that since GlassFish is a full container, we are able to run our full suite of test cases – CDI, regular JUnit, EJB, and servlet. You can run the GlassFish set of tests by using the following from the command line:

```
mvn clean install –Pglassfish-embedded-3.1
```

JBoss AS 6.x

JBoss AS 6 is a considerably different setup for embedded containers as compared to the rest of the pack. It is the only embedded container that requires you to have installed JBoss AS 6 outside of Maven. With every other container for embedded mode we have reviewed, they have been installed with your code, downloaded from the Internet, and run in a standalone, invisible to the user way. Another curious change with the JBoss AS 6 embedded container is that you have to manually define your JVM. This makes it considerably harder to test with, since you need to ensure that your JVM is consistent on both developer workstations as well as your continuous integration server.

To simplify your work, you'll want to ensure that everyone has an environment variable defined called `$JBOSS6_HOME`. It will be used in the code example as well. Assuming you're using JBoss 6.0.0 (and not JBoss 6.1) and you extracted the application server to `C:\jboss\jboss-6.0.0-Final`, then that would be your `$JBOSS6_HOME` (assuming that there is a bin directory right below this, you didn't create any additional directories). A second environment variable will be used, `$JBOSS6_HOME`, to represent the profile you want to use – all, default, and so on. You should strongly consider creating a dedicated profile for your unit testing. This will help you alleviate any problems running your application locally instead of running it in your test cases. To create a separate profile, you just need to copy the source directory. If you want your new profile to be based on "default," simply copy default to a new name. Ironically, there is nothing else required to set up the profile; it's that simple.

Honorable mentions

I decided to create a section for configurations you may be curious about at this point, but I'm not sure if this is the right place for them. For the most part, they are covered later on in other sections.

Arquillian Daemon Server

This container is a brand new offering for those who want to test using Arquillian but don't actually target a container. Your code is just standalone. The Arquillian Daemon Server simply allows you to take your code, test case included, and run a standalone JVM to execute your test cases. This server simply runs and executes your test case. There is no web container or anything. The server can be backed by anything, but the current implementation is a Netty server that can work with the Arquillian Wire Protocol.

Spring

I can't think of a single developer I know that isn't aware of Spring. Spring is the tried and true IoC framework. Spring was not brought up in this chapter yet since it's not a standalone deployable container. You can easily add Spring support to your test cases using an Arquillian extension but you need to still use a true container, for example, Tomcat or Weld. The main difference is that Weld has a bootable runtime, whereas Spring does not. If you are using the Tomcat Server, you can refer to it as a Tomcat instance in Managed and Remote modes.

JPA/Hibernate

Similar to the issue with Spring, neither JPA nor Hibernate is a standalone container. It is possible to start up JPA or Hibernate from within a test case but it is not possible to start a server for JPA or Hibernate. The main issue is that there is no native injection capabilities to use when starting up JPA or Hibernate. The same is true for other JPA implementations – EclipseLink, OpenJPA, and so on.

Embedded container summary

As you can see, you have a wide variety of tools available when you test using an embedded container. Embedded containers can operate in a single JVM with your test case but can suffer from some classloader isolation issues. These are really good solutions if you want to test more thoroughly with your CDI-based applications or your EJB-based applications. It may not be feasible for you to start up every possible service during your unit testing phase; for example, is it necessary to make an HTTP request to a servlet during your tests? Can you just inject members and work with them for your unit testing? It's simply a decision around needs and goals for your test strategy. My recommendation is to remain with simply unit tests; it's a strong factor if your unit testing phase takes less than ten minutes to run, but should be expected to run in under five minutes. You may not be able to do that with every container configuration (in fact, running the example code on my high-end laptop takes about three minutes; and the largest test suite only has four cases, where most of the time is taken up starting up a GlassFish instance).

Integration testing is a completely different paradigm. When you are running integration tests, the system under test is specific for this case. Whether it's an automated function, an off-hours function, or a manually run function, it's important to remember this difference.

One interesting issue we started seeing with managed containers is that they do require a full install of the container locally for your tests to execute. As we move in to remote containers, we'll see a few interesting side effects. First, remote containers can be run on the same machine or different machine(s) than where the test is running. Second is that in most cases you still need a local install of your application server available. This is to ensure that any client libraries are available for use in the deployment to the remote container.

Managed containers

Managed containers are generally very different from embedded containers. Embedded containers are typically run inside the same JVM as your existing test case (and thus no classloader isolation). Managed containers are run outside of the test case's JVM, but are started by the test case. Typically, you will only start the managed container once and during the testing process multiple archives will be deployed and undeployed from the server.

Managed containers require a non-local protocol for Arquillian to work with. They are Java EE 5 compliant, they will usually use Servlet 2.5. If they are Java EE 6 compliant, they will use Servlet 3.0.

Since managed containers boot the container during the test, it requires the container to be installed (separate from your test cases) on the machine where your tests are running. Typically, environment variables will control which to use and which paths to work with.

In an ideal world, you would configure each of your containers to run in a distinct configuration for your specific build. If you are on a continuous integration server, this may just require some additional environment settings to be created for your job only. It may also need some unique configuration files if needed, based on your container's needs.

If embedded containers are designed for unit testing, then a managed container will be best for automated integration testing. You should expect that you are typically deploying at least a WAR file to your container, if not an EAR file. Since they are run in separate JVMs, they are ideal for ensuring that you do not have any of the classloader isolation issues that you may have with embedded containers. Since the container is managed, and you have access to its configuration, you can do things such as setting up datasources to be run against in your testing.

If you do not create a WAR file in your deployment, Arquillian will enhance your deployment with a wrapper WAR file; this is for its protocol support. If you do supply a WAR file, Arquillian's libraries will be added to it for the deployment. If you are running a Java EE 5 container, then your `web.xml` will be rewritten to support the Arquillian protocol servlet in the deployment. This will allow the actual communication. This is more or less true with any container that is Java EE 5 (or below). In Java EE 6, Arquillian relies on a `web-fragment.xml` to be included (in its libraries) that will be added to your application. This is one of the enhancements in Servlet 3.0, and thus shows how certain aspects of other tools (JAX-RS, JSF) are embedded within applications without needing to explicitly specify entries in your WAR file's `web.xml`.

When reviewing the sample code, please note that only the Maven profile information is available. Individual `arquillian.xml` configuration will need to be handled on an adhoc basis.

JBoss containers

There are several iterations of JBoss containers that are supported by Arquillian. They operate in all runtime modes, but their usability in each may be up to your scenarios. As noted, I would not recommend a full-blown application server for unit testing but it may be useful for integration testing.

JBoss AS 7.1

Similar to the setup for TomEE, JBoss AS 7 (and 7.1/EAP 6) support is provided by the JBoss AS team. As a result, certain problems can exist when using JBoss AS 7 for testing compared to other containers. The first issue is the default protocol. AS 7 uses JMX for its test execution protocol. If you are expecting an HTTP request to initiate a CDI scope, this will not work for you, as it's different than typical processing. You can set the default protocol to override this – in this case you would want Servlet 3.0 as your protocol. This will ensure that your tests are running more like user requests are driving your processes.

By default, each of these containers will look for a predefined `JBOSS_HOME` environment variable on your system. When found, this will be your root of JBoss AS 7. I strongly recommend renaming this to `$JBOSS7_HOME` or `$JBOSS71_HOME`. This can be done by using an `env` prefix when creating your `arquillian.xml` project structure.

Older containers – AS 5.1 and AS 6

Putting aside the Java EE profile support in each container, each of these containers are configured very similarly. In fact, it's actually easier to set up AS 6 in the managed mode than it is to set it up to run in the embedded mode. All options can be specified in your `arquillian.xml` (common point of configuration) and the Maven profiles look almost identical (just slightly different versions). As a result, if your goal is to test with JBoss you should strongly consider the managed containers over the embedded. These containers will start on the first test case that uses Arquillian. If your test cases are a mix of standard JUnit and Arquillian tests, then it will be booted on the first Arquillian test and remain running until the last test completes in the suite (not necessarily the last Arquillian test).

GlassFish 3.1

One of the interesting things that you'll notice when looking at managed versus embedded versus remote containers is where simplicity lies with each. GlassFish is one of the few containers that can support all three deployment models. Each one has some pros and cons for its operations and if GlassFish is your target production container you may find yourself running test cases in at least two, if not all three, of the container deployment models.

The GlassFish managed container is a lot like running in the embedded mode, except that you don't have the full download required to do your build; you simply need it for your testing. It does require that you have GlassFish installed and a `$GLASSFISH_HOME` environment variable set pointing to where you installed it (hint: `$GLASSFISH_HOME` is the directory that has the `glassfish` folder in it).

Unfortunately, one side effect is that you have to sync your container's start-up configuration with your local server's start-up configuration. This may generally be OK as long as you institute a common configuration on all developer workstations and continuous integration servers for specific domains and port configurations for a specific application.

Tomcat

Three versions of Tomcat are supported for managed deployment – 5.5, 6, and 7. The only injection supported out of the box with any of these is resource injection. If your target production container is Tomcat, then this is what you will want.

Tomcat-managed is similar in nature to JBoss managed-containers. Due to dependency requirements, it's likely easier for you to point to an internally configured Tomcat server. If your application uses Spring or Hibernate (or both) as well as perhaps some partial EE libraries, you may be interested in testing them completely in your application at this point. It may not make sense to unit test your application outside of this configuration, unless you did something to add your libraries to your application.

Further information around Spring testing and JPA/Hibernate testing is available when we discuss those specific extensions.

Managed container summary

Managed containers are more for automated integration tests. Since they are integration and automated, these are probably run less frequently on your code base. Where a unit test may run every build, an integration test may only run nightly, or every few hours if code has changed.

There should be only few cases where you consider a managed container for your unit tests. One may be that these are test cases that you need to run in a certain configuration, for example, testing out a data access tier to your application. You need to ensure that a database connection exists and points to a specific unit test instance of your application. These cases should be run on demand and not necessarily with every build; this is to avoid slowing down your development simply because a build kicked off.

Remote containers

There is actually a pretty short list of remote containers that are supported. Requests for support for more containers is based on community input; this can help define the overall plans for adoption.

A managed container, as noted earlier, is controlled by Arquillian including start and stop. Some remote containers can handle this scenario, where they will start automatically if not already running; this includes TomEE.

Tomcat

Only Tomcat 6 is a supported container for remote deployment. All other iterations of supported Tomcat (including Tomcat 6) are done via managed and embedded containers. Deployment is performed using the HTTP API and simply posts the deployment to the URL as required by Tomcat. You will need access to a user who can deploy to the server, as well as HTTP access to the server to perform the deployment.

Tomcat 6

Tomcat 6 is a Java EE 5 container that supports Servlet 2.5/JSP 2.1. As such, much of your business logic will be implemented using Spring or Hibernate and you will need to ensure that all of these libraries are embedded within your application. If you are using the ShrinkWrap 2.0 Maven Resolver API, this should be taken care of for you.

JBoss

With little surprise, you can expect that JBoss is a supported platform for Arquillian. Versions going as far back as 4.2 are supported for testing purposes.

Legacy JBoss – 4.x, 5.x, and 6.x

The support for remote containers using JBoss 4.2, JBoss 5/5.1, and JBoss 6/6.1 are provided by Arquillian. Note that since EAP and AS share the same code base, usually just patches and bug fixes that differ between them, each of your containers will work fine with either configuration.

In the cases of JBoss 5, 5.1, and 6 the dependencies are very similar. There are slightly different container artifact IDs as well as different versions for the `jboss-as-client` artifact (5.0.1.GA, 5.1.0.GA, and 6.0.0.Final).

I grouped these containers together since architecturally they are very similar and as a result they share a lot of configuration similarities. There are some differences to consider for their EAP editions. For JBoss 5 series, we have EAP 5.1. As known previously, there is no EAP version for JBoss AS 6 and instead EAP 6 was built from JBoss AS 7.1. JBoss EAP 5.1 requires considerably more configuration than the other containers. You will need to use AS 6's profile service when working with it.

JBoss AS 7

Similar to the managed container for JBoss AS 7, it has a remote container. It has essentially the same limitation about JMX configuration versus servlet configuration that may require you to switch to Servlet 3.0 protocol for your test to run. This is similar to the managed container from the configuration and Maven-dependency perspective.

GlassFish

Similar to the other GlassFish containers, the remote container for GlassFish will support any application through Java EE 6. The adapter is generally backward compatible with GlassFish 3.x releases, including 3.0.

GlasshFish 3.x

Using the GlassFish remote container, you can still deploy as you did using embedded and managed. The remote container allows you to also deploy to a cluster or specify a unique server instance to target for your deployment. This allows you to target different server instances in GlassFish for multiple deployment scenarios. You can do something very similar with the managed configuration as long as you choose unique domain names. This does have a little more overhead since you need to maintain all of the unique domains rather than maintaining a single cluster of servers. Since it is purely remote, it does require you to have started up the necessary server(s) for testing runtime usage.

WebLogic

WebLogic 10g and 12c both support remote containers for Arquillian. You do have to have the server running beforehand for each. The two containers are almost identical – 12c supports the Servlet 3.0 protocol and the WAR with EJBs deployment model (obviously, based on it using the Java EE 6 spec). The 10g container requires EJBs in JARs or JARs in EARs for deployment purposes. The protocol used is the Servlet 2.5 protocol.

The two containers are almost identical. They both support cluster-level deployment as well as deploying artifacts to individual managed servers (instances in other containers) that are part of an already-running system. WebLogic's deployment model allows you to start up a single AdminServer that can manage multiple clusters or manage servers without needing unique AdminServers.

WebLogic 10g

The only difference between the 10g and 12c containers has to do with the name of the artifact used for runtime. If you want to use the 10g container, use the adapter ending in 10.3 as the ID. Any standard Java EE 5 application will deploy to this container; you can deploy JARs, EARs, or WARs here.

WebLogic 12c

The only difference here is to use the appropriate adapter for 12c. The default protocol used here will be Servlet 3.0. You can deploy full WAR files (with EJBs embedded) as well as EARs and JARs.

Apache TomEE

The Apache TomEE remote container is actually the odd man out when looking at remote containers. The TomEE container supports a configuration that allows it to act like a managed instance or a remote instance. You can optionally specify a ZIP file that will include your TomEE download for you. This is done so that you don't need to have a preinstall to use TomEE in your test case. This makes the TomEE stack much more versatile. It is also automatically started with the test case – like a managed container would be. It also doesn't require any preinstalls, it can be run completely by your Maven test case.

TomEE 1.0/1.5

There are a few important configuration options required to run TomEE. TomEE provides a standard web profile as well as a "plus" version that includes some of the non-standard web profile tools such as JAX-RS, JAX-WS, and JMS included in the platform. You'll need to configure your surefire tests to set this up appropriately:

- Choose the "plus" qualifier in Maven for both the adapter and the container
- Set appropriate JVM settings, and `arquillian.xml` properties for any container-specific options

Whenever you try to start testing using TomEE, it will see if an instance is already running on the specified port. If none is found, the container adapter will start one automatically for you. You'll also notice that both TomEE 1.0 and 1.5 are listed. Both have support but contain different version numbers of their container adapters. No other change should be required.

Cloud deployments

Cloud-based testing is very new and, depending on how your cloud is set up, may require some special cases. If you are responsible for the entire infrastructure—setting up application servers, database servers, and so forth—you may need to do cloud-based development and testing. You can consider two primary types of cloud offerings, **Infrastructure as a Service** (**IaaS**) and **Platform as a Service** (**PaaS**). IaaS is essentially your application stack completely custom to your environment. PaaS leverages a provided and fairly standard platform for deploying your applications. If you are working in a IaaS environment, you can run your standard container deployment setup but in a remote setup. You could do embedded but then you would need to start up your build on that remote server (versus on a developer's workstation or your company's continuous integration server).

Cloudbees

As you are likely aware, Cloudbees offers development and runtime for applications; they are a PaaS offering. In order to test using Cloudbees, you should have a runtime account available to test with. Their development stack supports build management (Jenkins) for you.

Under the hood, Cloudbees is using Tomcat or JBoss as your runtime stack. You should expect that as long as your deployment works on those containers it should work correctly on the Cloudbees stack. As a result, you can sometimes save on the bandwidth and the expenses by testing your production, such as Cloudbees deployment on a local Tomcat or JBoss instance.

Since I could only find it with the source code, here is the qualifier to use with Cloudbees:

```
<container qualifier="cloudbees">
    <configuration>
        <property name="account">username</property>
        <property name="application">app-name</property>
        <property name="containerType">jboss</property>
    </configuration>
</container>
```

Obviously, you should also change `containerType` if you are not using JBoss (`tomcat` for Tomcat).

Remote testing strategy

A lot of what we have reviewed so far with managed and embedded containers can still apply to remote container testing; however, there are a few topics that need to be reviewed, especially if your remote containers are being used for other cases.

So what exactly is a remote container? With embedded and managed containers, Arquillian has been responsible for starting the container. Remote containers are assumed to be already running at the time of test execution. This gives additional configuration capabilities, such as deploying datasources, JMS queues, as well as defining configuration as needed (JVM arguments).

In addition, since these are remote containers they can be distributed across networked nodes, allowing you to test across multiple containers in parallel. This may also include Cloud-based containers such as OpenShift by RedHat and Google App Engine. Since Arquillian hides away a lot of the container details from the test cases, from your test perspective there is no difference between the two types of remote test cases.

Remote testing use cases

Remote testing scenarios are a little different than those we spoke of earlier. In general, the remote container should be used for integration and earlier-mentioned test cases. Since this is a remote container, we need to deploy to the container (possibly across network nodes). Due to the required processes it is not feasible to run a unit test across this setup. You could also run into cases where a container is not available, either local or remote, to the source of the test run. Since we should not put any possible external resources in the way of your unit testing, remote container testing shouldn't be considered for unit test purposes.

For other scenarios, pretty much every case, it makes sense to use remote test cases as long as you have the appropriate level of supported processes in place. There are a few cases that are unique to remote containers. Any test case that requires multiple targets for the same application is best done on a remote container (clustered application servers). Of course, if you want to access web requests in this setup, you need to ensure that you are using some form of a load balancer in between.

For example, if you want a nightly build that includes regression test cases, your CI server should include steps for starting the container before the tests are run and shutting down the container after the tests are complete.

Now you may wonder, when should I use a managed container and when should I use a remote container? Most of the time, you can manually start up the managed container to do the configuration prior.

Structuring the remote test case

Due to the use cases listed, remote test cases may need to be thought of differently as compared to the embedded test cases. A lot of this should also apply to the managed containers as well. The only difference (architecturally) between managed and remote is who is responsible for starting the container. Automated builds could be structured in a way that make remote servers start up before the build and shutdown after the build; it just requires a bit of ingenuity to make it happen.

Since remote containers include network hops to deploy the entire archive to the server, you should consider structuring your tests to have as few deployments as possible to increase the speed and decrease the network latency. One approach to this is to use test case delegates. This design paradigm is similar to the standard delegate pattern, except that it is done with test cases. You can still call your standard assert statements within your test case delegate, but you should plan to have a new instance available on each invocation (to ensure no state is shared). The easiest way to do this is to instantiate a new one each time, but this comes with a price – no access to container resources during the test. If you have CDI available, you can access these test resources via a managed instance. Here's an example that uses both.

First, the delegate class:

```
public class CalculatorDelegate {
  @Inject
  Instance<Calculator> calculatorInstance;

  public void testAdd() {
    Calculator calc = calculatorInstance.get();
    int expected = 5;
    int actual = calc.add(3, 2);
    Assert.assertEquals(expected,actual);
  }

  public void testSubtract() {
    Calculator calc = calculatorInstance.get();
    int expected = 3;
    int actual = calc.subtract(10,7);
    Assert.assertEquals(expected,actual);
  }
}
```

Next, the Arquillian test case:

```
@RunWith(Arquillian.class)
public class CDIDelegateTest {
  @Deployment
  public static Archive<?> createTestArchive() {
    return ShrinkWrap.create(JavaArchive.class)
        .addClasses(HelloWorld.class,Calculator.class,
         CalculatorDelegate.class)
        .addAsManifestResource(EmptyAsset.INSTANCE, "beans.xml");
  }

  @Inject
  Instance<HelloWorld> helloWorldInstance;

  @Test
  public void testCdiInstance() {
    assertFalse(helloWorldInstance.isUnsatisfied());

    HelloWorld helloWorld = helloWorldInstance.get();
    HelloWorld helloWorld2 = helloWorldInstance.get();

    assertEquals("Hello, World!", helloWorld.getText());
    assert helloWorld != helloWorld2;
  }

  @Inject
  Instance<CalculatorDelegate> delegateCalcInstance;

  @Test
  public void testDelegateAdd() {
    delegateCalcInstance.get().testAdd();
  }
  @Test
  public void testDelegateSubtract() {
    delegateCalcInstance.get().testSubtract();
  }
}
```

The delegate looks like any regular CDI bean. We do have to include the delegate in the bean archive to enable CDI for it. Otherwise it won't be registered as a bean always. Within the delegate, we structure it to retrieve a fresh instance of the CDI bean under the test for each test case. This is ideal to avoid any sort of state leaking (we know this class doesn't have state leaking, but still this is good practice).

The second step is to create the actual test case. We start with the skeleton of the Hello World test built earlier. This test case allows us first to demonstrate the usage of the instances to validate within a test. We use an `Instance<>` tag to get the instance of a bean that has CDI capabilities. We test this approach out with the Hello bean to also validate that the two calls to the `get()` method return different instances. Then we build test methods that call the `CalculatorDelegate` class to do the work. `CalculatorDelegate` tests out the `Calculator` object, both of which are CDI enabled. These methods are simple one-line methods – using the `Instance` object get a reference to a `CalculatorDelegate` class, then perform one of the test methods on that class.

Other remote cases

One of the topics that will be covered in a later chapter is testing the services deployed. This could include web services or REST services. The simplest case is the servlet example provided earlier – you are just performing an HTTP request and parsing the response. There is a remote testing capability that does not send your test case to the deployment and instead performs testing locally in your JVM but still does the deployment.

Remote testing tools

These are some important tips to keep in mind when using remote containers. This applies to both remote containers as well managed containers. These tips can apply any time when the container is in a separate JVM.

Deploying to multiple containers

Arquillian supports deploying to one or more containers within a test case. Each container needs to be defined and qualified. You can also define your test cases to operate against a specific deployment. Not every test case works with every deployment; as a result we need to ensure that we run the right set of tests with each deployment artifact. This may account for container-specific deployments.

Using this approach with some containers may also allow you to do things such as clustering or cross-container tests. With WebLogic, for example, you can build out a test cluster and deploy to the entire cluster (as long as you target the cluster in your configuration). This of course depends on the application server supporting it. You could, likewise, using TomEE (for example), create the same WAR or JAR multiple times and target it to each container – allowing you to create multiple containers for your archive.

The power of remote testing

One of the benefits of using a remote test case – either managed or remote (from the container deployment perspective)— is your ability to interact with it the way a client would. If your application has a web-based user interface and exposes a series of REST APIs or SOAP APIs, you can access those web resources from your test case to validate the results.

When we begin talking about test enrichment, we'll see that Arquillian is essentially using CDI to do its injection work and will allow the injection of resources into the test cases. This can include your application-defined resources as well as a series of internal-to-Arquillian resources that work with your test case.

Non-testable deployments

There are cases when you need to deploy an application but not repackage Arquillian with it. One of the changes that occur with Arquillian deployments is that the Arquillian libraries are deployed with your code. You can change the testable attribute of `@Deployment(testable=false)` to `false` to turn off Arquillian rewriting your deployment. What this also means is that your test case does not get deployed – it is now running in the test JVM.

When this occurs, you have options to interact with your application on a remote basis. This can include access to the remote JVM's JNDI (via Context) as well as URL-based access to the remote application. Now, you can also do this when you are running in the embedded mode but it doesn't always make sense when you look at it from an architectural perspective. This kind of testing makes sense when you are remote, but if you are in the same JVM then the results can be skewed.

Arquillian supports three container-testing models, which apply to your entire test case:

- **In container**: This is the default execution mode of Arquillian. Your test case is repackaged with your deployment artifact and tested in the container. Arquillian will gather results of the test execution via its protocol with the container.
- **As client**: Your deployment exactly as specified is deployed to the container and your test case runs in the client JVM (the test JVM or Surefire JVM in Maven).
- **Mixed mode**: Your deployment is deployed in the same way that the in-container version is deployed. Your tests are run in two ways, one on the server and the other in the client JVM. By default, tests are run on the server; however, the `@RunAsClient` annotation can be added to a test to make it execute on the client side.

Mixed mode simply allows you to have either client or in-container test methods in the same test case. This allows you to reuse deployments across remote test cases and local test cases. As noted, in container is the default scenario when you run Arquillian. Since Arquillian repackages your deployment with their artifacts for executing test cases there is an additional case to think about – if you are running in mixed mode, will there be a redeployment of the artifact to remove Arquillian? No there won't be. The Arquillian libraries will simply not be used during the test execution.

Remote container summary

As it should be clear now, remote containers can have more requirements for operation than managed do. They also have limitations around the number of instances that can be running as well as the overall network requirements. While it is true that you can run a remote test case locally, it does have certain prerequisites that may be difficult to ensure are present for a test case to run.

Container comparison

The one thing I cannot do for you is tell you what container you should deploy your production code to. I am assuming that this decision was already made and you are simply using Arquillian for testing purposes. What I can do so help is provide an insight as to what is used for testing purposes based on your production application server. While the types of container runtimes (embedded, managed, or remote) are listed throughout the chapter, you may consider this quick chart an easier interpretation. A "Y" in the column value means that this is a supported configuration.

Container	**Embedded support**	**Managed support**	**Remote support**
Apache OpenEJB	Y	N	N
Apache OpenWebBeans	Y	N	N
GlassFish 3.1	Y	Y	Y
JBoss AS 5	N	N	Y
JBoss AS 5.1	N	Y	Y
JBoss AS 6	Y	Y	Y
JBoss AS 7	N	Y	Y
JBoss AS 7.1/EAP 6	N	Y	Y
Jetty	Y	N	N
Tomcat 5.5	N	Y	N
Tomcat 6	Y	Y	Y

Container	Embedded support	Managed support	Remote support
Tomcat 7	Y	Y	N
TomEE	Y	Y	Y
Weld SE, EE	Y	N	N
WebLogic 10g	N	N	Y
WebLogic 12c	N	N	Y
Cloudbees	N	N	Y

As you can see, in general, full containers are not supported by embedded (GlassFish being the main exception, and JBoss AS 6 providing a clever workaround). Apache TomEE receives a "Y" for both managed and remote since the remote container performs an automatic start on the local machine.

Tomcat

If you are using Tomcat as your production deployment then you should try to run your test cases with Tomcat as well. Tomcat has supported embedded modes for 6 as well as 7 so all recent versions are supported. If you want to run something even lighter, you could run Jetty but you may run into compatibility problems. For integration testing, you should continue to deploy to Tomcat. If you want to run locally and have all configuration necessary, you can continue to use an embedded Tomcat, but you need to consider using a managed or a remote Tomcat. I would strongly recommend using a remote Tomcat for automated acceptance; this way your infrastructure has the ability to support the instance and confirm it as a production-like environment. You cannot do that with embedded and it may prove to be difficult to run your automated tests on an existing managed instance.

Apache TomEE

Apache TomEE is a relatively new container that targets the Java EE 6 web profile. If you are targeting it for production use then there is no reason to not use it for testing purposes as well. It uses OpenEJB and OpenWebBeans under the hood. If you are looking for containers that are lightweight, you can use those as your target containers to only test things such as EJB or CDI support in your application. You should also take care when looking at TomEE; there are two flavors available: the pure web profile version as well as a plus version, which have some minor differences that give your applications more functionality. These include the additional tools JAX-RS and JAX-WS (via Apache CXF), JMS (via Apache ActiveMQ and OpenEJB), as well as Geronimo's Connector architecture.

If you are deploying to TomEE plus and you need to test your application, which has SOAP or REST-based web services, you'll need to ensure that you use the TomEE plus version.

The TomEE container is very flexible in the remote JVM world. It essentially supports both managed and remote functionality in the same container libraries, which makes it much easier to use.

JBoss

As noted previously, different versions of JBoss are supported by Arquillian in different profiles. JBoss AS 7 is a complete redesign of the application server. As a result, the behavior of the containers and the configuration of the containers is different across the versions.

Pre-JBoss 6

If you are running an older version of JBoss, you only have a few options. If your application does not use any EJBs, then you may want to consider running or testing your application on the equivalent Tomcat version for that version of JBoss. Since JBoss is built on Tomcat, you can consider using an embedded Tomcat instance for your unit test purposes. This will provide you with an equivalent platform to test on. If you are used to JBoss's deployment of Hibernate with your application, you will need to ensure that you include the Hibernate libraries inside of your deployment (you will also need to limit your testing to WAR files). If you are using EJBs, you will need to decide if it is important to unit test them. If you do decide to unit test them, you will likely want to use a managed container to do that unit testing. Due to compatibility problems you may face, it is likely easier to just deploy to the managed container. If you are doing more rigorous testing – integration or acceptance testing—you should target the appropriate JBoss instance. If you have access to a remote container you should use that as much as possible. This will help simplify any problems running on the local machine and ensure that it is a supportable platform.

JBoss AS 6

JBoss 6 was one of the first Java EE 6-compliant containers around, targeting the web profile for deployment. As a result, it has many of the Java EE 6 capabilities you may want to use in your testing. If you want to unit test targeting this container, you have some choices based on what you use. If you want to test your CDI components, you can use Weld. You will need to use the Weld EE container and use a slightly older version. Weld 1.0 is what is shipped with JBoss AS 6, so make sure you use that version with your testing. While older versions of JBoss used closer Tomcat implementations, JBoss 6 uses a version with some major changes, referred to as JBoss Web.

JBoss AS 6 does include an embedded container; however, it is slower to start up since it launches all services. It also requires the server to be installed locally, so it may cause problems if run often. You can use a Tomcat 7 version with your testing but it will not be an exact copy of your test scenarios. For integration and acceptance purposes, you should use a managed or remote instance of JBoss AS 6.

JBoss AS 7

JBoss AS 7 is the current iteration of JBoss for production use. It is also Java EE 6-compliant and has support for both full and web profiles. If you are working with CDI components, you can also test this using the Weld EE container. This should work fine with the latest Weld version (though you should use the version that ships with JBoss AS 7). This container has no embedded mode, so it's not very feasible to run it during unit testing. You could run any web app pieces on a Tomcat 7 instance, but it will have some differences.

TomEE may be a suitable alternative if you are using JBoss AS 7 as your runtime. If you are interested in unit testing with a lightweight framework that should mostly be compatible, you might want to consider using TomEE as your base runtime. If you do use certain services explicitly (for example, RESTeasy for JAX-RS or Hibernate for JPA) you may need to include those libraries explicitly in your test suite. You may also need to consider using GlassFish for this, for essentially the same reason as TomEE. You will need to repackage certain libraries but in general you can bring up a compatible application server in this fashion.

GlassFish

I seriously believe that the GlassFish containers are the easiest to use and yet the most robust of all containers supported by Arquillian. Its startup time is pretty quick when using it in the embedded mode. It does have a large download to consider though, since it uses a purely embedded application server and does not use any external dependencies. You should ensure that your Maven configuration does not pull this down on every build and instead uses an already downloaded version of the JAR file. In addition, you can use the managed and remote containers to be able to deploy your entire application for testing purposes.

Cloud containers

This is where testing gets a little tricky. Since cloud containers generally have a monetary value, you'll need to weigh the need to test in the cloud with what you can do locally.

CloudBees

CloudBees is a bit easier to describe for testing purposes. For acceptance purposes, if you can get CPU power available to test in the cloud using CloudBees, you should. Otherwise, it is not difficult to bring up local internal instances of the platform to handle that. Since CloudBees uses both Tomcat and JBoss, you should review those options for alternatives to your cloud-based solution.

OpenShift

OpenShift gets a little more complicated with cloud-based testing. Similar to my recommendation for CloudBees, if you can afford to deploy to OpenShift for testing purposes, please do so. OpenShift uses a modified stack of RedHat Linux and JBoss AS 7 to drive its cloud. There are also certain capabilities turned off that are very hard to replicate. If you can deal with this piece, I would recommend following the JBoss AS 7 approach for testing for some additional ideas.

WebLogic

If you are deploying to WebLogic 10.3, then you should target that stack for all testing. Yes, this does mean you need to deploy to a remote container for unit testing purposes. I would strongly recommend against using WebLogic 10.3 for unit testing. The time line it requires is too long for unit tests to run in. It is suitable to run integration tests here as well as automated acceptance tests.

WebLogic 12c is a little different. I still would not recommend using WebLogic 12c for unit testing purposes since it only supports remote deployment, but there are other uses for it and potentially good matches for the containers. Under the hood, WebLogic became better aligned with GlassFish with this release. Many of the component integrations are from GlassFish. In addition, WebLogic understands many of the GlassFish descriptors and vice versa. As a result, you may want to consider using GlassFish in either the embedded or managed mode if you want to work in that setup.

One important thing to point out, which can get frustrating, is that WebLogic has some new libraries embedded. WebLogic has started shipping with SLF4J bundled in. This can cause some problems when trying to test, so please make sure you remove the SLF4J dependencies from your deployment. Since WebLogic 12c uses Weld as its CDI implementation, you can start using that for unit testing purposes around CDI beans.

Other setups – the embedded applications

There are going to be cases where what you've deployed is actually an embedded application – this can happen through some combination of using a native Tomcat instance plus some other libraries to run your application. This could also apply to your usage of Jetty. If this is your case, then you probably have already started using the right container for unit testing. Then, for integration testing and acceptance testing, you should look at how your application is started up to determine the right approach. If you have shell scripts or similar that start the application, using certain JVM settings, these should be accounted for in the configuration of your Maven surefire plugin. You should then be ready to continue to test using your application as deployed regularly. This could be assuming that you have certain classpath entries made or such. The point though is that if you use embedded for your production deployments, use them in all levels of your testing.

Unofficial containers

At the time of writing this, there are a number of containers where only partial support exists at this time. This may be driven based on timing or licensing issues. This includes Google App Engine, IBM WebSphere, and Caucho Resin. Unfortunately, these containers are not developed enough yet to speak in depth about.

Summary

As you've likely noticed, within the container chapters we have not dug much into code examples. The goal here was to explain how to work with the containers and their role in your test cases. As noted, when looking at the overall testing strategy versus the deployment methodologies, you'll see that managed and remote containers have much more capability than the embedded containers – at least in some cases. The embedded Tomcat, TomEE, and GlassFish versions support the same capabilities in all three setups.

Next, we'll review how testing works with Arquillian. Now that we know how to work with our containers, we can start to deploy test cases to the container and watch them execute, and luckily also pass as we run them.

4
Why Did the Test Fail?

Now that we've reviewed how containers work and are configured, it's important to talk about some of the difficulties faced when working in the container. This chapter is aptly titled *Why Did the Test Fail?* because you will get to a point in working with Arquillian where you just end up scratching your head trying to understand why it didn't work as expected. This chapter isn't necessarily going to fix all of those issues, but it will help you come up with some sort of troubleshooting approach to identifying those problems and ways to handle them.

Running your tests – the downside

This chapter focuses entirely on container-level problems when running tests. These problems can be incompatibility of the JDK used or a remote server being offline. While this isn't meant to be a comprehensive guide on how to fix every problem, the hope is that this gives you ideas on where you can look to solve your problems when deploying your code during testing.

Dealing with problems running the container can be frustrating. Embedded containers typically log well on the client side, but you will not find the logs in the Surefire reports. You will need to capture log output in the output of your Maven run. With managed containers, the logs will be captured within your server's logs. Output from running the test client will be captured within your Surefire reports. It's essentially the same for remote containers, except that managed containers will contain startup and shutdown commands when complete.

The frustration

One of the struggles you may face as you are building test cases is how to package your application. This section is all about packaging problems – whether it's the setup of your deployment, the contents of it, or how it's targeting a container, this section helps to figure out where these problems exist and how to try to rework your packages to match what your container may expect for the content.

Packaging structure

There's nothing better in Servlet 3.0 than the ability to deploy a WAR file without `web.xml`. Try doing that in Servlet 2.5; the results will be odd at best. You'll probably get a deployment exception from your container. Let's say we changed our Servlet test case to have a deployment like the following:

```
@Deployment
public static Archive<?> createTestArchive() {
  WebArchive wa = ShrinkWrap.create(WebArchive.class,"test.war")
      .addClass(HelloServlet.class);
  return wa;
}
```

You'll get different results with this depending on how you run it. In this case, I chose to run it using the Tomcat 6 Embedded profile. It turns out that this case actually deploys correctly but runs into issues when running. Mostly that's because the Tomcat 6 profile compiles against Servlet 3.0 APIs. The output from Maven/Surefire looks something like the following:

```
testGetText(com.tad.arqdevguide.chp3.servlet.HelloServletTest)
  Time elapsed: 0.932 sec  <<< ERROR!
java.io.FileNotFoundException: http://localhost:8080/test/hello
  at sun.net.www.protocol.http.HttpURLConnection.
  getInputStream(HttpURLConnection.java:1613)
  at java.net.URL.openStream(URL.java:1035)
  at com.tad.arqdevguide.chp3.servlet.HelloServletTest.
  testGetText(HelloServletTest.java:35)
```

Clearly the issue here is that we had a file not found. Why is that though? We deployed the same things in Tomcat 7, Jetty 8, and they all worked as expected. As you may be aware, one of the changes in Servlet 3.0 is the dropped need for `web.xml` and the addition of annotation-based servlet mapping. This was more or less ported from the EJB specification's annotation support for most of the configuration. Since our application is based on Servlet 3.0 we cannot use it in Tomcat 6 – even though we were able to make it compile using a Maven profile. If we want to run it in Tomcat 6, we need to bring back the `web.xml` configuration file as well as its servlet mapping.

This is simply one example. We wanted to try making our application run in older containers. We may need to evaluate how we have our application set up to choose what configuration files apply in this scenario. This is a very common problem that you may run in to when starting with Arquillian. You should obviously remember your requirements versus your build structure and make intelligent decisions around what you test against or what you compile against. Requirements come first, then your testing. This is also meant to show how simple packaging differences may cause problems with your testing.

The wrong type of archive

Sometimes when you are testing, you're going to run into test cases that you want to try in several containers. As a result, you may end up with some deployments that generate different archive types. Going back to *Chapter 3, Container Testing*, let's take a look at the following JAR file:

```
@Deployment
public static Archive<?> createTestArchive() {
  return ShrinkWrap.create(JavaArchive.class)
      .addClass(HelloLocalBean.class)
      .addAsManifestResource(EmptyAsset.INSTANCE, "ejb-jar.xml");
}
```

This EJB deployment works fine when you target OpenEJB, most full enterprise containers as well as embedded CDI containers. It won't work if you try to deploy it to a servlet container. The error you'll see is very obvious as well (again, taken from Tomcat 6 embedded):

```
testEJBBootstrap(com.tad.arqdevguide.chp3.ejb.HelloLocalBeanTest)
Time elapsed: 0.12 sec  <<< ERROR!
java.lang.IllegalArgumentException: ArquillianServletRunner not found.
Could not determine ContextRoot from ProtocolMetadata, please contact
DeployableContainer developer.
  at org.jboss.arquillian.protocol.
  servlet.ServletUtil.determineBaseURI(ServletUtil.java:64)
  at org.jboss.arquillian.protocol.servlet.
  ServletURIHandler.locateTestServlet(ServletURIHandler.java:60)
  at org.jboss.arquillian.protocol.servlet.
  ServletMethodExecutor.invoke(ServletMethodExecutor.java:77)
  at org.jboss.arquillian.container.test.impl.execution.
  RemoteTestExecuter.execute(RemoteTestExecuter.java:120)
```

The error here is nothing like the previous error but has a very similar result. The test fails in this container but not in others. Even though our code is very platform independent, the deployment test is not. As a result, you need to be careful about how you test and where. If you review the Maven profiles in the sample code, you will note that certain build configurations use different patterns to choose which test runs with which container. A similar practice should be followed in all containers or testing phases supported in Arquillian – choose which tests are run in unit, integration, acceptance, and performance testing phases.

The container-specific descriptor

A lot of times, especially if you are more on the product development side, you end up including descriptors for every container in your application in order to account for the quirks in various environments. Worst yet – you may end up with a compilation/packaging step on the client machines to account for these issues. I always prefer the build-once-and-deploy-anywhere setup that Java is so proud of.

You may be tempted to dynamically pick your container-specific descriptor during your testing. You may also be tempted to always add them to your archive. Each approach has pros and cons, but in general there are some best practices to remember to keep your project building in a stable fashion without having leaks.

In general, I recommend that you include all of the container-specific descriptors in all test cases. This allows you to have one method that creates your deployment archive and not care about building logic to read which container profile you are using and how to choose what files go into it. If you have a lot of files going into your `WEB-INF` folder, or `META-INF` folder, you should consider adding some code like the following to your deployment method:

```
public static Archive<?> createTestArchive() {
  File[] webinfs = new File("src/main/webapp/WEB-
  INF").listFiles();
  WebArchive wa = ShrinkWrap.create(WebArchive.class,"test.war")
      .addClass(HelloServlet.class);
  if(webinfs != null) {
    for(File f: webinfs) {
      if(f.isFile())
        wa.addAsWebInfResource(f);
    }
  }
  return wa;
}
```

It is a little more involved: you have a loop, which is a check to see if something is not null, and a check on each file to make sure it's a file. This also grabs files that we inadvertently missed in building the archive originally – `faces-config.xml`, `beans.xml` - that were already in `src/main/webapp/WEB-INF` but not included in our builds. Right now we're not using JSF in this sample app, but this makes it easier to add when we start adding other descriptors. This will also automatically bring in a `weblogic.xml` or `jboss-web.xml` to be included in your application.

If, for some reason, you want to choose which descriptors to deploy with your WAR file, the easiest way to determine them is to work based on the JVM properties passed in. As you have noticed, the deployment method is static. It will always run in the client JVM. As a result, if you modify how the client JVM is run then you can tell the deployment method to do something different. In the following example, I actually use the Jetty container, and specify an `argLine` tag in the build configuration, as follows:

```
<argLine>-DjettyMode</argLine>
```

In your test case, you can then add in some logic as follows:

```
@Deployment
public static Archive<?> createTestArchiveProperties() {
  WebArchive wa = ShrinkWrap.create(WebArchive.class,
  "test.war")
      .addClass(HelloServlet.class).
       addAsWebInfResource("web.xml");
  Properties props = System.getProperties();
  if (props.containsKey("jettyMode")) {
    wa.addAsWebInfResource("jetty-web.xml")
    .addAsWebInfResource("jetty-env.xml");
  }
  return wa;
}
```

All that we added in this example is an extra check to see if we're in the Jetty mode, and if so we add the Jetty-specific files to the deployment, but only when we have a JVM option indicating such. Similarly, the method can be extended with other options as follows:

```
  if (props.containsKey("jettyMode")) {
    wa.addAsWebInfResource("jetty-web.xml")
    .addAsWebInfResource("jetty-env.xml");
  } else if(props.containsKey("weblogicMode")) {
    wa.addAsWebInfResource("weblogic.xml");
  } else if(props.containsKey("jbossAs7")) {
    wa.addAsWebInfResource("jboss-web-as7.xml","jboss-web.xml");
  }
```

In this code snippet we've added a basic `if/elseif` block that checks for various JVM arguments. If you do end up doing this, you'll start wanting to create static variables representing your test containers to have specific keys. You will just need to repeat them once in your `pom` file, and then wherever the static values are.

Another approach you can take, if you don't want to add a lot of logic to your test case setup, is to dynamically set up test resources based on different profiles. Under your build tag, you can have `testResources` and then `testResource` tags. Each `testResource` should point to a directory. In one case, this may be `src/test/resources` and in another, it may point to `src/test/resources-jbossas7`. This would only work assuming that your filenames were the same but had different contents per application server.

Wrong package version

This doesn't happen in every container, but can cause a few problems with your test cases. Let's say you target WebLogic 10.3.5 for your testing, but somehow you built a Java EE 6 application (note that WebLogic 10.3.5 does support a subset of Java EE 6, but notably is missing CDI and EJB 3.1 services and deployment structure). Let's also presume that you have packaged it into an EAR file. As long as you use Stateless or Stateful Session Beans (MDBs would work here as well), your test cases should actually work fine. However, let's say that you decide to repack this into a WAR file, something which is now supported in Java EE 6, you will find that your tests fail. This is because WebLogic will at best treat this as a Java EE 5 package. As a result, your application will not function.

The hard hat area

As pointed out a few times, when you are working with remote containers (managed and already running) you can run into issues where the container may have some startup errors. This section focuses on how others may impact your testing capabilities – whether they are using your test server or have caused a bug upstream from your code, these can have negative impacts. Continue to watch out for what someone else may have given to you.

Sharing resources

This is one very dangerous place to stay with your testing. With all of the quality assurance, and quality engineering teams I've worked with, I have never come across a group that was actually enthused about using a shared environment.

You need to be careful when working with a remote or managed container during an automated build process. Do you have steps in place to ensure that simultaneous builds are not invoked? I was recently working on a release with someone when they kicked off the core libraries – it resulted in 35 downstream builds being kicked off. Some of these jobs ended up running multiple times due to staggering and cross dependencies. When this occurs, can your build environment support multiple builds each running a managed or remote container?

In most cases you cannot. It's simply too resource intensive to do so. If you have builds set up like this in an automated environment – say a Jenkins or a Bamboo infrastructure, you need to be able to control which jobs will launch containers and at what times. You can often run into build failures simply because another build was running that happened to be occupying the same ports as your container. This causes a failure in the start of your container and your tests to fail, usually requiring a manual restart to reset. If you can, avoid using managed or remote containers in a continuous integration build. It helps ease problems with running tests that could interfere with one another. Leave these to be a part of a nightly build process wherein your application is compiled and tested thoroughly.

Hello, are you listening?

Another problem you can run into when working with remote containers (particularly those that are started already and not started as part of your build) is that they go down. There could be issues with their **permgen** space (the segment of memory that allows you to deploy) or other problems that really have nothing to do with your test cases. In the case of a planned outage – server updates being applied, known downtime in some development or test database, or even as simple as something needed to be restarted; you should make sure it's clear to all impacted development team members. If needed, you should put your continuous integration builds in a quiet period before picking back up and building again.

Ideally you can have some kind of automated process that restarts your container in the case of an outage. Maybe this includes running some kind of shell script over SSH to the remote host, or the remote container is actually running locally, in which case you can just restart the process from the local command line. Either way, it should be easily done and hopefully automated to ensure that your tests are not impacted by an outage.

Watch out for falling bugs

As you know, all software is perfect and bugs never exist in production-released code. Oh wait, I'm sorry. That was a lie. In all actuality, you do need to watch out for problems with bugs in production containers that you may be deploying to. Arquillian is only as good as the container that it's running on. If your container does not properly handle JAX-RS resource binding, or it has limitations in its EJB implementation, then Arquillian will demonstrate these problems as well for your test cases.

If this is your production-level application server, then you will likely hit these limitations when moving to production. If you want to validate that these are problems specific to this container, you can try running your tests against another container. If you need full support, GlassFish embedded may do well for you. TomEE embedded may also do well if you just need a web profile.

Problems starting up

While I cannot tell you how to handle every single start-up issue, the information here may be a good resource for starting to understand problems you'll run into. This section should help you identify why your managed or embedded containers fail to start and how to try to resolve those types of problems.

The ports are working against you

This is where Arquillian proves to be a bit confusing. To show this, I will use the TomEE profile, but change the port in use to be `665`. This is an illegal port in Linux; ports below 1000 require root to start up the listener (and hence the pains of HTTP's 80 and 443 default ports). In the TomEE embedded profile, I'll change the following line:

```
<tomee.httpPort>665</tomee.httpPort>
```

Then change `HelloServletTest` to use port `665` in the URL. I'll then run the following test:

```
mvn test -Ptomee-embedded-1.5
```

The error that comes back (from the test case perspective) is:

```
testGetText(com.tad.arqdevguide.chp3.servlet.HelloServletTest):
Connection refused
```

This isn't the true error though. If we scroll all the way to the top of the test, we'll see the first problem that pops up:

```
INFO: Initializing ProtocolHandler ["http-bio-665"]
Nov 10, 2012 6:37:39 PM org.apache.coyote.AbstractProtocol init
SEVERE: Failed to initialize end point associated with ProtocolHandler
["http-bio-665"]
java.net.BindException: Permission denied <null>:665
  at org.apache.tomcat.util.net.JIoEndpoint.
  bind(JIoEndpoint.java:386)
```

This gives us a general idea about the problem, but nothing concrete. Maybe we'll search around (if you didn't already know about the 1000 port issue prior to the last paragraph).

The purpose here though was more important – just like regular Java stack traces, Arquillian exceptions need to be found all the way at the top of your tests, not at the bottom.

The flip side of this issue is also important – let's say I tell TomEE to run on a port that's already in use. The error there can be even more confusing:

```
  testGetText(com.tad.arqdevguide.chp3.servlet.HelloServletTest):
  Unexpected end of file from server
```

The underlying problem is a bit similar, but it shows why TomEE prefers to run on a random port:

```
INFO: Initializing ProtocolHandler ["http-bio-58606"]
Nov 10, 2012 6:43:26 PM org.apache.coyote.AbstractProtocol init
SEVERE: Failed to initialize end point associated with ProtocolHandler
["http-bio-58606"]
java.net.BindException: Address already in use <null>:58606
  at org.apache.tomcat.util.net.JIoEndpoint.
  bind(JIoEndpoint.java:386)
```

In the next chapter, we'll review Arquillian resources and how they help resolve problems similar to the one we just saw. These errors are very common, and you'll likely run into very similar responses from other containers (it's almost scary how many containers use Tomcat under the hood for Servlet support).

You need a container

Arquillian has a very simple ground rule. In order to run, you need one container on your classpath, no more and no less. Avoiding multiple containers on your classpath is very simple: just use Maven profiles to control the classpath. Making sure you have a container on your classpath can be tricky at times. Perhaps your default setup doesn't use Arquillian. Did you also tell Maven to not run the Arquillian tests in this configuration? If we use the sample application as an example, here's what happens when we just run `mvn test`:

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running com.tad.arqdevguide.chp3.junit.HelloWorldTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0,
Time elapsed: 0.16 sec
Running com.tad.arqdevguide.chp3.cdi.HelloCDITest
Tests run: 1, Failures: 0, Errors: 1, Skipped: 0,
Time elapsed: 0.057 sec <<< FAILURE!
Running com.tad.arqdevguide.chp3.ejb.HelloLocalBeanTest
Tests run: 1, Failures: 0, Errors: 1, Skipped: 0,
Time elapsed: 1.622 sec <<< FAILURE!
Running com.tad.arqdevguide.chp3.delegate.CDIDelegateTest
Tests run: 1, Failures: 0, Errors: 1, Skipped: 0,
Time elapsed: 0.004 sec <<< FAILURE!
Running com.tad.arqdevguide.chp3.servlet.HelloServletTest
Tests run: 1, Failures: 0, Errors: 1, Skipped: 0,
Time elapsed: 0.034 sec <<< FAILURE!

Results :

Tests in error:
  initializationError(com.tad.arqdevguide.chp3.cdi.
  HelloCDITest): Ljavax/enterprise/inject/spi/BeanManager;
  com.tad.arqdevguide.chp3.ejb.HelloLocalBeanTest: DeploymentScenario
  contains a target (_DEFAULT_) not matching any defined Container in
  the registry.
  initializationError(com.tad.arqdevguide.chp3.delegate.
  CDIDelegateTest): Ljavax/enterprise/inject/Instance;
  com.tad.arqdevguide.chp3.servlet.HelloServletTest:
  Could not invoke deployment method: public static org.jboss.
  shrinkwrap.api.Archive com.tad.arqdevguide.chp3.servlet.
  HelloServletTest.
  createTestArchiveProperties()
```

We had one successful case and four errors. The errors give no specific clue about what is missing, but here's what they mean (the order is based on the run list):

1. Success.
2. The `BeanManager` class could not be found.
3. No container was configured.
4. The `Instance` class could not be found.
5. The `HttpServlet` class could not be found (you'll need to read the Surefire report for this).

In three of the cases of running in this setup, we had failures due to missing class files. These occurred because the profiles listed are where we pull in the Servlet and CDI dependencies, since different container configurations want different versions of libraries involved. If you want to be able to just run `mvn clean install` or `mvn test` to execute something, you should have a default profile set up – perhaps your unit test profile - so that something will always be running and not necessarily need a profile to be defined on the command line. This doesn't necessarily need to include Arquillian, but you would probably be testing more thoroughly if it did.

Obviously, this also reminds me to reiterate a problem I raised earlier in the book. You need to test the right types of components in the right environment. While CDI allows you to run an EJB, it's not the same as the EJB container. You have absolutely no way to run a Servlet in a CDI container. You just need to remember this when you are setting up your test cases in your profile configuration. This is similar to configuring your test cases based on your run configuration – are you running unit tests, integration tests, or acceptance tests? One piece that I skipped over during the description is that your unit tests may require multiple tests to run through the entire suite. If your unit tests include Servlet tests, CDI tests, and EJB tests, you may need to run the following:

```
mvn clean install -Punit-cdi
mvn test -Punit-servlet
mvn test -Punit-ejb
```

So we compile the code base once, but then run tests for the `unit-cdi` profile, then the `unit-servlet` profile, and finally the `unit-ejb` profile. This would be the same in the sample application if we did the following:

```
mvn clean install -Pweld-ee-embedded-11
mvn test -Ptomee-embedded-1.5
```

We end up running all of our CDI test cases in Weld, which matches our production CDI container as well as the full suite of tests against TomEE. This can easily be set up in our Continuous Integration environment as just a second build step. Depending on what you use, all test cases should get recorded and reported (though, only once in most cases since the prior run will be overridden by Maven). You can always investigate the console to see what was run and how.

No container? No problem!

What if your application doesn't run in a container? Let's say you've built a socket server that needs to be tested. There is no container involved, but perhaps some frameworks such as Netty and Spring that run your code. Perhaps your code runs from the command line and a series of shell scripts that invoke them. Arquillian has a new container to support this approach. The Arquillian Daemon container is a lightweight standalone JVM for running your tests. If you want to use Arquillian, you can use this container to run your tests.

Consulting the remote server

Using a remote container to perform tests has some unexpected side effects. Sometimes your tests rely on the classpath setup that is within your test code, but not necessarily on the container's classpath. Likewise, when you run an embedded test, these classpath entries may exist but not in your remote test case. You can always end up in a situation where it's not clear what is going on. In some cases, you can end up with errant WAR files still being deployed to your application server – even though they should have been undeployed during your testing.

The most common problem you'll run into when testing on remote servers that fail is that they report something along the lines of the following:

```
org.jboss.arquillian.container.spi.client.container.
DeploymentException: Unable to deploy
```

When you see this exception, you need to review your remote container to see what errors are shown, if any. In this case, I'm using TomEE remote. The error I see is the following:

```
SEVERE: Application cannot be deployed as it contains
deployment-ids which are in use: app: /tmp/1/test
org.apache.openejb.DuplicateDeploymentIdException:
Application cannot be deployed as it contains
deployment-ids which are in use: app: /tmp/1/test
arquillian-protocol.war.Comp
```

I need to point out that I caused this error to occur behind the scenes. Midtest, I shut down TomEE. This caused my "test" deployment to still exist and not undeploy correctly, since Arquillian can only undeploy the application if the container is running. This means that next time and all subsequent times I did this to simulate a stuck deployment, which can happen on any number of containers. In WebLogic, you just need to delete the deployment on next startup. In TomEE, you need to delete the work directory and modify `conf/deployments.xml` to remove the test entries.

You should also monitor logs. Once you resolve deployment-type issues, it is important that you monitor your application as you would if this were production. This can include tailing logs, monitoring for e-mail alerts, and checking database statuses to ensure that your application is operating as expected.

Managing your test runtimes

One item that has been touched upon but not discussed in great detail yet is how to control which tests run where and how to best manage them. We've discussed creating Maven profiles but we will discuss this topic in the subsequent sections.

Make it compile

This can be the most difficult part of the testing phase. You need to ensure that every profile in your Maven build compiles both the main source and the test source correctly, even if only a part of that code is being tested. Unfortunately, there is no catch at all for every profile on how to set up the compilation properly. You should ensure that every profile has access to the basic Java EE libraries if that is what you need to compile your code.

You should ensure that only the needed libraries are included in your bundle. If you are using the Maven Dependency Resolver, you can bring in ShrinkWrap. Here is an example:

```
public static Archive<?> createTestArchiveAllFiles() {

  EffectivePomMavenDependencyResolver resolver =
  DependencyResolvers.use(MavenDependencyResolver.class).
  loadEffectivePom("pom.xml");

  File[] files = resolver.importAllDependencies().
  resolveAsFiles();

  File[] webinfs = new File
  ("src/main/webapp/WEB-INF").listFiles();
```

```
        WebArchive wa = ShrinkWrap.
        create(WebArchive.class, "test.war")

            .addClass(HelloServlet.class);

        if (webinfs != null) {

          for (File f : webinfs) {

            if (f.isFile())

              wa.addAsWebInfResource(f);

          }

        }

        wa.addAsLibraries(files);

        return wa;

      }
```

You can optionally use a filter on the `import` method to only choose the scopes you're interested in – `compile` and `runtime` in this case. In this setup, you will have tight control over which libraries get pulled in to your deployment WAR file.

Set up your profiles for long term

In most of the profiles in the sample code, you will see a section like the following for the Surefire plugin:

```
<plugin>

  <groupId>org.apache.maven.plugins</groupId>

  <artifactId>maven-surefire-plugin</artifactId>

  <configuration>

    <includes>

      <include>**/junit/**</include>

      <include>**/cdi/**</include>

      <include>**/delegate/**</include>

    </includes>

  </configuration>

</plugin>
```

This tells Surefire which test cases are to be included in the execution. Maven typically only executes every class that ends with `Test` and it will search every package under `src/test/java`. Using this inclusion filter allows you to define which packages are executed with each test phase. The way the sample project is set up is more around which application server you are deploying to. As previously recommended, you should have different profiles for unit testing, integration, acceptance, and any other testing phase you are supporting through Arquillian.

Overlapping profiles

Sometimes it's necessary to define the same test setup in different profiles. You may have some minor differences – JVM options, different test cases, or anything else you feel should be different. I've previously recommended setting up multiple profiles if you need to run only subsets of tests. This can avoid some permgen issues or deployment conflicts that you may need to watch out for. If you do overlap your profiles, they should be named in a way that makes it clear when each is active and which tests are run within the setup.

If you do overlap your profiles, you can run into problems where you expected a test to run but it didn't. This can usually be corrected by changing your inclusion filters (you can also set up exclusion filters, depending on which way is more appropriate for your setup).

When overlapping profiles, it's also important that it's clear to others when to run each. I made a note earlier in the chapter to set up a default profile. Usually, this would be a unit test-level profile, which by definition should not take more than five minutes to run. If you have multiple unit test profiles in your project, you need to have consensus on which is the default one.

The dos and don'ts

These are meant to be some very general do's and don'ts when dealing with the various container types. This is meant to help identify when and how to run tests against a container. These aren't hard requirements, and you should of course plan based on your own needs.

Embedded container dos and don'ts

Embedded containers are generally the easiest to deal with. For the ones that do start up listening ports they are generally random or easily controllable. Do not start up multiple embedded test cases that use common ports. Your tests will be inconsistent and will likely fail.

You should aim to run your unit tests on an embedded container. The cost of starting up an embedded container is typically small. In most cases, troubleshooting problems early on with the embedded containers allows you to find small issues earlier and reduce your overall deployment problems. As a result though, you should use an embedded container that represents your deployment-level container.

Managed containers dos and don'ts

For a number of reasons, I am strongly against managed containers in an automated fashion. Simply put, it's because their results cannot always be recreated. Automated builds should always be reproducible; the results should not vary. Managed containers work well in a controlled environment when kicked off by someone on purpose as a part of a test suite they are monitoring.

Make sure that your managed container is dedicated for your use in testing. Do not use a managed container that is a part of your standard development or testing environment; make sure it will not be in use by anyone else. Looking at this from the Jboss perspective, this can be as simple as creating a new server directory (prior to Jboss AS 7) or creating additional standalone instances in Jboss AS 7. This will help you to avoid cross-deployment issues. It does imply that you need unique port sets for your application. You should strongly consider this for your WebLogic deployments as well. This is less of an issue for Tomcat since you can choose the port easily, and it's the same for TomEE (especially since TomEE will dynamically choose the ports for you).

Do not try to deploy a lot of tests if you only have a little bit of memory available. You may want to avoid running any tests on your local machine if you also have your IDE and appserver running locally, though it depends on the amount of memory available. Your tests may end up failing inappropriately due to limited resources on your local environment. This case may be more appropriate to a development environment and a remote server instance. Start up your normal development server (locally) and simply undeploy your application. Then go ahead and start running the application's test suite and point to the remote server you just started up.

Remote container dos and don'ts

Remote containers are easily managed but can be a problem when running large test suites that have multiple deployments involved. You may need to break up test suites into multiple profiles (almost like a series of batch jobs) that need to be run in some sort of chain. This allows you to restart the container between runs if you need to, or perhaps change some configuration between tests.

If you do run remote containers during an automated test suite (through your Continuous Integration server), make sure this includes a startup before test execution and a shutdown after test execution. Make sure that your listener ports can be configured so that if multiple tests kick off, they do not interfere with one another. This could include environment settings being passed back and forth to make this configuration work.

Summary

As with any framework out there, as you learn Arquillian you are going to have a learning curve as you come up to speed on how it operates and some of its nuances. None of these should be considered blocking issues for you – you should research the errors as presented and try to remedy them.

Next, we're going to get more in depth over creating test cases. We'll review in more depth how to construct your Arquillian test cases and form them into test suites. In the first four chapters we've learned a bit about how Arquillian came to be, as well as being introduced to container-style testing. The remainder of the book focuses on the core testing functionality of Arquillian, eventually leading to information about Arquillian extensions.

5
Enriching the Enterprise Test Case

In the earlier chapters, we reviewed the container setup and problems you may find when working with those containers. This chapter focuses on test cases, and the enrichment that occurs when using Arquillian in those test cases. The enrichment of your test cases is all about the dependency injection that occurs in your test case. Enrichment is the core of using Arquillian. It allows you to inject resources into your test case for use. This chapter focuses around the introductory information for creating tests that use the dependency injection.

Testing the dependency injection

One of the important things to realize when building Arquillian test cases is that your test cases are essentially components within the application server. Arquillian will process the injection points in your code. This goes against the typical runtime of JUnit or TestNG, where the test case operates by itself. You have the container; it's the gift that Arquillian has given you.

This is one of the reasons I strive hard to point out that Arquillian isn't necessarily unit testing, but you're likely performing functional tests, integration tests, or acceptance tests. Your unit tests may rely on being run in a standard JUnit fashion, whereas your testing in Arquillian may require a different setup.

Arquillian by default provides three injection points to your test case, assuming that you are running with a container that supports them:

- `@Inject`: As long as you are running with the CDI implementation and your deployment is a valid bean archive, you will have CDI injection available to you.

Yes I wrote `@Inject` (JSR-330) and CDI (JSR-299) in the same sentence. They are separate specifications but the support container configuration is for CDI with `@Inject` driving injection points. The other JSR-330 supporting containers can be supported (Spring has a separate extension) but CDI is the support platform here.

The way this injection is performed is that your test case is literally part of your bean archive. This allows your CDI container to do necessary injection to your test case. CDI injection is not available if you are running client tests.

- `@EJB`: It provides injection of an EJB into your test case. This can be any type of session bean that can be injected using `@EJB` – stateless, stateful, and singleton all work as a result.
    - Arquillian has a set of JNDI locations that it will look for your EJB in. If the EJB is not in one of these locations the injection point will fail.
- `@EJB` and `@Resource`: Provide injection of a container-managed resource into your test code. With EJB support, this would take the form of a business object you have defined in the JNDI tree. The EJB injection will work in either the EJB containers or the CDI containers; however, CDI containers will not have the transaction support by themselves. This can be anything from a JMS queue to `UserTransaction` or `DataSource`. This can also be custom environment entries that you have defined. The JNDI location specified must be found. As long as it is found it will be injected. Similar to EJB injection, if it is not found it will fail the injection.

  As you may have noticed, these default injection points are the standard injection points when you are working with Java EE 6. If you are targeting a Java EE 5 container you will only have Java EE 5 injection capabilities available (`@EJB` and `@Resource`).

A note about the examples within this chapter

In order to demonstrate all functionality of Arquillian, the tests are being run with full containers – JBoss AS 7, GlassFish, specifically containers that can be pulled down and run easily without additional installs.

`@ArquillianResource` are typically your custom injection points defined by Arquillian core or an Arquillian extension.

CDI injection – simplicity in action

CDI injection leverages the container to populate injection points. As a result, your test cases must obey scope activation. For example, CDI Request, Session, and Conversation scopes are only active based on how your application is accessed. For example, if you use Servlet 3.0 protocol against a Java EE 6 container any `@RequestScoped` object defined will be activated. You do need to be careful when testing `@RequestScoped` objects then. Consider this example, of a simple CDI-based controller object:

```
@Model
public class DataInputController {
  private Logger logger = Logger.getLogger(DataInputController.class.
  getCanonicalName());
  private String name;
  private String value;
  public String processInput() {
    logger.info(String.format("Received name %s and value
    %s",name,value));
    return "/pages/input.xhtml";
  }
  //.. getters and setters omitted.
}
```

If we want to test this example, we would build the following test case:

```
@RunWith(Arquillian.class)
public class DataInputControllerTest {
  @Deployment
  public static JavaArchive createTestArchive() {
  return ShrinkWrap.create(JavaArchive.class)
      .addClass(DataInputController.class)
      .addAsManifestResource(EmptyAsset.INSTANCE, "beans.xml");
  }

  @Inject
  private DataInputController controller;

  @Test
  public void testController() {
    controller.setName("xxx");
    controller.setValue("yyy");
    String result = controller.processInput();
    assertEquals("/pages/input.xhtml",result);
  }
}
```

When this test case is run in different containers, you will see different results. If you run it with the Weld EE and JBoss AS 7 profiles, you will see that the tests pass without any issue. If you change the JBoss AS 7 profile to use JMX instead of Servlet, it will fail. Likewise, if you run this against Weld SE or OpenWebBeans, you will receive an error along the lines of `No active contexts for scope type javax.enterprise.context.RequestScoped`. What this means is that the request scope is not active. Similar problems will occur if you use session-scoped or conversation-scoped objects. To avoid this, you should test objects that are bound to HTTP-oriented contexts within HTTP-based containers and make sure you are using HTTP protocols to execute them.

One curious thing to point out, merely because of how the specification is written, is that if you use `@EJB` within your CDI-only container, it will still be injected for you. This is because CDI can handle EJB injection for you. This includes injection of `@PersistenceUnit` and `@PersistenceContext`. Since CDI can handle this for you, CDI is essentially a power tool that Arquillian can use to handle all injection cases. Your test case is a CDI bean deployed to the container; as a result, it's allowing the container to handle all the injection points. As you can see, the focus here is CDI. To demonstrate this, suppose we create a very simple greeter EJB:

```
@Stateless
public class GreeterService {
  public String sayHello(String name) {
    return String.format("Hello, %s!",name);
  }
}
```

Next, let's say we have created an Arquillian test case that tests these two ways – using EJB injection and using CDI injection:

```
@RunWith(Arquillian.class)
public class GreeterServiceTest {
  private static Asset BEANS = EmptyAsset.INSTANCE;
  @Deployment
  public static JavaArchive createTestArchive() {
    return ShrinkWrap.create(JavaArchive.class)
        .addClass(GreeterService.class)
        .addAsManifestResource(BEANS, "beans.xml");
  }

  @Inject
  private GreeterService greeterBean;
```

```
    @EJB
    private GreeterService greeterService;

    private void testGreeter(GreeterService greeter) {
      String name = "Dave";
      assertEquals("Hello, Dave!",greeter.sayHello(name));
    }

    @Test
    public void testGreeterBean() {
      testGreeter(greeterBean);
    }

    @Test
    public void testGreeterService() {
      testGreeter(greeterService);
    }

    @Test
    public void testInjectedIndependent() {
      assertFalse(greeterService == greeterBean);
    }
  }
```

As we can see, there are two test cases that use different injection points and styles and both work the same. In fact, we validate that the container injected different objects to both as a separate test case.

Next, let's say we want to test out a JAX-RS resource that is using CDI injection. As you are likely aware, JAX-RS allows CDI beans to be defined as resources as long as they are appropriately scoped – this applies only to `@RequestScoped` and `@ApplicationScoped` resources, which makes sense from a usage standpoint: JAX-RS is stateless; it doesn't make sense for a `@SessionScoped` object to work. We can create a resource that uses this defined EJB to do the work (note that you could also annotate the EJB as a JAX-RS resource in this case).

```
@Path("/greeter")
@RequestScoped
public class GreeterResource {
  @EJB
  private GreeterService greeter;
```

```
    @GET
    @Produces("text/plain")
    @Path("/{name}")
    public String greeter(@PathParam("name") String name) {
      return greeter.sayHello(name);
    }
  }
```

So now our resource is defined. However, let's say we want to test it without needing to worry about reading the HTTP URL and processing data; instead, we just want to call the object in more of a unit test setup and do work against it like any other managed bean. The following test case does just that:

```
  @RunWith(Arquillian.class)
  public class GreeterResourceTest {
    @Deployment
    public static JavaArchive createTestArchive() {
      return GreeterServiceTest.createTestArchive()
          .addClass(GreeterResource.class);
    }
    @Inject
    private GreeterResource resource;

    @Test
    public void testResource() {
      assertEquals("Hello, Dave!",resource.greeter("Dave"));
    }
  }
```

So, as you can see, we start off by reusing the deployment archive from the prior test case. Since our resource depends on the archive created there, it allows us to aim for code reusability. All we have to do is add our resource to the deployment archive. We then create a standard CDI test case in Arquillian and validate that the resource works as expected.

Using this test case, we can also do a more end-to-end test case that involves deploying our rest resource and validating that the contents come back as expected. We'll do this in greater detail when working with the Arquillian Warp/Drone extensions, but for now we'll use a simple web page scraping mechanism to validate the contents of a page.

First, we need to create an `@ApplicationPath` resource in the bundle. To do this we'll add the following class:

```
@ApplicationPath("/resources")
public class JaxRsActivator extends Application {
}
```

This simply forces JAX-RS activation within your application. It needs to be included in the deployment archive. Then, we'll update our deployment to include the following class:

```
@Deployment
public static WebArchive createTestArchive() {
  return ShrinkWrap.create(WebArchive.class,"test.war")
      .addClasses(GreeterService.class,JaxRsActivator.
       class,GreeterResource.class)
      .addAsWebInfResource(EmptyAsset.INSTANCE, "beans.xml");
}
```

Finally, we'll create our test case that reads the resource:

```
@Test
@RunAsClient
public void testResourceInjection(@ArquillianResource URL
baseUrl) throws Exception {
  URL url = new URL(baseUrl,"/test/resources/greeter/Dave");
  InputStream is = url.openStream();
  BufferedReader br = new BufferedReader
  (new InputStreamReader(is));
  String result = br.readLine();
  String expected = "Hello, Dave!";
  assertEquals(expected, result);
}
```

This test case does look a bit different as compared to the other test cases. The first difference is the inclusion of the annotation `@RunAsClient` – this tells Arquillian to run this test case in the source JVM (in this case, the JVM that Maven/Surefire is running from). The second is a new form of enrichment, `@ArquillianResource`. In this case, the `@ArquillianResource` (when annotated on a URL) represents the deployment URL. We can use that `baseUrl` to add the expected deployment path. That path will take the form of `/test/resources/greeter`, where test is the deployment context (`test.war`), resources is the `ApplicationPath` in the `JaxRsActivator` class, and the greeter path is `GreeterResource`. Finally, we have the value of the parameter.

Arquillian can inject `@ArquillianResource` objects whenever we are running in the client JVM. This injection strategy is not available when running on the server's JVM. They are specific to the Arquillian runtime and are meant to be able to connect your client JVM to the server's JVM.

As you have seen, CDI scopes are active during your testing. This is dependent on certain configurations. In the case of JBoss AS 7, this means that the servlet protocol must be used for your testing (this is already enabled in the Maven profile provided with the code examples). This should work out of the box in Glassfish.

There are certain things you cannot do here without some additional configuration; the first is testing against a CDI `@ConversationScoped` beans. This scope is specific to JSF and requires some server-level components to validate the conversation as a result.

We've spent this section thinking about CDI. What about JSF? Are JSF contexts or objects in scope here? Unfortunately not, at least not without invoking a JSF page during your test. One approach you could take is to include a JSF page within your deployed WAR file and invoke that page during your test to validate CDI components. In *Chapter 7*, *Functional Application Testing*, we will review some of the capabilities of testing JSF applications with Arquillian using tools such as Drone and Warp.

Testing external extensions

CDI supports the creation of external extensions; they can be provided in other JAR files that are deployed with your code. ShrinkWrap provides a resolver API for loading dependencies from Maven projects. This can automatically include any dependencies that those libraries have.

JBoss Solder provides a number of extensions like this. We can test that beans are properly vetoed from being loaded in to the CDI context by verifying against the Veto Extension in Solder. We can define a very simple Veto Bean in our code:

```
@Veto
public class VetoBean {
  public String doWork() {
    return "Work was done.";
  }
}
```

Then we define a test case that verifies that the bean is not injected. We do this using an Instance object that should contain the `VetoBean` class.

```
@RunWith(Arquillian.class)
public class VetoBeanTest {
  @Deployment
  public static WebArchive createTestArchive() {
    File[] solder = Maven.resolver().loadPomFromFile("pom.xml")
      .resolve("org.jboss.solder:solder-impl")
      .withMavenCentralRepo(false).withTransitivity().
      as(File.class);
      return ShrinkWrap.create(WebArchive.class)
        .addClass(VetoBean.class)
        .addAsLibraries(solder)
        .addAsWebInfResource(EmptyAsset.INSTANCE, "beans.xml");
  }

  @Inject
  private Instance<VetoBean> vetodInstance;

  @Test
  public void validateBeanNotInstalled() {
    assertTrue(vetodInstance.isUnsatisfied());
  }
}
```

We can also validate that the extension is working as expected, since the following log statement is written:

```
INFO  [org.jboss.solder.core.CoreExtension] (MSC service thread 1-11)
Preventing class com.tad.arquillian.chp5.cdi.VetoBean from being
installed as bean due to @Veto annotation
```

Using this resolver API we can see that the following JARs get added, even though we only declared usage of the `solder-impl` dependency:

```
/WEB-INF/lib/solder-logging-3.1.1.Final.jar
/WEB-INF/lib/solder-impl-3.1.1.Final.jar
/WEB-INF/lib/solder-api-3.1.1.Final.jar
```

This makes it overall easier to pull in dependencies than trying to manipulate individual JAR files. This type of setup is easily reusable for any extensions you may use in your application.

Building and testing extensions

Let's say we were building a CDI extension that helped us fix some issue during development. In this case, whenever we encounter a JAX-RS resource that is annotated `@SessionScoped`, we veto the bean and write a logger message indicating as such. First, we define the class in our `META-INF/services/javax.enterprise.inject.spi.Extension` as follows:

```
com.tad.arquillian.chp5.cdi.ext.SessionResourceValidationExtension
```

Then, we create the implementation of this class:

```
public class SessionResourceValidationExtension implements Extension {
  private Logger log = Logger.getLogger
  (SessionResourceValidationExtension.
  class.getCanonicalName());

  <X> void processAnnotatedType
  (@Observes final ProcessAnnotatedType<X> pat) {
    final AnnotatedType<X> annotatedType = pat.getAnnotatedType();
        final Class<X> javaClass = annotatedType.getJavaClass();
        if (annotatedType.isAnnotationPresent
        (SessionScoped.class) &&
            annotatedType.isAnnotationPresent(Path.class)
            ) {
          log.warning("Not loading bean "+javaClass.getCanonicalName()
          + " SessionScoped and JAX-RS Path - not compatible");
          pat.veto();
        }
  }
}
```

That exists under our `src/main/java`. Then in our `src/test/java` directory structure, we create one of these invalid beans:

```
@SessionScoped
@Path("/location")
public class InvalidSessionBean implements Serializable {
  @GET
  @Produces("text/plain")
  public String getValue() {
    return getClass().getCanonicalName();
  }
}
```

Then finally, we create the test case:

```
@RunWith(Arquillian.class)
public class SessionValidationExtensionTest {
  @Deployment
  public static JavaArchive createTestArchive() {
    return ShrinkWrap.create(JavaArchive.class)
        .addClasses(SessionResourceValidationExtension.
         class,InvalidSessionBean.class)
        .addAsResource("META-INF/services/javax.enterprise.
         inject.spi.Extension")
        .addAsManifestResource(EmptyAsset.INSTANCE, "beans.xml");
  }

  @Inject
  private Instance<InvalidSessionBean> sessionBeanInstance;

  @Test
  public void testNotResolved() {
    assertTrue(sessionBeanInstance.isUnsatisfied());
  }
}
```

This deployment structure is a little different. First, we need to add an extra resource to our package. We add it as a regular resource to the JAR file, not necessarily as a manifest resource. Since we read the asset from the filesystem, we use the full path to the file. Note that the file used here is actually from `src/test/resources`, not `src/main/resources`. If you want to use `src/main/resources`, you will need to add that to your test build path.

We can add some additional testing logic to our extension if we want. If we change the extension definition to provide a way to get the collection of vetoed beans, we could simply change the extension as such:

```
public class SessionResourceValidationExtension implements Extension {
  private Logger log = Logger.getLogger
  (SessionResourceValidationExtension.class.getCanonicalName());
  private Set<Class<?>> vetodClasses = new HashSet<Class<?>>();
  <X> void processAnnotatedType
  (@Observes final ProcessAnnotatedType<X> pat) {
    final AnnotatedType<X> annotatedType = pat.getAnnotatedType();
        final Class<X> javaClass = annotatedType.getJavaClass();
```

```
            if (annotatedType.isAnnotationPresent
            (SessionScoped.class) &&
                annotatedType.isAnnotationPresent(Path.class)
                ) {
              log.warning("Not loading bean "+javaClass.
              getCanonicalName() + " SessionScoped and
              JAX-RS Path - not compatible");
              pat.veto();
              vetodClasses.add(javaClass);
            }
      }
      public Set<Class<?>> getVetodClasses() {
        return vetodClasses;
      }
    }
```

This will keep track of all classes veteod via this extension. We then add this to our test case:

```
      @Inject
      private SessionResourceValidationExtension extension;

      @Test
      public void testExtensionResults() {
        Set<Class<?>> contents = extension.getVetodClasses();
        assertTrue("Wrong number of beans veto'd",
        contents.size() == 1);
        Class<?> vetod = contents.iterator().next();
        assertEquals("Wrong class veto'd",InvalidSessionBean.class,vetod);
      }
```

The extensions become injectable resources during runtime; this is true even when running outside of Arquillian. As a result, as long as we keep track of what actions are performed we can validate that the contents of the extension match what we expect in the test case.

EJB and resource injection

These injection patterns are very similar so it makes sense to speak about them together. Both the injection patterns are based on the JNDI lookup that Arquillian will perform to find the object. The difference is that your test case is not an EJB itself; injection is just occurring against your component as if it were a ManagedBean.

Special considerations for EJB deployments

Singleton EJBs support a `@Startup` annotation. If this EJB is included in your deployment then it will be executed with your deployment. If your singleton does a lot of work – for example, it upgrades database objects on startup, validates system integrity, you should factor this in when testing using this type of object. Otherwise, if you're doing something along the lines of following the `singleton-dao` pattern, you should be fine.

Testing persistence

Testing JPA over EJB is a very powerful approach. Since you can deploy persistence units with your unit tests, you can perform tests that drop and recreate your database schema, create new entries, and then perform actions against them. The Arquillian persistence extension will be discussed in more detail in *Chapter 6*, *Arquillian Extensions*, since it includes DBUnit integration and more in-depth coverage.

JBoss AS 7 ships with H2, and we'll use that for demonstration purposes. First, we'll need to add a new datasource to our application server to run our application with. Edit the `standalone.xml` file and add this to your datasources section after the example datasource:

```
<datasource jndi-name="java:/jdbc/AppDS"
pool-name="AppDS" enabled="true" use-java-context="false">
    <connection-url>jdbc:h2:mem:app;DB_CLOSE_DELAY=-1
    </connection-url>
    <driver>h2</driver>
    <security>
        <user-name>sa</user-name>
        <password>sa</password>
    </security>
</datasource>
```

Also, we'll need to add a `persistence.xml` file to our project. Here's a simple one:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
  xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
  http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
  <persistence-unit name="arquillian-chapter5">
    <jta-data-source>java:/jdbc/AppDS</jta-data-source>
    <class>com.tad.arquillian.chp5.jpa.Book</class>
    <properties>
```

```
        <property name="hibernate.hbm2ddl.auto"
        value="create-drop" />
        <property name="hibernate.show_sql" value="true"/>
      </properties>
    </persistence-unit>
  </persistence>
```

Next, let's define the entity and service classes that will be used in the test case:

```
@Entity
@Table(name="BOOKS")
public class Book implements Serializable {
  @Id @Column(name="ID")
  private long id;
  @Column(name="NAME")
  private String name;
  @Column(name="TITLE")
  private String title;
  @Temporal(TemporalType.TIMESTAMP)
  @Column(name="CREATE_DATE")
  private Date createDate;
  //getters & setters omitted for brevity
}
@Stateless
public class BookService {
  @PersistenceContext
  private EntityManager em;
  public void save(Book book) {
    em.merge(book);
  }
  public Book find(long id) {
    return em.find(Book.class,id);
  }
  public List<Book> searchByName(String name) {
    String input = "%"+name.toUpperCase()+"%";
    List<Book> books = em.createQuery("select b
    from Book b where UPPER(b.name) LIKE :input order by b.id")
    .setParameter("input", input)
    .getResultList();
    return books;
  }
}
```

Now if we want to test this, we have to be careful. Let's suppose we want to populate the table, then run some test cases against it, and perform them within separate methods. JUnit does not guarantee test method execution order. As a result, you will need to invoke the methods yourself to run the full set of test cases. See the following test scenario for instance:

```
@RunWith(Arquillian.class)
public class BookServiceTest {
  @Deployment
  public static JavaArchive createTestArchive() {
  return ShrinkWrap.create(JavaArchive.class)
      .addClasses(Book.class,BookService.class)
      .addAsManifestResource(EmptyAsset.INSTANCE, "beans.xml")
      .addAsManifestResource("persistence.xml");
  }

  @EJB
  private BookService service;

  @Test
  public void testCreateBooks() {
    Book b1 = new Book();
    b1.setId(1l);
    b1.setName("Book One");
    b1.setTitle("Arquillian Testing Guide");
    b1.setCreateDate(new Date());
    service.save(b1);
    Book b2 = new Book();
    b2.setId(2l);
    b2.setName("Book Two");
    b2.setTitle("Hitchhiker's Guide to the Universe");
    b2.setCreateDate(new Date());
    service.save(b2);

    testFindB1();
    testFindByNameMulti();
    testFindABookOne();
    testFindABookTwo();
  }
```

```
    public void testFindB1() {
      Book b1 = service.find(1l);
      assertEquals("Book One",b1.getName());
      assertEquals("Arquillian Testing Guide",b1.getTitle());

      Book b2 = service.find(2l);
      assertEquals("Book Two",b2.getName());
      assertEquals("Hitchhiker's Guide to the
      Universe",b2.getTitle());
    }

    public void testFindByNameMulti() {
      List<Book> books = service.searchByName("Book");
      assertEquals(2,books.size());
    }

    public void testFindABookOne() {
      List<Book> books = service.searchByName("One");
      assertEquals(1,books.size());
      Book b = books.get(0);
      assertEquals("Book One",b.getName());
    }

    public void testFindABookTwo() {
      List<Book> books = service.searchByName("Two");
      assertEquals(1,books.size());
      Book b = books.get(0);
      assertEquals("Book Two",b.getName());
    }
  }
```

Ideally, you would not depend on one test case to ensure that the next runs. For simplicity, we simply delegate the validation logic to other methods for testing purposes. This does reduce our number of test methods but the code coverage should still look good.

The test archive does not look all that different compared to other test cases. We define a similar structure but ensure that we include our `Book` class, `BookService` class, as well as a `persistence.xml` file. It will pull `persistence.xml` from the `src/test/resources` directory (instead of `src/main/resources`) since it's part of the test classpath. We do enable SQL output to help with any troubleshooting, so you will see the following statements run during your testing:

```
INFO  [stdout] (MSC service thread 1-5) Hibernate: drop table BOOKS if
exists
```

```
INFO  [stdout] (MSC service thread 1-5) Hibernate: create table BOOKS
(ID bigint not null, CREATE_DATE timestamp, NAME varchar(255), TITLE
varchar(255), primary key (ID))
INFO  [stdout] (pool-4-thread-1) Hibernate: select book0_.ID as ID0_0_,
book0_.CREATE_DATE as CREATE2_0_0_, book0_.NAME as NAME0_0_, book0_.TITLE
as TITLE0_0_ from BOOKS book0_ where book0_.ID=?
INFO  [stdout] (pool-4-thread-1) Hibernate: insert into BOOKS (CREATE_
DATE, NAME, TITLE, ID) values (?, ?, ?, ?)
....
INFO  [stdout] (pool-4-thread-1) Hibernate: select book0_.ID as ID0_0_,
book0_.CREATE_DATE as CREATE2_0_0_, book0_.NAME as NAME0_0_, book0_.TITLE
as TITLE0_0_ from BOOKS book0_ where book0_.ID=?
....
INFO  [stdout] (pool-4-thread-1) Hibernate: select book0_.ID as ID0_,
book0_.CREATE_DATE as CREATE2_0_, book0_.NAME as NAME0_, book0_.TITLE
as TITLE0_ from BOOKS book0_ where upper(book0_.NAME) like ? order by
book0_.ID
INFO  [stdout] (MSC service thread 1-3) Hibernate: drop table BOOKS if
exists
```

This helps to ensure that the queries are the way we expected.

Another step in validation that can occur in our test case is to run queries directly against the entity manager. Since your test case is any other component in your application server, you can inject the persistence context into it. We can add an extra validation step in our test case as a result:

```
...
  @PersistenceContext
  private EntityManager em;
...
    service.save(b2);

    validateLoaded(1l);
    validateLoaded(2l);

    testFindB1();
...
  private void validateLoaded(long id) {
    Book b = em.find(Book.class,id);
    assertNotNull(b);
  }
```

This snippet, when added to our test case, will show that the entity was correctly stored in the database and can be run before we do any of our database queries.

Testing Session beans

Testing Session beans is the same as testing CDI beans: you can inject them, do work with them, call methods on them, and validate results. Now that we have a datasource defined in our application server, let's take that `DatabaseService` CDI bean from earlier and use it as an EJB. First, here's the EJB implementation. The only difference is the JNDI location:

```
@Stateless
@LocalBean
public class DatabaseIntegrator {
  @Resource(mappedName="java:/jdbc/AppDS")
  private DataSource ds;
  private Connection conn;
  @PostConstruct
  public void init() throws SQLException {
    this.conn = ds.getConnection();
  }
  @PreDestroy
  public void cleanup() throws SQLException {
    this.conn.close();
  }
  public void runQuery(String query) throws SQLException {
    this.conn.createStatement().execute(query);
  }
}
```

The usage of `@LocalBean` is optional, but recommended. This is a no-interface view of an EJB. The use of this annotation makes it clear to other developers—as well as yourself six months later—how this becomes an EJB. Next we create a test case – again, fairly simple:

```
@RunWith(Arquillian.class)
public class DatabaseIntegratorTest {
  @Deployment
  public static JavaArchive createTestArchive() {
    return ShrinkWrap.create(JavaArchive.class)
        .addClass(DatabaseIntegrator.class)
        .addAsManifestResource(EmptyAsset.INSTANCE, "beans.xml");
  }
```

```
    @EJB
    private DatabaseIntegrator dbIntegrator;

    @Test
    public void testDbServiceInjectionFailure() throws Exception{
      dbIntegrator.runQuery("select 1");
    }
}
```

We'll leverage CDI's `beans.xml` here still. This will make this a bean archive and deploy the EJBs contained to the EJB container. This is easier than trying to create an `ejb-jar.xml` file for deployments.

When the test case runs, you'll notice the following JNDI entries being printed out:

```
java:global/def20cc5-fcaf-4e39-ba2f-f6092a874d4c/DatabaseIntegrator!com.
tad.arquillian.chp5.ejb.DatabaseIntegrator

java:app/def20cc5-fcaf-4e39-ba2f-f6092a874d4c/DatabaseIntegrator!com.tad.
arquillian.chp5.ejb.DatabaseIntegrator

java:module/DatabaseIntegrator!com.tad.arquillian.chp5.ejb.
DatabaseIntegrator

java:global/def20cc5-fcaf-4e39-ba2f-f6092a874d4c/DatabaseIntegrator

java:app/def20cc5-fcaf-4e39-ba2f-f6092a874d4c/DatabaseIntegrator

java:module/DatabaseIntegrator
```

The obscure name is actually a random UUID being assigned to the archive. You can change this in your deployment methods by specifying a name for the deployment; call the following:

```
ShrinkWrap.create(JavaArchive.class,"test.jar")
```

Instead of this:

```
ShrinkWrap.create(JavaArchive.class)
```

You'll notice that half of the six locations are the EJB standard locations as defined by Java EE 6. The other half will represent those same locations, except with pointers to the specific implementation class.

Your Arquillian component will essentially be an EJB client (similar to a client-jar). You should consider this fact when thinking about transaction state and propagation within your test case.

Working with Remote Session Beans

Let's say your application is structured in a multi-tiered environment. Your frontend may be a Struts or JSF application running on Tomcat and connects to a remote EJB container, maybe JBoss or WebLogic. The EJB tier performs all database interaction for you. Arquillian can test these remote EJBs, and actually in two different ways. The first assumes that we're running on the server and as a result we can access the server resources directly. We'll create our EJB and interfaces as follows:

```
@Remote
public interface RemoteDatabaseIntegrator {
  public void runQuery(String query) throws SQLException;
}

@Remote
@Stateless
public class RemoteDatabaseIntegratorImpl implements
RemoteDatabaseIntegrator {
//implementation skipped for brevity, see DatabaseIntegrator
}
```

Then we create the same test case as with `DatabaseIntegrator`, only with a different injection point and a slightly different artifact structure:

```
@RunWith(Arquillian.class)
public class RemoteDatabaseIntegratorTest {
  @Deployment
  public static JavaArchive createTestArchive() {
    return ShrinkWrap.create(JavaArchive.class,"test.jar")
        .addClasses(RemoteDatabaseIntegrator.
         class,RemoteDatabaseIntegratorImpl.class)
  .addAsManifestResource(EmptyAsset.INSTANCE, "beans.xml");
  }

  @EJB
  private RemoteDatabaseIntegrator dbIntegrator;

  @Test
  public void testDbServiceInjectionFailure() throws Exception{
    dbIntegrator.runQuery("select 1");
  }
}
```

You will see that the test case operates the same way as the local version did. The other way we can test this is in client mode. When running against a remote container, you can test based on some additional injection points. These injection points can differ. For JBoss AS 7 you can inject a remote `InitialContext` to perform lookups against. For TomEE, you can inject a URL to the deployment and use that information to construct the initial context. Here is how to do it with TomEE:

```
@RunWith(Arquillian.class)
@RunAsClient
public class RemoteDatabaseIntegratorTest {
  @Deployment(testable=false)
  public static JavaArchive createTestArchive() {
    return ShrinkWrap.create(JavaArchive.class,"test.jar")
        .addClasses(RemoteDatabaseIntegrator.
         class,RemoteDatabaseIntegratorImpl.class);
  }

  @ArquillianResource
  private URL deploymentURL;

  private static final String JNDI_LOC =
  "RemoteDatabaseIntegratorImplRemote";

  @Test
  public void testDbServiceInjectionFailure() throws Exception{
    Properties p = new Properties();
    p.put("java.naming.factory.initial",
    "org.apache.openejb.client.RemoteInitialContextFactory");
    String providerUrl = String.format("http://%s:%s/tomee/ejb",
    deploymentURL.getHost(),deploymentURL.getPort());
    p.put("java.naming.provider.url", providerUrl);
    Context context = new InitialContext(p);
    RemoteDatabaseIntegrator dbIntegrator =
    (RemoteDatabaseIntegrator)context.lookup(JNDI_LOC);
    dbIntegrator.runQuery("select 1");
  }
}
```

You can expect that this problem will be addressed in later TomEE releases, since this is written against TomEE 1.5. In this example, we simply need to build `InitialContext`. It turns out that JNDI lookups in TomEE are performed over HTTP, so we simply need to insert the host and port when doing the lookup. Other containers will support the injection point `@ArquillianResource private Context context;` to represent the remote JNDI lookup.

Failure scenarios

Since Arquillian is deploying an artifact to your container for you, it stands to reason that some level of module packaging problems can be validated by Arquillian. Since ShrinkWrap builds based on a programmatic structure, rather than using your Maven project's build structure, it may not be a 100 percent match for your tests compared to your real deployment. Of course, you can use some ShrinkWrap APIs to point to your already-created file and use that if you choose. This can include problems with your object definitions.

Dependency failures

Since we are handling dependency injection, we can handle failures as a result of attempting to activate beans. This does take a slightly different approach; for one, you need to ensure that you work against instances and look ups against them rather than directly injecting beans. If you're not familiar yet with the `Instance` object of CDI, it is a placeholder object that allows you to look up beans more dynamically–it allows you to select based on qualifiers and get an instance based on what you have configured.

Let's suppose we build a really simple database service–it connects to the database and exposes methods to run queries (this is essentially very dangerous, but meant to prove a point).

```
@RequestScoped
public class DatabaseService {
  @Resource(mappedName="jdbc/MyDB")
  private DataSource ds;
  private Connection conn;
  @PostConstruct
  public void init() throws SQLException {
    this.conn = ds.getConnection();
  }
  @PreDestroy
  public void cleanup() throws SQLException {
    this.conn.close();
  }
  public void runQuery(String query) throws SQLException {
    this.conn.createStatement().execute(query);
  }
}
```

If we attempt to test this when a database connection was not deployed, we should expect a runtime deployment exception being thrown. This is similar to regular JUnit tests where we can tell JUnit to expect an exception to be thrown. It still applies in Arquillian. If we create our test archive without the datasource, we can still check if the database connection being missing still throws an exception:

```
@RunWith(Arquillian.class)
public class DatabaseServiceTest {
  @Deployment
  public static JavaArchive createTestArchive() {
    return ShrinkWrap.create(JavaArchive.class)
        .addClass(DatabaseService.class)
        .addAsManifestResource(BEANS, "beans.xml");
  }

  @Inject
  private Instance<DatabaseService> dbServiceInstance;

  @Test(expected=RuntimeException.class)
  public void testDbServiceInjectionFailure() throws Exception{
    dbServiceInstance.get().runQuery("select 1");
  }
}
```

If you look at `DatabaseService`, you'll see that running queries can throw `SQLExceptions`, but we validate that a `RuntimeException` is what we expect to have returned. If you look at the `RuntimeException` passed back, you'll see that it's actually wrapping a CDI exception because the datasource does not exist. This is not the way to do this with a plain member-level injection but we can verify this with method argument injection, based on the following test case:

```
  @Test(expected=RuntimeException.class)
  public void testDbServiceInjectionFailure
    (Instance<DatabaseService> dbServiceInstance)
    throws Exception{
    dbServiceInstance.get().runQuery("select 1");
  }
```

Likewise, this case where the exact service is injected (not an Instance object) can still throw the same type of runtime exception:

```
  @Test(expected=RuntimeException.class)
  public void testDbServiceInjectionFailure
  (DatabaseService dbService) throws Exception{
    dbService.runQuery("select 1");
  }
```

This gives a lot of flexibility when testing failure cases with your bean setup. One downside to this problem is that it may not catch every problem properly. Since the real exception thrown is `RuntimeException`, you may not be able to catch `NullPointerExceptions` during your test cases correctly. As a result, you should only use this type of structure for debugging/troubleshooting purposes. You should not use this long term for your builds (unless you really do expect a `RuntimeException` to be thrown based on some specific cases). You should ideally use your application's exception stack to test.

Introduction to Arquillian extensions

Besides the core injection capabilities of Arquillian, a number of extensions exist that can provide additional injection points within your test cases.

Framework extensions

Several extensions exist to support injection of more proprietary injection technologies. This focuses around Spring, since it is a popular framework and will be used when Warp is reviewed in *Chapter 7*, *Functional Application Testing*.

Spring

In order to demonstrate Spring integration, a separate project exists; take a look at the `arquillian-chapter5-spring` project to see the example code. The Arquillian Spring Extension supports annotation-driven and programmatic bootstrapping of Spring as well as classic XML-based configuration. This gives you better diversity when building your archive. For the sake of these examples, the container in use is Tomcat 6 managed, and you should have `CATALINA_HOME` set prior to running the examples. This allows us to separate the classpath entries from the deployed container.

First, let's look at what can be placed in your `arquillian.xml` file:

```
<extension qualifier="spring">
    <property name="customContextClass">
    org.springframework.context.support.
    ClassPathXmlApplicationContext</property>
    <property name="customAnnotationContextClass">
    org.springframework.context.annotation.
    AnnotationConfigApplicationContext</property>
</extension>
```

Neither of these properties are required. You may want to or need to override these to ensure application compatibility. For example, older versions of JBoss require the use of their own Spring deployer and that needs to be noted here. If not present, the default values specified here will be used. This allows you to, if needed, change the default context class as well as the annotation context class. You may be interested in doing this if you want to use filesystem-based configuration instead of classpath, or based on configuration in a `web.xml` file. In `arquillian.xml` we'll also need to add our Tomcat settings:

```
<container qualifier="tomcat" default="true">
  <configuration>
  <property name="user">tomcat</property>
  <property name="pass">tomcat</property>
  </configuration>
</container>
```

This will tell Arquillian what user to connect to in Tomcat, since we have to call the deploy API over HTTP, which requires authentication.

So now let's dig into our Spring-based application. It's a very simple bean setup: we have a `UserProvider` interface which represents our database connection. We'll have a simplistic implementation that has a hard coded list of values, as well as a POJO representing the model objects:

```
public interface UserProvider {
  public List<User> findAllUsers();
}
```

We have our implementation, which is annotated as a component in Spring:

```
@Component
public class BasicUserProvider implements UserProvider {
  @Override
  public List<User> findAllUsers() {
    User[] users = new User[]{ new User("booboo","Boo Boo"), new 
User("foo","Foo Bar") };
    return Arrays.asList(users);
  }
}

public class User {
  private String username;
  private String realname;
  //getters, setters, constructors omitted
}
```

The Spring extension supports configuration in different ways depending on how your application is built. I'll show two of those styles: via a `Configuration` class and via an `applicationContext.xml`.

To do our programmatic style setup, we'll need one additional class, `ConfigBuilder` in this case. This provides beans based on what should be in the container.

```
@Configuration
public class ConfigBuilder {
  @Bean
  public UserProvider createUserProvider() {
    return new BasicUserProvider();
  }
}
```

Next, we'll create a utility class that contains most of our deployment structure:

```
public class SpringTestUtils {
  public static File[] getSpringDependencies() {
    return Maven
        .resolver()
        .loadPomFromFile("pom.xml")
        .resolve("cglib:cglib", "org.springframework:
         spring-context",
            "org.springframework:spring-web")
        .withMavenCentralRepo(false).withTransitivity().
         as(File.class);
  }

  public static WebArchive createTestArchive() {
    return ShrinkWrap
        .create(WebArchive.class, "test.war")
        .addClasses(User.class, BasicUserProvider.class,
            UserProvider.class)
        .addAsLibraries(getSpringDependencies());
  }
}
```

The first method uses ShrinkWrap's Resolver API to load the necessary dependencies to run Spring. `Maven.resolver()` gives you the resolver and we can point to our local `pom.xml`. We then tell the resolver to resolve CGLIB, SpringContext, and SpringWeb as files. This will include additional dependencies as needed. As a result, these are the libraries brought in:

```
/WEB-INF/lib/spring-aop-3.1.1.RELEASE.jar
/WEB-INF/lib/spring-beans-3.1.1.RELEASE.jar
/WEB-INF/lib/spring-context-3.1.1.RELEASE.jar
/WEB-INF/lib/cglib-2.2.jar
/WEB-INF/lib/aopalliance-1.0.jar
/WEB-INF/lib/spring-core-3.1.1.RELEASE.jar
/WEB-INF/lib/spring-expression-3.1.1.RELEASE.jar
/WEB-INF/lib/spring-asm-3.1.1.RELEASE.jar
/WEB-INF/lib/commons-logging-1.1.1.jar
/WEB-INF/lib/spring-web-3.1.1.RELEASE.jar
/WEB-INF/lib/asm-3.1.jar
```

Finally, let's see what our test case looks like:

```
@RunWith(Arquillian.class)
@SpringAnnotationConfiguration(classes = {ConfigBuilder.class})
public class AnnotationUserProviderTest {
  @Deployment
    public static WebArchive createTestArchive() {
        return SpringTestUtils.createTestArchive().
        addClass(ConfigBuilder.class);
    }
  @Autowired
    private UserProvider userProvider;
  @Test
  public void testInjection() {
    assertNotNull(userProvider);
  }

  @Test
  public void testUsersContent() {
    List<User> users = userProvider.findAllUsers();
    assertEquals(2,users.size());
  }
}
```

The first line should look very familiar: we tell JUnit to run this test using Arquillian. Next, specific to the Spring extension, we indicate what should be used as the configuration class. We also have our normal deployment method, which uses the utility method but also adds the configuration class to the archive. From there on it looks like a typical test case, except that our `@Inject` will be `@Autowired`. Arquillian will enrich this test case to include injection of Spring beans into the class.

The other version of this test case includes a Spring `applicationContext.xml` file. We can tell Spring to just look for everything under certain packages to ensure injection occurs, using the following code snippet:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/
       schema/context"
       xsi:schemaLocation="http://www.springframework.org/
       schema/beans http://www.springframework.org/schema/beans/
       spring-beans-3.0.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/
       spring-context-3.1.xsd
       http://www.springframework.org/schema/tool
       http://www.springframework.org/schema/tool/
       spring-tool-3.1.xsd
       http://www.springframework.org/schema/
       mvc http://www.springframework.org/schema/mvc/
       spring-mvc-3.1.xsd">
  <context:component-scan base-
  package="com.tad.arquillian.chp5.spring"/>
</beans>
```

Next, we look at the test case that will run the XML version:

```
@RunWith(Arquillian.class)
@SpringConfiguration({ "test-userProviderContext.xml" })
public class XMLUserProviderTest {
  @Deployment
  public static WebArchive createTestArchive() {
    return SpringTestUtils.createTestArchive().addAsResource(
        "test-userProviderContext.xml");
  }
```

```
    @Autowired
    private UserProvider userProvider;
    @Test
    public void testInjection() {
      assertNotNull(userProvider);
    }
    @Test
    public void testUsersContent() {
      List<User> users = userProvider.findAllUsers();
      assertEquals(2, users.size());
    }
  }
```

The first line is still present to enable Arquillian, but instead of using `@SpringAnnotationConfiguration` we use `@SpringConfiguration`.
We indicate what files will be loaded, and since we're using the default `ClassPathApplicationContext`, the file will be read from the archive's classpath, which is why it is added as a resource. We specify the files to be loaded in the annotation. The remainder of the test is identical to the annotation-based one.

Arquillian versus Spring test

Comparing Spring test to the Arquillian test shows some additional features that Arquillian supports, but Spring may not. Since Arquillian handles your deployment capabilities, you can do real testing in an application server. Spring could be deployed to Tomcat or WebLogic; only Arquillian can simulate both for your testing purposes. This will allow you to detect any application server-specific issues that you may face. Arquillian Spring supports a similar context configuration: an annotation-driven way to configure test cases. In addition, if your database connections use JNDI you can point to that same JNDI location instead of defining properties for the connection.

Summary

As you can see, Arquillian provides a lot of flexibility in your testing setup. This gives you the ability to test your code as thoroughly as possible in an automated fashion while still having it behave as it would within your deployment structure.

In the next few chapters, I will be exploring more Arquillian extensions, where test cases will range from functional scenarios and browser testing via Selenium to invoking web services on the client side; as well as looking at how to integrate DBUnit with your tests; testing older Seam2 applications as well as Persistence and Transactions to name a few. The next chapter will discuss more general Arquillian extensions such as Persistence, Spock, and JaCoCo.

6 Arquillian Extensions

Throughout the book so far, we've spoken about a number of extensions to Arquillian. Extensions are built on top of Arquillian to cover some domain-specific use cases. As we've seen so far, there are extensions for Spring and Seam2 support as alternative dependency injection containers, support for Selenium, and functional testing via Warp and Drone. In this chapter, we go more in depth about Spring and how we can leverage Warp with it, as well as working with the Persistence and Transaction extensions. Finally, we talk about an alternative test runner for the Spock testing library.

Extending Arquillian

Arquillian is designed for extensibility, allowing you to get better control over your test environment. These extensions enhance your testing runtime and are available within the core of Arquillian. The first extension to look at is `@InSequence`, which will allow you to sort the execution of the test methods. The default behavior in JUnit is to sort based on the compiled order of the methods. There is a second option in JUnit to sort alphabetically. The `@InSequence` annotation can be placed on your test methods to sort them in a numeric (ascending) order. The idea behind this extension was to emulate functionality in TestNG that was not present in JUnit, specifically direct ordering of methods. Throughout the rest of the chapter, we'll see cases of how Arquillian can be extended, outside of JUnit.

Testing against a database using JPA

It's fairly simple to test JPA logic. You will need to ensure that you are deploying to a nearly complete container. You won't need to deploy JPA applications to a CDI embedded container; you need the resources of your application server – managed database connection's transaction support.

Assuming that you have an EJB that exposes your DAO, it's fairly simple to create test data and load it to your database. In simple terms, one way you can do this is to create the test data in the first step of a test and then use that data in the subsequent tests, finally deleting it when the test is over. One thing to keep in mind is that you should make sure you always work off a clean database and that you create and drop your test database with each test. Let's consider a simple `persistence.xml` file:

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://java.sun.com/
             xml/ns/persistence http://java.sun.com/
             xml/ns/persistence/persistence_2_0.xsd"
             version="2.0">
   <persistence-unit name="Chapter6Persistence">
      <jta-data-source>java:/jdbc/Chapter6DS</jta-data-source>
      <properties>
         <property name="hibernate.hbm2ddl.auto"
         value="create-drop"/>
      </properties>
   </persistence-unit>
</persistence>
```

We're using Hibernate and telling it to create and drop the schema on each test case. Also, we need to add the following datasource to our application server:

```
<datasource jndi-name="java:/jdbc/Chapter6DS" pool-name="Chapter6DS"
enabled="true" use-java-context="false">
    <connection-url>jdbc:h2:mem:chp6;
    DB_CLOSE_DELAY=-1</connection-url>
    <driver>h2</driver>
    <security>
        <user-name>sa</user-name>
        <password>sa</password>
    </security>
</datasource>
```

Once those are in place, we can think about the entity that will be in our code base, the specific scenario we want to test around `Person`. First the entity:

```
@Entity
@Table(name="PEOPLE")
public class Person {
  @Id
```

```
    @GeneratedValue
    private int id;
    private String firstName;
    private String lastName;
    //getters & setters ommitted...
  }
```

Next we have the DAO, which will perform all of our database interaction:

```
  @Stateless
  @LocalBean
  public class PersonDAO {
    @PersistenceContext(unitName="Chapter6Persistence")
    private EntityManager em;

    public void deleteAllPeople() {
      em.createQuery("Delete from Person p").executeUpdate();
    }

    public Person save(Person p) {
      return em.merge(p);
    }

    public Person find(int id) {
      return em.find(Person.class, id);
    }

    public List<Person> findAllPeople() {
      return em.createQuery("select p from Person p order
      by p.id",Person.class).getResultList();
    }
  }
```

We can actually leverage our DAO to do work when setting up a test case. It has the necessary operations to delete and create rows. One way to do that is to use the DAO in our `@Before` section of the test case. Observe the following code snippet:

```
  @EJB
  private PersonDAO dao;

  @Before
  public void setupTestData() {
    dao.deleteAllPeople();
```

```
        Person p1 = new Person();
        p1.setFirstName("Bob");
        p1.setLastName("Smith");
        dao.save(p1);
        Person p2 = new Person();
        p2.setFirstName("Jane");
        p2.setLastName("Smith");
        dao.save(p2);
    }

    @Test
    public void testLoadedPeople() {
        List<Person> allPeople = dao.findAllPeople();
        Assert.assertEquals(2, allPeople.size());
        Person p1 = allPeople.get(0);
        Assert.assertEquals("Bob", p1.getFirstName());
        Assert.assertEquals("Smith", p1.getLastName());

        Person p2 = allPeople.get(1);
        Assert.assertEquals("Jane", p2.getFirstName());
        Assert.assertEquals("Smith", p2.getLastName());
    }
```

We actually use the DAO to help in the test execution, and then run tests against it to find all rows and validate the results. A slightly different `@Before` method can be used:

```
    @EJB
    private PersonDAO dao;

    @PersistenceContext(unitName="Chapter6Persistence")
    private EntityManager em;

    @Inject
    private UserTransaction utx;

    @Before
    public void setTestData() throws Exception {
        utx.begin();
        em.joinTransaction();
        em.createQuery("delete from Person p").executeUpdate();
        Person p1 = new Person();
        p1.setFirstName("Bob");
        p1.setLastName("Smith");
        em.merge(p1);
        Person p2 = new Person();
        p2.setFirstName("Jane");
        p2.setLastName("Smith");
        em.merge(p2);
        utx.commit();
    }
```

In this version, we leverage the entity manager to do work. Either of these cases may work for you when working directly with the DAO. You get to actually use a little more of your code directly, thus improving your code coverage during testing.

However, if you are not in a case where you are testing the DAO directly, there may be a few issues with this approach. Here are some problems that I see:

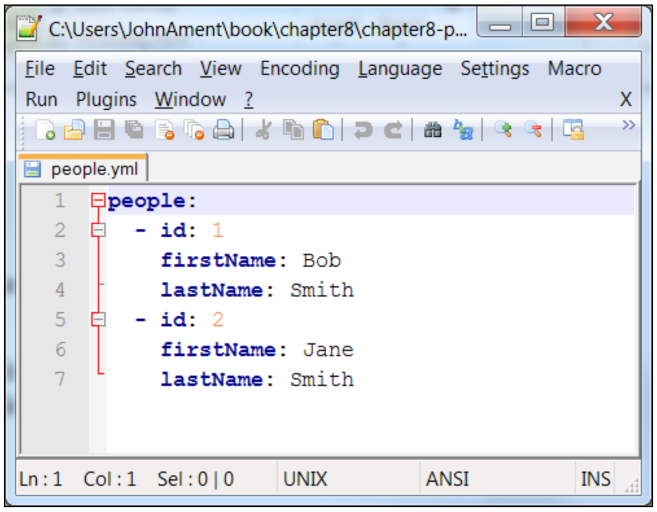
- These cases work fine only because we are testing the DAO's methods directly. If we were testing another component, say a REST API on top of this DAO, we may not need to work with the DAO or the Entity Manager directly.
- A bug in the DAO could incorrectly report passes or failures when running the tests.
- There are readability issues. If you are working on the API tier, you may not care about the HQL or JPQL or even native SQL that needs to be run.

The Persistence Extension

The Arquillian Persistence Extension can help you to test out your database-driven applications. This extension allows the seeding and automatic cleanup of the database tables. Most applications have some prerequisite data—data that's needed for initializing the system and performing basic functions against it.

Preloading the database

The Persistence Extension can help clean up some of this work. Primarily, it uses DBUnit to interact with the database to seed the initial data before a test. We can essentially replace all of that setup logic with a simple YML file and an annotation. Observe the contents of our YML file:


C:\Users\JohnAment\book\chapter8\chapter8-p...
File Edit Search View Encoding Language Settings Macro
Run Plugins Window ? X
people.yml

```
people:
  - id: 1
    firstName: Bob
    lastName: Smith
  - id: 2
    firstName: Jane
    lastName: Smith
```

Ln : 1 Col : 1 Sel : 0 | 0 UNIX ANSI INS

Then, our test case becomes:

```
@Test
@UsingDataSet("people.yml")
public void testUsersFound() {
  List<Person> allPeople = dao.findAllPeople();
  Assert.assertEquals(2, allPeople.size());
  Person p1 = allPeople.get(0);
  Assert.assertEquals("Bob", p1.getFirstName());
  Assert.assertEquals("Smith", p1.getLastName());
  Person p2 = allPeople.get(1);
  Assert.assertEquals("Jane", p2.getFirstName());
  Assert.assertEquals("Smith", p2.getLastName());
}
```

By default, all datasets will be searched for under `src/test/resources/datasets`; however, you can override the location in the annotation or in your `arquillian.xml` file. If you don't like YML for your format, you can also use JSON or XML. Here is an example of that same document in XML:

```
<dataset>
  <people id="1" firstName="Bob" lastName="Smith"/>
  <people id="2" firstName="Jane" lastName="Smith"/>
</dataset>
```

The following is in the JSON format:

```
{
  "people":
  [
    {
      "id": "1",
      "firstname" : "Bob",
      "lastname" : "Smith"
    },
    {
      "id": "2",
      "firstname" : "Jane",
      "lastname" : "Smith"
    }
  ]
}
```

As you can see, the XML version is the most concise and the JSON version is the most verbose. If you want, you can also use plain SQL scripts; just use a different annotation. Use `@ApplyBeforeScript` and then point to a SQL script, such as the following script:

```
INSERT INTO PEOPLE(ID,FIRSTNAME,LASTNAME) VALUES(1,'Bob','Smith');
INSERT INTO PEOPLE(ID,FIRSTNAME,LASTNAME)
VALUES(2,'Jane','Smith');
```

Then you can rerun the test.

Another feature allows custom DDL operations. Instead of rebuilding the schema using JPA settings, you can run your DDL operations during the test execution. At the class level we can add an annotation to create the schema, shown as follows:

```
@CreateSchema("scripts/ddl.sql")
public class PersonPersistenceTest {
```

Obviously, since we're using a script now to do it, we should change our `hbm2ddl.auto` setting in `persistence.xml` to be `none`. We don't want to validate since the schema doesn't get created until the test begins to execute.

```
<property name="hibernate.hbm2ddl.auto" value="none"/>
```

This likely matches how you run in production, where the schema is updated through your own deployment process. Finally, the file indicated by this annotation will need to include the following lines to make your table exist:

```
create table PEOPLE (id integer not null, firstName varchar(255),
lastName varchar(255), primary key (id));
create sequence hibernate_sequence start with 1 increment by 1;
```

Validate and clean up your database

Another use of the datasets is to check the results of your test case execution against what was performed by the database. Instead of reading and validating the data that was loaded, we can write the data to the database and ensure that it's saved properly using the following code snippet:

```
@RunWith(Arquillian.class)
@CreateSchema("scripts/ddl.sql")
public class PersonSaveTest {
...
@Test
@ShouldMatchDataSet("people-saved.xml")
public void testsSavedProperly() {
  Person p = new Person();
  p.setFirstName("Jane");
  p.setLastName("Doe");
  dao.save(p);
}
```

The `@ShouldMatchDataSet` annotation indicates what dataset(s) can be used to validate the results of executing your test case. The following XML matches that content:

```
<dataset>
  <people id="1" firstName="Jane" lastName="Doe"/>
</dataset>
```

When we run this test, we'll see that it passes but nothing was actually asserted, or was it? The persistence extension adds assertions based on this dataset to validate your results for you. If you change the last name here to something else, you'll see that it fails, with an error similar to the following:

```
java.lang.AssertionError: Test failed in 1 cases.
value (table=people, row=0, col=lastName)
expected:<[Do]e> but was:<[Jan]e>
```

Note that the values start with `row=0` and not `row=1`.

If you noticed, I actually created a separate test case for the saving of data over the reading of data. Next we'll review the data cleanup strategies to be able to combine these operations into one. Assume that we want to execute the following test case:

```
@RunWith(Arquillian.class)
@CreateSchema("scripts/ddl.sql")
public class PersonCombinedTest {
  @Deployment
  public static JavaArchive createArchive() {
    JavaArchive ja = ShrinkWrap.
    create(JavaArchive.class, "chapter6-persistence.jar")
        .addClasses(Person.class, PersonDAO.class)
        .addAsManifestResource("db-
         persistence.xml","persistence.xml")
        .addAsManifestResource(EmptyAsset.INSTANCE, "beans.xml");
    return ja;
  }

  @EJB
  private PersonDAO dao;

  @Test
  @UsingDataSet("people.xml")
  @InSequence(1)
  public void testUsersFound() {
    List<Person> allPeople = dao.findAllPeople();
    Assert.assertEquals(2, allPeople.size());
    Person p1 = allPeople.get(0);
    Assert.assertEquals("Bob", p1.getFirstName());
    Assert.assertEquals("Smith", p1.getLastName());
    Person p2 = allPeople.get(1);
    Assert.assertEquals("Jane", p2.getFirstName());
    Assert.assertEquals("Smith", p2.getLastName());
  }

  @Test
  @ShouldMatchDataSet("people-saved.xml")
  @InSequence(2)
  public void testsSavedProperly() {
    Person p = new Person();
    p.setFirstName("Jane");
    p.setLastName("Jane");
    dao.save(p);
  }
}
```

Using `@InSequence` will force JUnit to run the test cases in a certain order. When we run this test case it will fail because the table already exists. One of the downsides to the `@CreateSchema` annotation is that it forces you to recreate the schema after each test case. A good way to do this is to add a drop schema after the test, which can be done by using the `@CleanupUsingScript` annotation on your last test method in a test case. As a result, the second test method becomes the following:

```
@Test
@ShouldMatchDataSet("people-saved.xml")
@InSequence(2)
@CleanupUsingScript("drop-schema.sql")
public void testsSavedProperly() {
  Person p = new Person();
  p.setFirstName("Jane");
  p.setLastName("Doe");
  dao.save(p);
}
```

Here `drop-schema.sql` is:

```
drop table PEOPLE;
drop sequence hibernate_sequence;
```

As a result, each test case cleans up the tables for you automatically. You may be wondering at this point why the test didn't fail between the two methods in `PersonCombinedTest`. That's actually simple to explain: by default, there is a cleanup run on all test methods. You can override the default behavior by placing the `@Cleanup` annotation on your method. You can then specify a cleanup strategy and phase. Different cleanup strategies can include only deleting the rows created by the dataset. The default behavior is to delete everything. Likewise, you can change the strategy. I would recommend against changing the strategy; the default works well. It will delete after the method executes, which is typically what you want.

Another cleanup style to consider is the `@SeedDataUsing` annotation. This controls how data will be loaded into your database. If you are using multiple data loads in a single test case (across several methods), I would recommend using the clean insert option which does a cleanup first then inserts your data. This will remove any potential conflicts.

Mix and match

You shouldn't think that the persistence extension has to be used by itself. You should consider using it with other techniques as described in the book. For example, you can seed some data prior to running a web service request to use it for your test data.

The transaction extension

Formerly, transaction support was built into the persistence extension. It's now considered as a separate extension from persistence to allow other tools to use it. One case may be if you want to test out a JMS message queue using transactions from your test case.

By default, when you add it to your classpath and run a persistence-based test, transaction support will be enabled. It should be a safe assumption that any extension would also load transaction support when needed, as long as it's on the classpath. If desired, you can tweak the transaction support in your test cases (note that this only applies to your test classes; EJBs and CDI beans behave as is when deployed to the application server).

The API for the transaction extension is simply an enum and an annotation. Adding `@Transactional` to your test case will turn on the transaction support. Setting the transaction mode attribute will control some aspects of the test. The default behavior is to commit after a test case. However, in some cases an easier way is to roll back after the test executes—perhaps an easier way to clean up the test.

The Transaction Extension also gives you a clean way to keep your data cleaned between tests. The `@Transactional` annotation supports `ROLLBACK` and `DISABLED` options, as well as the default `COMMIT`. The `ROLLBACK` option will delete the transaction in progress, removing any contents you may have added. Consider the following test setup similar to the persistence examples we've already reviewed:

```
@RunWith(Arquillian.class)
public class TransactionTest {
    @Deployment
    public static JavaArchive createArchive() {
        JavaArchive ja = ShrinkWrap.
        create(JavaArchive.class, "chapter6-persistence.jar")
                .addClasses(Person.class, PersonDAO.class)
                .addAsManifestResource("META-INF/persistence.
                 xml","persistence.xml")
                .addAsManifestResource
                (EmptyAsset.INSTANCE, "beans.xml");
        return ja;
    }

    @EJB
    private PersonDAO dao;
```

```
        @Test
        @Transactional(TransactionMode.ROLLBACK)
        @InSequence(1)
        public void testCreatePerson() {
            Person p1 = new Person();
            p1.setFirstName("Sam");
            p1.setLastName("Transaction");
            Person p = dao.save(p1);
            Assert.assertTrue(p.getId() > 0);
        }

        @Test
        @InSequence(2)
        public void testFindPerson() {
            List<Person> allPeople = dao.findAllPeople();
            //the person created above will not be in this list
            for(Person p : allPeople) {
                Assert.assertFalse("Transaction".
                equals(p.getLastName()));
            }
        }
    }
```

We use `@InSequence` to order the tests, to ensure that the creation occurs before the second test. The first test creates a person and saves them; we assert this occurred by checking that the `Id` field is populated in the result. The second test though looks to find all possible people and checks that none of them are the created person. This test passing will show that the results were rolled back in the database. This would help a bit with automatically cleaning up the code that was created in your test.

Multiple artifacts

So far, we've reviewed creating simple artifacts. Most deployments have either been a WAR file or a JAR file. Arquillian does support RAR and EAR deployment; it's just that in Java EE 6 the need for having EARs has been greatly reduced–you can now deploy your EJBs into your WAR module, so it's unnecessary to have an external EJB-JAR.

There are some modular approaches to the application development and deployment that may require you to build out many deployable archives. OSGi is one standard that helps with it, but other tools exist such as JBoss modules that support similar packaging structures.

Creating EARs

Your application may choose to deploy as an EAR file. This would be based on the library usage as well as the decided deployment structure. In Java EE 6, you get all of the capabilities of an EAR deployment with EJB JARs packaged in the WAR, as well as being able to package EJBs directly in your WAR file.

The enterprise archives with Arquillian can be a little tricky. There are certain steps needed in EAR deployments that aren't needed otherwise. First, following are the deployment methods that we'll discuss:

```
@Deployment
public static Archive<?> createArchive() {
  JavaArchive ejb = ShrinkWrap.create
  (JavaArchive.class, "chapter6.jar")
      .addClass(TemperatureConverter.class)
      .addAsManifestResource(EmptyAsset.INSTANCE, "beans.xml");
  JavaArchive cdi = ShrinkWrap
      .create(JavaArchive.class, "chapter6-cdi.jar")
      .addClass(ConversionController.class)
      .addAsManifestResource(EmptyAsset.INSTANCE, "beans.xml");
  JavaArchive test = ShrinkWrap.create(JavaArchive.class).addClass(
      ConversionControllerTest.class);
      return ShrinkWrap
      .create(EnterpriseArchive.class, "chapter6-ear.ear")
      .addAsModule(ejb)
      .addAsModule(cdi)
      .addAsLibrary(test)
      .addAsManifestResource(createApplicationXml(),
          "application.xml");
}

private static StringAsset createApplicationXml() {
  return new StringAsset(Descriptors.create(ApplicationDescriptor.
  class)
      .version("6").displayName("chapter6-ear")
      .ejbModule("chapter6.jar").ejbModule("chapter6-cdi.jar")
      .exportAsString());
}
```

We create three Java archives, one representing the EJB module, another representing the UI that is exposed via CDI, and finally the test archive. In EAR deployments we have to add the test class to the archive as a library. This is to ensure that it is found by the classloaders. Then we create the enterprise archive. We add the two modules and one library. Finally, we create the manifest resource. In order to create the `application.xml` file, we use the ShrinkWrap descriptors. That work is delegated, which returns a `StringAsset` that will represent the file contents. We do need to ensure that the two files are named correctly based on the names in the archive creation.

The ShrinkWrap descriptors allow you to create deployment descriptors in a programmatic fashion to be added to ShrinkWrap deployments. In order to be added to an archive they need to be converted into assets, which is done here by calling `exportAsString()` and wrapping all of this in a constructor for `StringAsset`.

The CDI component that we're going to test in the following code snippet is very simple, takes inputs, and validates results:

```
@RequestScoped
@Named("conversion")
public class ConversionController {
  @EJB
  private TemperatureConverter converter;

  private Double celsius;
  private Double fahrenheit;

  public String handleConversion() {
    if(celsius != null) {
      fahrenheit = converter.convertToFahrenheit(celsius);
    } else if(fahrenheit != null) {
      celsius = converter.convertToCelsius(fahrenheit);
    }
    return "/mainView.xhtml";
  }
  // getters and setters omitted
}
```

In order to test this, we have two scenarios to test out. The first is when celsius is set and the other when fahrenheit is set. We can test it out using the following methods:

```
@Inject
private Instance<ConversionController> controllerInstance;
```

```
@Test
public void testConvertedToFareinheit() {
  ConversionController controller = controllerInstance.get();
  double celsius = 10;
  double expectedFahrenheit = 50;
  controller.setCelsius(celsius);
  controller.handleConversion();
  assertEquals(expectedFahrenheit,controller.getFahrenheit(),0);
}

@Test
public void testConvertedToCelsius() {
  ConversionController controller = controllerInstance.get();
  double expectedCelsius = 20;
  double fahrenheit = 68;
  controller.setFahrenheit(fahrenheit);
  controller.handleConversion();
  assertEquals(expectedCelsius,controller.getCelsius(),0);
}
```

In this test case, we validate based on simulated form input. We also inject a new instance per method, since it is request scoped. This simulates closer how each individual request will hit the server.

Another way to build this EAR involves using ShrinkWrap's Resolver API to bring in the dependencies. This makes it simpler since we don't need to rebuild the JAR. It also ensures that we use the real JAR file, not only parts of the JAR file. This gives our application more evidence of functioning as expected, especially since we use a fully packed EJB JAR as it would be built for deployment. Refer to the following code snippet:

```
@Deployment
public static Archive<?> createArchiveWithResolver() {
  File[] files = Maven.resolver().loadPomFromFile("pom.xml")
      .resolve("com.tad.arquillian:chapter6").withTransitivity()
      .asFile();
  File ejb = files[0];
  JavaArchive cdi = ShrinkWrap
      .create(JavaArchive.class, "chapter6-cdi.jar")
      .addClass(ConversionController.class)
      .addAsManifestResource(EmptyAsset.INSTANCE, "beans.xml");
  JavaArchive test = ShrinkWrap.create
  (JavaArchive.class).addClass(
      ConversionControllerTest.class);
```

```
        return ShrinkWrap
            .create(EnterpriseArchive.class, "chapter6-ear.ear")
            .addAsModule(ejb,"chapter6.jar")
            .addAsModule(cdi)
            .addAsLibrary(test)
            .addAsManifestResource(createApplicationXml(),
            "application.xml");
    }
```

The other difference here is that the default packaging will include the version number of the dependency. In order to use it correctly in our code, since we didn't change the descriptor, we have to rename the file as it goes into the EAR file.

JaCoCo and Arquillian – code coverage

Now's actually a good time to talk about JaCoCo. It's time to switch gears a little bit and discuss code coverage and Arquillian. If you're familiar with code coverage techniques, the code coverage tool supported by Arquillian is **JaCoCo**. Arquillian works a bit differently: the test is distributed across multiple JVMs, rather than the normal in-VM approach used. With other tools such as Emma, you can test how much of your code is validated as long as it runs locally. JaCoCo works well with the multiple JVM setup of Arquillian. In order to turn on code coverage in your project, you simply need to add the following dependencies to your project:

```
<dependency>
  <groupId>org.jboss.arquillian.extension</groupId>
  <artifactId>arquillian-jacoco</artifactId>
  <version>1.0.0.Alpha5</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.jacoco</groupId>
  <artifactId>org.jacoco.core</artifactId>
  <version>0.6.0.201210061924</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>asm</groupId>
  <artifactId>asm-util</artifactId>
  <version>3.3.1</version>
  <scope>test</scope>
</dependency>
```

In addition, there is some configuration you can add to `arquillian.xml`. The following are the includes and the excludes:

```
<extension qualifier="jacoco">
  <property name="includes">some.package.*</property>
  <property name="excludes">some.package.ignores1.*;
  some.package.ignores2.*</property>
</extension>
```

This will tell the JaCoCo extension to only look under `some.package` and explicitly ignore `ignores1` and `ignores2` from there. We can use a `;` between packages in both the properties.

When you run JaCoCo, the results will be in a binary file, `target/jacoco.exec`. Within Maven, there is a site generated with the JaCoCo output that can be viewed and shows the source code where the probes were placed and the ones which were invoked. This binary file can also be opened in Eclipse if you have the JaCoCo extension plugin installed. Alternatively, if you are running Jenkins, you can add the JaCoCo coverage report plugin to view the code coverage reports. So you have many options for reading through the results of your code coverage and some decent views of the data. Just remember that JaCoCo will only work on your testable archives; Arquillian integration is only enabled for testable archives. If your classes are in other archives, they will not be considered for code coverage when generating the report. This is simply a limitation to the deployment structure of Arquillian, since it requires enrichment of the archive at deployment time.

Deploying dependencies

Another way to handle the modular structure is to deploy your dependencies to the application server directly. This may work well if you have a lot of EJBs that you look up via reference or have a very modular application structure that can be broken down. This is how OSGi is structured and how JBoss modules are structured. This allows you to build out your application piece by piece, instead of maybe deploying a full EAR or WAR file representing the entire application.

Creating a multiple archive test is not as simple as creating the deployments and expecting them to be testable immediately. Arquillian needs a bit more configuration in your test to make multiple deployments work correctly (in fact, it does not work):

```
@RunWith(Arquillian.class)
public class DualArchiveTest {
  @Deployment(order=1,name="archone")
  public static JavaArchive createArchiveOne() {
    return ShrinkWrap.create(JavaArchive.class, "archiveone.jar")
        .addClass(BeanOne.class)
        .addAsManifestResource(EmptyAsset.INSTANCE,"beans.xml");
  }

  @Deployment(order=2,name="archtwo")
  public static JavaArchive createArchiveTwo() {
    return ShrinkWrap.create(JavaArchive.class, "archivetwo.jar")
        .addClass(BeanTwo.class)
        .addAsManifestResource(EmptyAsset.INSTANCE,"beans.xml");
  }

  @Inject
  @Test
  @OperateOnDeployment("archone")
  public void testArchOne(BeanOne beanOne) {
    Assert.assertEquals("Bean One",beanOne.getMessage());
  }

  @Inject
  @Test
  @OperateOnDeployment("archtwo")
  public void testArchTwo(BeanTwo beanTwo) {
    Assert.assertEquals("Bean Two",beanTwo.getMessage());
  }
}
```

This setup is a bit broken, you have essentially two unrelated beans that you are trying to test in a single test case. Do you believe they need to be different archives? Is this the right approach? You cannot load CDI components across archives like this. Which JAR should the test be deployed to? The main problem with this structure is that there is no structure. The deployment order is not clear, the dependencies are not clear.

Instead, let's look at a slightly modified version of the test case in the following code snippet. In this case, we use a CDI component that calls an EJB component. Only one archive is testable.

```
@Stateless
@LocalBean
public class BeanOne {
  public String getMessage() {
    return "Bean One";
  }
}

public class BeanTwo {
  @EJB(lookup = "java:global/archiveone/BeanOne")
  private BeanOne beanOne;
  public String getMessage() {
    return "Bean Two";
  }
}

@RunWith(Arquillian.class)
public class EJBDualArchiveTest {
  @Deployment(order = 1, name = "archone", testable = false)
  public static JavaArchive createArchiveOne() {
    return ShrinkWrap.create(JavaArchive.class, "archiveone.jar")
    .addClass(BeanOne.class)
    .addAsManifestResource(EmptyAsset.INSTANCE, "beans.xml");
  }

  @Deployment(order=2,name="archtwo")
  public static JavaArchive createArchiveTwo() {
    return ShrinkWrap.create(JavaArchive.class, "archivetwo.jar")
    .addClass(BeanTwo.class)
    .addAsManifestResource(EmptyAsset.INSTANCE,"beans.xml")
    .addAsManifestResource(createDeploymentAsset(),
    "jboss-deployment-structure.xml");
  }

  private static StringAsset createDeploymentAsset() {
    StringBuilder sb = new StringBuilder();
    sb.append("<JBoss-deployment-structure
    xmlns=\"urn:JBoss:deployment-structure:1.1\">");
    sb.append("<deployment><dependencies>");
    sb.append("<module name=\"deployment.archiveone.jar\" />");
    sb.append("</dependencies></deployment>
    </jboss-deployment-structure>");
    return new StringAsset(sb.toString());
  }
```

```
    @Inject
    private BeanTwo beanTwo;

    @Test
    @OperateOnDeployment("archtwo")
    public void testArchTwo() {
      Assert.assertEquals("Bean Two",beanTwo.getMessage());
    }
  }
```

There is some application server-specific task going on here, but in general it should be reproducible on other containers with little trouble. We still have two deployments, but the first deployment which has an EJB is not considered testable. We need to ensure that there is only one testable archive to avoid classloader issues; Arquillian will not know where to send the test case without this. Making the archive testable makes it deploy with the Arquillian libraries, whereas setting `testable=false` makes it deploy as configured; no further enrichment occurs.

We also add a JBoss-specific deployment descriptor to the second archive. This descriptor tells JBoss to add the first deployment's classes to the classloader of the second deployment. This allows the CDI bean to see the EJB in another archive. We do need to specify the JNDI location of the EJB; but that's easy now because we have standardized JNDI naming conventions for the platform.

This approach would also work if you wanted to deploy multiple web services in a single operation to the container and then execute them all serially. You shouldn't assume any specific order though in your testing, unless you build it all in a single method.

If you do deploy across many archives, you should keep your project's structure in mind. Your deployment order (for non-tested archives) should match how it would be deployed in production. Wherever possible, you should match your production deployment. Perhaps instead of building your archive, you could import it using ShrinkWrap:

```
  @Deployment(order = 5, name = "myarchive.jar")
  public static JavaArchive importMyDependency() throws Exception {
    File f = Maven.resolver().loadPomFromFile("pom.xml")
        .resolve("com.mycompany:myarchive").withTransitivity()
        .asSingleFile();
    return ShrinkWrap.createFromZipFile(JavaArchive.class, f);
  }
```

Using this approach, if we have a prepackaged JAR file, we can simply import it as that file into ShrinkWrap's model and process it as a deployment to the application server. This simplifies things immensely, as long as we know the archive exists. We also remove any compile-time dependencies on the archive's contents, which simplifies impact if something changes about the archive. Since it's now in ShrinkWrap's format, we can override contents of the file (obviously without changing the file) so that we can change `persistence.xml` files (to make them configured for test environments) or add new classes and descriptors to test out different configurations.

Spock, very groovy test writing

Spock is a specification-oriented testing framework where your tests are written in a requirements-oriented way. The test cases are written in Groovy, but also use a JUnit test-case runner to execute the test case. Spock uses datasets to drive the test cases. Spock does things based on **Behavior-driven development** (**BDD**) to create test cases.

Spock uses specifications to define test cases, which will fail until the requirement is implemented. We can use Spock with Arquillian to execute deployments and run test cases against server-side code.

Let's suppose we have the following requirement:

Certain objects may have a state. States that these objects have can progress from 1 to 5. If one of these objects has state less than 1, then set its state to 1; also, if the state is greater than 5, set its state to 1. The server will expose the code that progresses the state of these objects, which will be sequential from 1 to 5.

From a design standpoint, we make a few assumptions:

- Objects that have this state attribute will be known as "Stateful" and all instances shall extend from `Stateful`.
- The class `StateProcessor` shall represent the server-side code that progresses the state. This class will also handle validating the top and the bottom of the test case.

Next, we'll review how the Spock case is built for that scenario:

```
class StateManagementSpecification extends Specification {
  @Deployment
  def static JavaArchive "create deployment"() {
    return ShrinkWrap.create(JavaArchive.class)
      .addClasses(Stateful.class, BasicStateObject.
       class, StateProgressor.class)
     .addAsManifestResource(EmptyAsset.INSTANCE, "beans.xml");
  }

  @Inject
  StateProgressor progressor

  def "state less than one or greater than five reset value"() {
    setup:
    def inState = new BasicStateObject("State Zero","Zero",0)
    def resultState = 2

    when:
    progressor.progressToNextState(inState)

    then:
    inState.state == resultState
  }
}
```

The class is made up of three parts:

- Just like every other Arquillian test we've built, a `@Deployment` method that creates the archive is included.
- Next we have our injection target. Arquillian/Spock enriches the `Groovy` class to inject this field into the test case. Obviously, we can have more injection points in the `Groovy` class, based on what needs to be tested.
- Finally, we have our test method. Using groovy conventions we can use any String as a method name, which helps Spock to create very descriptive test cases.

One important thing to notice is that we don't use a `@RunWith` on the class. Arquillian is automatically added based on the dependencies being added to the classpath. Within the test method, or as Spock refers to it the feature method, there are a number of blocks that can be run: `setup`, `when`, `then`, `expect`, `cleanup`, and `where`.

The example feature method provided earlier was a `setup`/`when`/`then`. Setup can be considered your prequalifications for entering the feature. You should not try to mix the `setup` and `when` codes. The `expect` blocks can be used to validate simple conditions, perhaps a condition right after instantiation. For example, if we add a new requirement that there be a way to validate the state of a `Stateful` object, we can add the following feature method:

```
def "validate the state of a stateful"() {
  setup:
  def inState = new BasicStateObject("State Zero","Zero",0)
  expect:
  false == progressor.isValidState(inState)
}
```

In this feature, we are performing a negative test so that state 0 is invalid. We implement the feature as follows:

```
public boolean isValidState(Stateful stateful) {
  return stateful.getState() >= 1 && stateful.getState() <=5;
}
```

You can have multiple statements in your `setup` block; in addition, you can have multiple statements in your `expect` block. We can then make this feature more robust and add a few more scenarios to test as shown:

```
def "validate the state of a stateful"() {
  setup:
  def invalidState = new BasicStateObject("State Zero","Zero",0)
  def invalidState6 = new BasicStateObject("State Six","Six",6)
  def validState = new BasicStateObject("State One","One",1)
  expect:
  false == progressor.isValidState(invalidState)
  false == progressor.isValidState(invalidState6)
  true == progressor.isValidState(validState)
}
```

Now, this might be a bit verbose, or even error prone. Spock allows us to create data-driven test cases. We can rewrite the last two scenarios as follows:

```
def "state less than one or greater than five reset value"() {
  when:
  progressor.progressToNextState(inState)
  then:
  inState.state == resultState
  where:
  resultState << [2,1,2]
  inState << [new BasicStateObject("State Zero","Zero",0),
    new BasicStateObject("State Six","Six",6),
    new BasicStateObject("State One","One",1)]
}

def "validate the state of a stateful"() {
  expect:
  valid == progressor.isValidState(inState)
  where:
  valid << [false,false,true]
  inState << [new BasicStateObject("State Zero","Zero",0),
    new BasicStateObject("State Six","Six",6),
    new BasicStateObject("State One","One",1)]
}
```

As a result, the test cases are much more concise and the assertions only need to be written once. One more thing that we cannot do is as follows:

```
def "validate the state in a database driven way"() {
  expect:
  (inState.state >= 1 && inState.state <= 5) ==
  progressor.isValidState(inState);
  where:
  inState << progressor.getAllStates();
}
```

Spock expects the test data to be static in nature, not database driven. You can only access your injected fields in your `when` and `expect` blocks, not in your `then` or `where` blocks. If needed, you can use them in your `cleanup` block as well to do some cleanup work. However, there is a clever workaround:

```
def "validate the state in a database driven way"() {
  expect:
  progressor.getAllStates().each { inState ->
    assert (inState.state >= 1 && inState.state <= 5) == progressor.
    isValidState(inState);
  }
}
```

Instead of injecting the progressor directly, we can inject an instance of it and use a `when`/`then`/`where` block set to use a data-driven test case. This isn't ideal since we are now mixing the `setup` and iteration logic with our `expect` clause, but in order to be able to validate the results this provides a convenient and easy-to-work-with.

Spock has some additional features around test-driven development, but these were meant to showcase what you can do with its integration with Arquillian in order to better execute tests.

Summary

We've reviewed three additional extensions now for use with Arquillian. These extensions were meant to show the diversity of what you can do with the tool and give some ideas about how they can be combined with one another, as well as with other extensions reviewed throughout the book.

In the next chapter, we'll start to look into testing web applications using Arquillian via the Warp, Drone, and Graphene extensions. We'll also dig into Spring more with support for Spring MVC. This is meant to be UI oriented, and will use Arquillian, Drone, Warp, and Selenium. You should download and install the Selenium IDE before going through that chapter. We will also build a simple web application using JSF and then test it using this suite of tools.

7
Functional Application Testing

Since Arquillian provides deployment support, it is possible to deploy your entire application and perform end-to-end functional testing on the final deployed product. This makes your testing much more robust – not only are you testing how your code interacts, but you're testing how your UI is rendered, and how forms are filled out, and then submitted back to the server.

Arquillian has a few possible solutions for UI-based testing. The first that we'll review is Drone, which uses Selenium to execute test cases and run verifications. Next is Warp, which extends Drone to add assertions on both the client side and server side. Finally, we'll look at Graphene, another Drone-based extension for testing rich, AJAX powered applications. Within this chapter, we'll use two distinct projects. The first project is based on plain Selenium, then Drone, and Warp. The second project is based purely on Graphene.

The distinction between these projects will become very apparent, and which one to use will be clear once you see how each performs. In addition, we will compare using these extensions to running plain Selenium JUnit test cases, as well as understand how **Test Driven Development** (**TDD**) methodologies will drive your testing tool approaches.

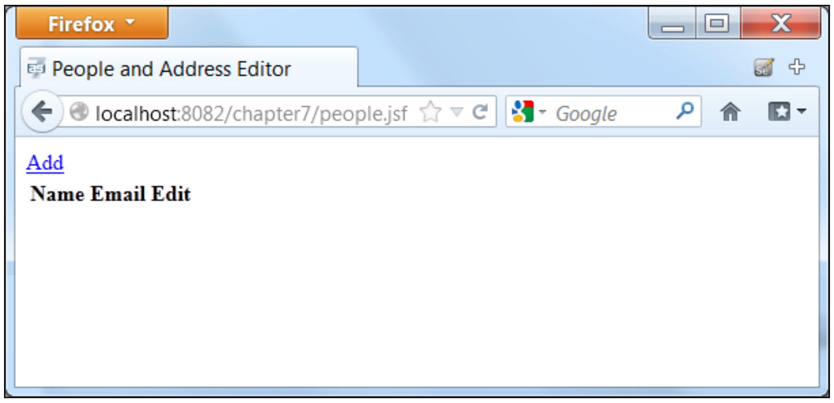
Functional testing

Functional testing is a necessary part of application testing to ensure that the functional requirements of what you are building are satisfied. These can be assumed as passing by reading requirements or acceptance criteria, performing the said function, and validating that the result is as expected.

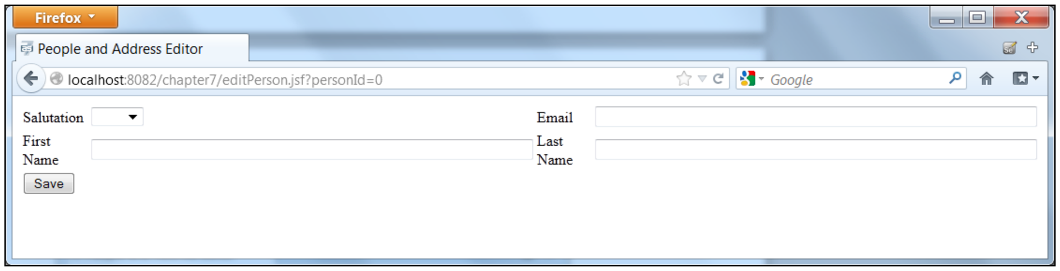
Historically, functional testing has been performed manually. There is a certain amount of automated testing that has existed, such as automating the browser function and running scripts to ensure system status. Arquillian adds some more powerful assertions to help you get the most out of this automated testing. They are exposed in Drone, Warp, and Graphene, which this chapter covers in depth. We'll also look at some specific Spring MVC examples that use Warp.

Functional testing using Selenium

Once installed, Selenium IDE will be a menu option in Firefox. You can add the icon to your menu bar and launch the recording tool. The recording tool will capture clicks you perform based on accessing your code in a browser. Once you've built the sample project, deploy it to your JBoss instance. Launch your browser window and navigate to `http://localhost:8080/chapter7/people.jsf`. Go ahead and click on **Add** and add someone to your people list. When you launch the application, by default it's going to look like the following screenshot:



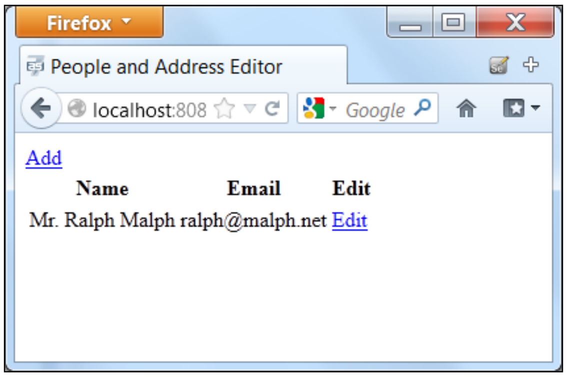
When you click on **Add**, you'll get the following screenshot:


Firefox
People and Address Editor
localhost:8082/chapter7/editPerson.jsf?personId=0
Google
Salutation
Email
First Name
Last Name
Save

Enter the following values:

- **Salutation**: `Mr.`
- **First Name**: `Ralph`
- **Last Name**: `Malph`
- **Email**: `ralph@malph.net`

Then click on **Save**. You'll be brought back to the following screenshot:



Add

Name	Email	Edit
Mr. Ralph Malph	ralph@malph.net	Edit

Once you have gone through these steps, you'll have a functioning test case. Once done, stop recording. You can play back the recording from here. You can also export the test case as a JUnit 4 test case. Here's an example of what gets exported:

```
@Before
public void setUp() throws Exception {
  driver = new FirefoxDriver();
  baseUrl = "http://localhost:8082/";
  driver.manage().timeouts().implicitlyWait(30, TimeUnit.SECONDS);
}

@Test
public void testSelenium2WebDriver() throws Exception {
  driver.get(baseUrl + "/chapter7/people.jsf");
  driver.findElement(By.linkText("Add")).click();
  new Select(driver.findElement(By.id("editPersonForm:salutation"))).
  selectByVisibleText("Mr.");
  driver.findElement(By.id("editPersonForm:email")).clear();
  driver.findElement(By.id("editPersonForm:email")).sendKeys("ralph@
malph.net");
```

```
    driver.findElement(By.id("editPersonForm:firstName")).clear();
    driver.findElement(By.id("editPersonForm:firstName")).
    sendKeys("Ralph");
    driver.findElement(By.id("editPersonForm:lastName")).clear();
    driver.findElement(By.id("editPersonForm:lastName")).
    sendKeys("Malph");
    driver.findElement(By.name("editPersonForm:saveButton")).click();
  }
```

This test case can actually be extended by Arquillian to support deployment. Simply add the following deployment method:

```
  @Deployment(testable=false)
  public static WebArchive createChapter7Archive() {
    returnShrinkWrap.create(WebArchive.class, "chapter7.war")
        .addClasses(Addresses.class,People.class,
            PeopleDAO.class,PeopleController.class)
        .addAsResource("META-INF/persistence.xml","META-INF/persistence.
         xml")
        .addAsWebResource("editPerson.xhtml","editPerson.xhtml")
        .addAsWebResource("people.xhtml","people.xhtml")
  .addAsWebInfResource("WEB-INF/faces-config.xml","faces-config.xml")
  .addAsWebInfResource("WEB-INF/web.xml","web.xml")
  .addAsWebInfResource(EmptyAsset.INSTANCE,"beans.xml");
  }
```

This deployment method is marked as not testable – we don't enrich it when deploying to the server. In addition, the following `pom` changes were made; this allows us to easily include primary files in our deployment. This is cleaner than redefining the contents of those files, since they may change at any time. Note that we cannot import the source WAR file since it hasn't been built yet. Packaging doesn't run until after tests.

```
  <testResources>
    <testResource>
      <directory>src/main/webapp</directory>
    </testResource>
    <testResource>
      <directory>src/main/resources</directory>
    </testResource>
  </testResources>
```

This adds the web resources and Java resources to our test classpath, making them accessible. We also build out the archive to match our production deployment as closely as possible. Next, we inject a reference to the URL of our deployment. This is the same as any other client test that we execute.

```
@ArquillianResource
private URL url;
```

Finally, we make some minor changes to the test method. The preceding code gets integrated into the test method as well as the following code.:

```
@Test
@RunAsClient
public void testSelenium2WebDriver() throws Exception {
  String baseUrl = String.format("http://%s:%s",url.getHost(),url.
  getPort());
  WebDriver driver = new FirefoxDriver();
  driver.manage().timeouts().implicitlyWait(30, TimeUnit.SECONDS);
  driver.get(baseUrl + "/chapter7/people.jsf");
  driver.findElement(By.linkText("Add")).click();
  new Select(driver.findElement(By.id("editPersonForm:salutation"))).
  selectByVisibleText("Mr.");
  driver.findElement(By.id("editPersonForm:email")).clear();
  driver.findElement(By.id("editPersonForm:email")).sendKeys("ralph@
  malph.net");
  driver.findElement(By.id("editPersonForm:firstName")).clear();
  driver.findElement(By.id("editPersonForm:firstName")).
  sendKeys("Ralph");
  driver.findElement(By.id("editPersonForm:lastName")).clear();
  driver.findElement(By.id("editPersonForm:lastName")).
  sendKeys("Malph");
  driver.findElement(By.name("editPersonForm:saveButton")).click();
  driver.quit();
}
```

This method will launch your Firefox browser and execute the form and save flow. When we execute this test case we will observe that no errors were encountered during the processing; the form saved correctly based on the programmed flow.

Consider the approach

This approach works well if you have a working application. However, you may want to define the test case first, then execute the test case against the completed code. When we create the test scenario first, we are using more of a test-driven approach.

Test Driven Development

Test Driven Development (TDD) is an approach where test cases are built first, and followed up by the implementation. The test is failing until the implementation is placed into your build stack. Test Driven Development may work best for you for functional testing. There is usually enough to go on – form layouts, and wireframes that can be used for testing purposes.

Assuming that you also develop against interfaces, if the interface is defined then you can build your test cases against those interfaces. When your test case is done, and the code is done, then you should expect your test cases to pass. The test case is considered failing until that occurs.

Simplifying the test case

Arquillian Drone allows you to define some custom `@ArquillianResources`; the primary one here is the Drone resource, which provides injection of your `WebDriver` class.

```
@ArquillianResource
private URL url;

@Drone
privateWebDriver driver;

@Test
@RunAsClient
public void testSomething() {
  String baseUrl = String.format("http://%s:%s",url.getHost(),url.
  getPort());
  driver.manage().timeouts().implicitlyWait(30, TimeUnit.SECONDS);
  driver.get(baseUrl + "/chapter7/people.jsf");
  driver.findElement(By.linkText("Add")).click();
  new Select(driver.findElement(By.id("editPersonForm:salutation"))).
  selectByVisibleText("Mr.");
  driver.findElement(By.id("editPersonForm:email")).clear();
  driver.findElement(By.id("editPersonForm:email")).sendKeys("ralph@
  malph.net");
  driver.findElement(By.id("editPersonForm:firstName")).clear();
  driver.findElement(By.id("editPersonForm:firstName")).
  sendKeys("Ralph");
  driver.findElement(By.id("editPersonForm:lastName")).clear();
  driver.findElement(By.id("editPersonForm:lastName")).
  sendKeys("Malph");
  driver.findElement(By.name("editPersonForm:saveButton")).click();
  driver.quit();
}
```

The major change is that now instead of instantiating the Firefox driver, we allow Drone to inject the driver to the test case. This means we can simply configure the driver rather than changing test code to use one driver over another.

When working with Drone natively, as well as Warp, we use the internal Selenium APIs. One thing we'll notice when comparing is that Graphene uses its own DSL. This may be beneficial for those who are looking to be tool agnostic.

Page delegate pattern

Something unique to this setup type of pattern is the **Page Delegate**. This is similar to any other delegate pattern, except it is meant to simulate browser interaction. Let's say we take this page and refactor it into a concrete object. First, we design our object to match the calls a user may make. Click on the **Add** button, set the field values, and then click on **Save**. This is how our object ends up looking:

```
public class EditPersonForm {
private static final String EMAIL = "editPersonForm:email";
private static final String FIRST_NAME = "editPersonForm:firstName";
private static final String LAST_NAME = "editPersonForm:lastName";
private static final String SALUTATION = "editPersonForm:salutation";
privateWebDriver driver;
private URL url;
publicEditPersonForm(WebDriverdriver,URLurl) {
this.driver = driver;
        this.url = url;
navigate();
    }
private void navigate() {
        String baseUrl = String.format("http://%s:%s",url.
getHost(),url.getPort());
driver.manage().timeouts().implicitlyWait(30, TimeUnit.SECONDS);
driver.get(baseUrl + "/chapter7/people.jsf");
    }
private void select(String field, String value) {
new Select(driver.findElement(By.id(field))).
selectByVisibleText(value);
    }
private void setFieldValue(String field, String value) {
WebElement el = driver.findElement(By.id(field));
el.clear();
el.sendKeys(value);
    }
public void setEmail(String email) {
setFieldValue(EMAIL,email);
    }
public void setFirstName(String firstName) {
setFieldValue(FIRST_NAME,firstName);
    }
public void setLastName(String lastName) {
setFieldValue(LAST_NAME,lastName);
    }
public void setSalutation(String salutation) {
select(SALUTATION,salutation);
    }
public void clickAdd() {
driver.findElement(By.linkText("Add")).click();
    }
public void clickSave() {
driver.findElement(By.name("editPersonForm:saveButton")).click();
    }
}
```

So then our test method matches the actions performed:

```
@Test
@RunAsClient
public void testViaDelegate() {
  EditPersonFormepf = new EditPersonForm(driver,url);
  epf.clickAdd();
  epf.setSalutation("Dr.");
  epf.setEmail("dr.killjoy@cnn.net");
  epf.setFirstName("David");
  epf.setLastName("Killjoy");
  epf.clickSave();
}
```

This may be extra work, but can be more reusable when testing multiple scenarios in an acceptance setup.

Warp drive

Arquillian Warp is an Arquillian extension that provides additional functionality on top of Drone to handle client and server assertions within a single test case. Using Drone we can drive Selenium, which allows us to test the UI. What we cannot do is verify application functionality end to end. We cannot test the result of the UI interaction in our database via web service interactions. Warp allows the tester to run the UI and then validate server state, both before and after the execution steps. There are some minor differences in a Warp test compared with a regular Drone test. Here's an example:

```
@WarpTest
@RunWith(Arquillian.class)
```

```
@RunAsClient
public class WarpDroneTest {
    @ArquillianResource
    URL url;

    @Drone
WebDriver driver;
@Deployment(testable=true)
public static WebArchive createChapter7Archive() {
return DeploymentUtils.createChapter7Archive();
    }

    @Test
public void test() {
final String baseUrl = String.format("http://%s:%s",url.getHost(),url.
getPort());
driver.get(baseUrl + "/chapter7/people.jsf");
driver.findElement(By.linkText("Add")).click();
new Select(driver.findElement(By.id("editPersonForm:salutation"))).
selectByVisibleText("Mrs.");
driver.findElement(By.id("editPersonForm:email")).clear();
driver.findElement(By.id("editPersonForm:email")).sendKeys("jane.doe@
anyplace.com");
driver.findElement(By.id("editPersonForm:firstName")).clear();
driver.findElement(By.id("editPersonForm:firstName")).
sendKeys("Jane");
driver.findElement(By.id("editPersonForm:lastName")).clear();
driver.findElement(By.id("editPersonForm:lastName")).sendKeys("Doe");
Warp.initiate(new Activity(){
            @Override
public void perform() {
driver.findElement(By.name("editPersonForm:saveButton")).click();
            }
        })
        .inspect(new Inspection(){
private static final long serialVersionUID = 1L;
            @EJB
privatePeopleDAOdao;

@AfterPhase(Phase.RENDER_RESPONSE)
public void testAfterRenderResponse() {
                List<People> people = dao.findAllPeople();
boolean found = false;
for (People p : people) {
```

```
if (p.getEmail().equals("jane.doe@anyplace.com")) {
found = true;
                    }
                }
assertTrue(found);
            }
        });
    }
}
```

The first thing to notice is the test setup. First, we added the `@WarpTest` annotation to the class, which tells **Arquillian** to launch Warp. Next, we note the class as `@RunAsClient` – all Warp tests must be `@RunAsClient`. We also need to make the deployment testable. Warp requires the archive to be enriched to add more test cases. This is because the server-side assertions happen on the server, so those assertions will be deployed to the server and run.

Next, we can inject the Drone and the URL the same way as in a standard Drone test case. Using something similar to the previous example, we'll add `Mrs. Jane Doe` to the list of people for the application. In this case, I simply modified the test case to change the input data. The form input is broken down into two parts. There are a series of steps that occur before the test begins to enter data to the form. The next part is to create the actual `Activity` – this is what Warp will intercept to check that an HTTP request occurred. Due to the nature of how activities work, you want to ensure that only one HTTP request occurs within an activity unless you use groups. Whenever an activity occurs, Warp will then kick off an inspection.

The last part is an `Inspection`. Inspections are how you validate the server-side state after the scenario has been run. This is where your assertions should live. Your inspections can run before a test as well as after a test. It is meant to research server state and perform assertions based on that server state. This could be as simple as validating HTTP requests or more complex, such as using an EJB to retrieve data from a database and report against it.

Note: Inspections must have a `serialVersionUID` defined.

There is an optional part, omitted from here for now. There is an observe method call you can add, before you inspect and after you perform, which will dictate which of the requests performed should be tracked. Most requests do not require this. This step is a bit dangerous in that it filters out certain calls from your stack, removing pulling down of artifacts or static text.

Phases

Phases are an important part of inspections. Phases dictate when inspection should occur. Provided with the base Warp are two phases—`BeforeServlet` and `AfterServlet`. They are bound to the servlet request cycle, and can do inspections either before a servlet (HTTP) request is processed or after it has been processed. There are annotations that you use to apply

There is a Warp extension for JSF support. You'll notice that the preceding example uses the annotation `@AfterPhase` instead of `@BeforeServlet` or `@AfterServlet`. `AfterPhase` is provided by the JSF extension; it provides handling for additional phases that are specific to the JSF render cycle and other than the servlet ones; all are mapped back to the appropriate JSF lifecycle event for HTTP processing.

Grouping requests

Grouping requests leverages distinct observations to handle unique validations. With the way Warp operates, you perform your assertions on a per-request basis. These validations are matched to URLs that are being requested. This obviously doesn't work with JSF. JSF doesn't use URLs to determine views; it uses an internal view state to manage that and handle requests in certain orders.

In order to run, you keep with a single activity, but a multiple activity observes and inspects pairings in order to operate. In order to demonstrate this, we'll use a simple REST-based call structure to ensure data is shown as expected. We'll create a REST resource that generates a person's salutation, first name, and last name as a formatted string as the output of the REST call. Let's add this resource to our code base:

```
@RequestScoped
@Path("/people")
public class PeopleResource {
    @EJB
privatePeopleDAOpeopleDao;

@Produces("text/plain")
@Path("/email/{email}")
public String find(@PathParam("email") String email) {
        List<People> peeps = peopleDao.findByEmail(email);
if(peeps.size() > 0) {
            People p = peeps.get(0);
returnString.format("%s %s %s",p.getSalutation(),p.getFirstName(),p.
getLastName());
        } else {
return "none found.";
        }
    }
}
```

Based on this, if we pass in an e-mail address to the URL, we'll get back that person's name or **none found** if the e-mail address is not found in the system. If we want to test this out, we can add some assertions during the browsing. Using our driver, we can grab the source of the page and assert based on that content. In this test case, we'll try to access the REST API using two e-mail addresses. First, Jane Doe's, and then an account that doesn't exist. That test case ends up looking like this:

```
@Test
public void testWithGroups() {
  final String baseUrl = String.format("http://%s:%s",url.
getHost(),url.getPort());
  Warp
  .initiate(new Activity(){
    String baseRest = baseUrl + "/chapter7/rest/people/email/";
    @Override
    public void perform() {
      driver.get(baseRest+"jane.doe@anyplace.com");
      janeDoeContent = driver.getPageSource();
      driver.get(baseRest+"not.a.valid@email.net");
      notValidContent = driver.getPageSource();
    }
  })
  .group()
  .observe(new DefaultHttpFilterBuilder().uri().contains("jane.doe"))
  .inspect(new Inspection(){
    private static final long serialVersionUID = 1L;
    @AfterServlet
    public void testAfterServlet(){
      assertEquals("Ms. Jane Doe",janeDoeContent);
    }
  })
  .group()
  .observe(new DefaultHttpFilterBuilder().uri().
   contains("not.a.valid"))
```

```
      .inspect(new Inspection(){
        private static final long serialVersionUID = 1L;
        @AfterServlet
        public void testAfterServlet(){
          assertEquals("none found",notValidContent);
        }
      });
    }
```

In this case, one activity drives two different inspections. The first inspection will run based on `Jane Doe` since the e-mail address will appear in the URL, something along the lines of `http://localhost:8080/chapter7/rest/people/email/jane.doe@anyplace.com`. During the execution we capture the content of the page, which is short since it's a basic REST call that shows some simple text on a page. At any point a new group can be created for a single HTTP request that occurred. The groups are executed based on the HTTP requests made within your test code.

Spring MVC and Warp

Arquillian's plugin structure allows for dynamic loading as long as the plugin is on the classpath. This allows you to load multiple plugins at once during a single test case. If you are developing using Spring MVC, you can leverage the Spring extension as well as the Warp extension to build robust test cases.

You simply need to build your MVC application as you would normally do. Let's suppose the following controller:

```
@Controller
@RequestMapping("login.page")
public class LoginController {
  private static String SUCCESS = "loggedin";
  private static String SHOW_FORM = "login";
  private static String FAILURE = "login";
  private static String USER = "admin@admin.com";
  private static String PASS = "admin1";
  @RequestMapping(method = RequestMethod.GET)
  public String showForm(Map model) {
    LoginForm form = new LoginForm();
    model.put("loginForm", form);
    return SHOW_FORM;
  }
```

```
    @RequestMapping(value = "/login", method = RequestMethod.POST)
    public String login(@Valid LoginForm form, BindingResult result, Map
     model) {
      if(form.getUsername().equalsIgnoreCase(USER) && form.
      getPassword().equals(PASS)) {
        return SUCCESS;
      } else {
        return FAILURE;
      }
    }
  }
```

This controller is used for logging in to the application. It sets up the initial form and processes requests – in a very simple fashion, though that's more to drive home the point. We also have the form that drives this. We use some hibernate validator annotations to validate the fields.

```
  public class LoginForm {
    @NotEmpty
    @Email
    private String username;
    @NotEmpty
    @Size(min=1,max=30)
    private String password;
    //getters, setters omitted
  }
```

The UI is equally simple, a basic login form:

```
  <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
  <%@ taglib prefix="form" uri="http://www.springframework.org/tags/
  form"%>
  <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
  "http://www.w3.org/TR/html4/loose.dtd">
  <html>
  <head>
  <meta http-equiv="Content-Type" content="text/html;
  charset=ISO-8859-1">
  <title>Chapter8 Spring</title>
  </head>
  <body>
    <h3>Login Form</h3>
    <FONT color="blue">
```

```
      <h5>Login using admin@admin.com/admin1</h5>
    </FONT>
    <form:form action="login.page" commandName="loginForm">
      <table>
      <tr>
      <td>User Name:<FONT color="red"><form:errors
          path="username" /></FONT></td>
      <td><form:input path="username" /></td>
      </tr>
      <tr>
      <td>Password:<FONT color="red"><form:errors
      path="password" /></FONT></td>
      <td><form:password path="password" /></td>
      </tr>
      <tr>
      <td colspan="2"><input type="submit" value="Login" /></td>
      </tr>
      </table>
    </form:form>
  </body>
  </html>
```

This page will represent the form, which gets bound to the setup method (GET request) in MVC. Now that we have model, view, and controller defined, we can begin working on a test case. In this test case, we want to validate that the errors present include an invalid username and password. Warp's ability to do server-side assertions will be used to do that.

```
  @WarpTest
  @RunWith(Arquillian.class)
  @RunAsClient
  public class SpringWarpTest {
    @ArquillianResource
    URL baseUrl;
    @Drone
    WebDriver driver;
    @Deployment
    public static WebArchive createDeployment() {
      File[] libs = Maven
          .resolver()
          .loadPomFromFile("pom.xml")
          .resolve("org.springframework:spring-webmvc:3.1.1.RELEASE",
              "org.hibernate:hibernate-validator:4.2.0.Final")
```

```
            .withTransitivity().asFile();
             return ShrinkWrap
            .create(WebArchive.class, "chapter8-springmvc.war")
            .addPackage(LoginController.class.getPackage())
            .addAsWebInfResource("WEB-INF/web.xml", "web.xml")
            .addAsResource("applicationContext.xml")
            .addAsResource("views/login.jsp")
            .addAsLibraries(libs);
      }
      @Test
      public void testInvalidSubmission() {
        Warp.initiate(new Activity(){
                @Override
                public void perform() {
                    driver.get(baseUrl + "/chapter7-springmvc/");
                    driver.findElement(By.name("username")).clear();
                    driver.findElement(By.name("username")).sendKeys("not-
                    an-email");
                    driver.findElement(By.name("password")).clear();
                    driver.findElement(By.name("password")).sendKeys("");
                    driver.findElement(By.name("Login")).click();
                }
            })
            .observe(new RequestObserver(){})
            .inspect(new Inspection(){
                private static final long serialVersionUID = 1L;
                @SpringMvcResource
                private ModelAndView modelAndView;

                @SpringMvcResource
                private Errors errors;

                @AfterServlet
                public void testAfterServlet() {
                    assertEquals(2,errors.getErrorCount());
                }
            });
      }
    }
```

The setup should be very similar to what we're already used to. In the `Deployment` method we create our WAR file, including some external libraries that may be used. Then we build out all of the needed files (views, classes) that make up the WAR. Finally, we have our test case.

The Warp-Spring extension provides some additional injection points to inject Spring MVC. If we wanted to, we can also test out the case where we successfully logged in to the application. In order to avoid repeated code, we should refactor the inspection and the activity to be private classes, so that the test methods can simply invoke them. The activity should take username/password arguments and the inspection should take a number of errors to inspect. This is the result:

```
@Test
public void testValidLogin() {
  Warp.initiate(new LoginActivity("admin@admin.com", "admin1"))
      .observe(new RequestObserver() {
      }).inspect(new ErrorInspection(0));
}
private class ErrorInspection extends Inspection {
  private static final long serialVersionUID = 1L;
  private int numberOfErrors;
  public ErrorInspection(int numberOfErrors) {
    this.numberOfErrors = numberOfErrors;
  }
   @SpringMvcResource
  private ModelAndView modelAndView;

  @SpringMvcResource
  private Errors errors;

  @AfterServlet
  public void testAfterServlet() {
    assertEquals(numberOfErrors, errors.getErrorCount());
  }
}
private class LoginActivity implements Activity {
  private String username;
  private String password;
   public LoginActivity(String username, String password) {
    this.username = username;
    this.password = password;
  }
  @Override
  public void perform() {
    driver.get(baseUrl + "/chapter8-springmvc/");
    driver.findElement(By.name("username")).clear();
    driver.findElement(By.name("username")).sendKeys(username);
```

```
        driver.findElement(By.name("password")).clear();
        driver.findElement(By.name("password")).sendKeys(password);
        driver.findElement(By.name("Login")).click();
      }
    }
```

This test structure allows us some reusability. The `LoginActivity` can be used in other test cases if it's needed. The first test case then looks like this:

```
    @Test
    public void testInvalidSubmission() {
      Warp.initiate(new LoginActivity("not-an-email", ""))
          .observe(new RequestObserver() {
          }).inspect(new ErrorInspection(2));
    }
```

With all of this, we now have the infrastructure to start testing out our web application. Other portions that may need access to logging in (perhaps as a prerequisite) can use this class, as long as we make it more accessible by increasing visibility to `public` or `package` level.

Rich application testing with Graphene

Graphene is a two-pronged type of API which simplifies the building of Rich UI-oriented test cases as well as pure UI validation – the DOM of the page is browsed in order to locate elements and assert their values.

In order to demonstrate Graphene, we're going to make a few changes to our application. You will see them if you browse to `http://localhost:8080/chapter7/graphenePeople.html` which will give you a simple `jQuery + REST` styled application for maintaining people. We also need to add a few more methods to our REST resource:

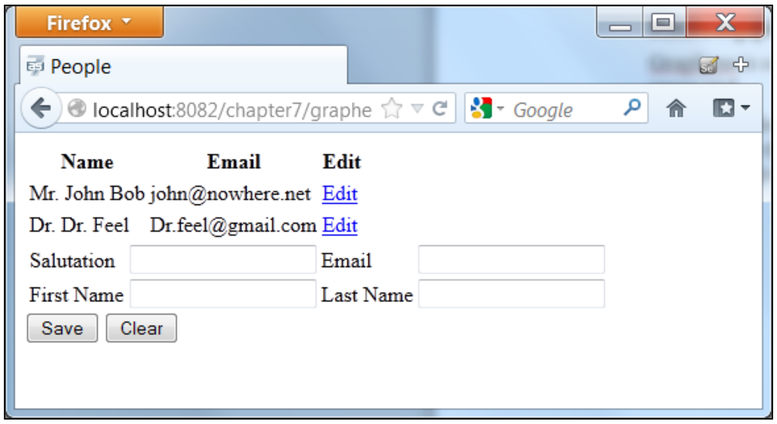
```
    @Produces("application/json")
    @Path("/list")
    @BadgerFish
    @GET
    public List<People> list() {
      returnpeopleDao.findAllPeople();
    }

    @Produces("application/json")
    @Path("/find/{personId}")
    @BadgerFish
    @GET
```

```
public People find(@PathParam("personId") Integer personId) {
  returnpeopleDao.find(personId);
}

@Consumes("application/x-www-form-urlencoded")
@Produces("application/json")
@Path("/save")
@BadgerFish
@POST
public Response update(@FormParam("personId")Integer personId,
    @FormParam("email")String email,
    @FormParam("firstName")String firstName,
    @FormParam("lastName")String lastName,
    @FormParam("salutation")String salutation
    ) {
  People person = new People();
  if(personId != null &&personId> 0) {
    person.setPersonId(personId);
  }
  person.setEmail(email);
  person.setFirstName(firstName);
  person.setLastName(lastName);
  person.setSalutation(salutation);
  peopleDao.savePeople(person);
  returnResponse.ok().build();
}
```

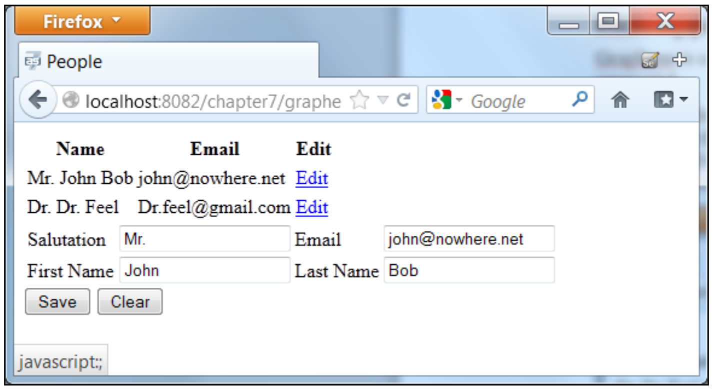
Once that's available, we'll create a simple HTML page with some JavaScript using jQuery to handle our UI interaction. It will look at our page and perform HTTP requests against the REST API. By default, you'll see a screen like this:



Name	Email	Edit
Mr. John Bob	john@nowhere.net	Edit
Dr. Dr. Feel	Dr.feel@gmail.com	Edit

Salutation		Email	
First Name		Last Name	

If you click on **Edit**, the selected row will be shown in the edit form, which will load asynchronously, not via a page refresh.

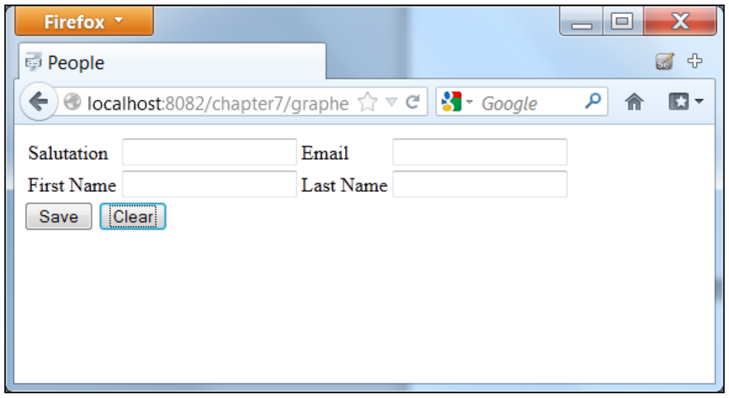


Name	Email	Edit
Mr. John Bob	john@nowhere.net	Edit
Dr. Dr. Feel	Dr.feel@gmail.com	Edit

Salutation	Mr.	Email	john@nowhere.net
First Name	John	Last Name	Bob

Save Clear

In addition, we can click on **Clear** to remove all rows from the table. This can be used if you want to hide the table.



Salutation		Email	
First Name		Last Name	

Save Clear

Graphene's goal is to help with application testing, but it provides read-only access to the content of the pages. It allows you to test your browser interactions as you would navigate around pages. First, we can navigate to the page and verify that we see the default row:

```
@Drone
private GrapheneSelenium driver;

@Test
public void testShowPeople() throws MalformedURLException {
  driver.open(new URL(url,"/chapter7/graphenePeople.html"));
```

```
    JQueryLocator peopleList = jq("#peopleList");
    JQueryLocator rows = peopleList.getChild(jq("tr"));
    int count =driver.getCount(rows);
    assertEquals("Wrong count",2,count);
  }
```

The test case uses the custom Graphene driver to perform functions that would be used in jQuery to handle page rendering and actions. This process tells the page to open the homepage and look up the table, then validate that there are exactly two rows, one being the header and the other being the content. Next, we can click on the **Edit** link and validate that the following form shows content. We would expect that clicking on the **Edit** link for the first row would show that row in the form. We can validate it this way:

```
  @Test
  public void testClickEdit() throws Exception {
    driver.open(new URL(url,"/chapter7/graphenePeople.html"));
    driver.click(jq("#edit1"));
    assertEquals("1",driver.getValue(jq("#personId")));
    assertEquals("John",driver.getValue(jq("#firstName")));
    assertEquals("Bob",driver.getValue(jq("#lastName")));
    assertEquals("john@nowhere.net",driver.getValue(jq("#email")));
  }
```

If we want to, we can click and edit. This should mirror some user clicking on edit and changing the first name to include the word `Boy` in it.

```
  @Test
  public void testClickEditAndChange() throws Exception {
    driver.open(new URL(url,"/chapter7/graphenePeople.html"));
    driver.click(jq("#edit1"));
    assertEquals("1",driver.getValue(jq("#personId")));
    assertEquals("John",driver.getValue(jq("#firstName")));
    assertEquals("Bob",driver.getValue(jq("#lastName")));
    assertEquals("john@nowhere.net",driver.getValue(jq("#email")));
    driver.type(jq("#firstName"), "Boy");
    driver.click(jq("#save"));
  }
```

We use the jQuery selector to find the appropriate element to change and the driver's type method to enter the data. Finally, we click on the **Save** button to save our changes.

Captures

Graphene allows us to capture two additional types of output from our test. The first is a full dump of network traffic, which we can filter based on type. The second is screenshots of the test case.

The network traffic is very important since we can use it to show what requests/responses were made and what was in them. In order to enable it, take a look at the following example:

```
@Test
public void testWithNetwork() throws Exception {
  NetworkTrafficjsonTraffic = driver.captureNetworkTraffic(NetworkTraf
ficType.JSON);
  // do work..
  System.out.println(jsonTraffic.getTraffic());
}
```

At the start of the method, we turn on network traffic and at the end, we retrieve the content. If we want to, we can capture multiple types like this:

```
@Test
public void testWithMultipleNetwork() throws Exception {
  NetworkTrafficjsonTraffic = driver.captureNetworkTraffic(NetworkTraf
ficType.JSON);
  NetworkTrafficplainTraffic = driver.captureNetworkTraffic(NetworkTra
fficType.PLAIN);
  // do work..
  System.out.println(jsonTraffic.getTraffic());
  System.out.println(plainTraffic.getTraffic());
}
```

The next output can be a buffered image. While not natively displayed here, the image can be output to your test directory and shown with the tests. This can be useful if you want to capture snapshots of your test case as its running.

```
@Test
public void testOpenPageAndScreenCap() throws Exception {
  driver.open(new URL(url,"/chapter7/graphenePeople.html"));
  BufferedImagehomePageImage = driver.captureScreenshot();
  //output image to file
}
```

We simply open a page of our application and then capture a screenshot. The image can be output to a file at this point. It will be a PNG type. While not an automated validation, it adds necessary assurance to the business owner describing what was shown, so that the full acceptance test case can be proven visually. Performing a screenshot is standard practice when executing test cases as a way to document what was shown. This also gives time to clean up any issues; the graphics can be modified to show changes needed.

Note that there are certain limitations, so please consult the manual. At the time of authoring, it only worked in Firefox.

Selectors and interactions

As noted, we're using this method called `jq` to do a lot of work. This `jq` method is actually a wrapper for jQuery-styled selectors. It will allow for any jQuery call that could be used in the browser. All of your standard selectors are supported, such as by ID, by class, by element type.

Graphene also allows you to perform user interactions. This can be drag-and-drop actions, mouse-over effects, or even page scrolling. Here's a quick example:

```
@Test
public void testInteractions() throws Exception {
  driver.altKeyDown(); //press the alt key
  driver.type(jq("#email"), "jane.doe@gmail.com");//find the elment
email, set the value to jane.doe@gmail.com
  driver.scrollIntoView(jq("#somePageElement"), true); //scroll to the
given element's location on the page.
  driver.click(jq("#save")); //click the save button.
  driver.mouseOver(jq("#lowerDiv")); //moves mouse over the lowerDiv
element.
  driver.check(jq("#isEligible")); //checks the checkbox isEligible
}
```

Graphene allows you to also extend the functionality of your page. This can allow closer inspection of JavaScript content to ensure that the behavior is as expected. This can help troubleshoot or debug your pages.

```
@Test
public void testUsingJavascript() throws Exception {
  JavaScript js = JavaScript.js("(\"#peopleListCont\").empty();");
  js.append("\n").append("window.alert('Hello, world!');");
  // do more work.
  driver.addScript(js);
}
```

In this case, we're doing some basic events – clearing out our data table and creating a window alert. However, after this we can add an assertion that the window alert is shown:

```
driver.addScript(js);
// do more work..
assertTrue(driver.isAlertPresent());
```

The Graphene extension is quite powerful. Not only does it execute test cases but it can also validate what is being shown to the end user. This is along the lines of automated acceptance testing. Since we can perform browser-level interaction in a rich fashion, we can behave like a real user would behave when interacting with the application.

Summary

As we've seen, by using the functional test tools of Arquillian we can automate the execution of browser navigation. Arquillian Drone provides a lot of base functionality to perform functional application testing. Using Drone, the team was able to create additional extensions for server-level assertions as well as Graphene for client-level assertions. In the next chapter, we'll start reviewing how to work with web services and REST APIs and better work within Arquillian. We'll also work a bit with EJB testing and how to perform some test cases that invoke web services directly with their Java APIs.

8
Service Testing

When looking at service-oriented architectures, we have to consider automated testing scenarios. In most cases, it may not be as simple as deploying a single artifact to your container to do testing. It will depend on how your project is structured and what artifacts may already be deployed.

This chapter will show off some deployment techniques as well as how to use soapUI with your Arquillian test cases to perform more automation, for both SOAP and REST based APIs. We'll also review how to test interactions across modules in an EAR.

Service testing approaches

There are many ways you can tackle testing a web service using Arquillian. The most straightforward way is to create a client stub to handle the web service integration and invoke it from a client. You can also get more sophisticated and use soapUI. Either way, this type of testing is designed for client-mode type tests. Your test cases will not run in the container, but instead run on the client that is doing the deployment. This will give you a better feel for how your application would fare across the network.

This chapter makes heavy use of two key components of the Arquillian API. The first is the annotation `@RunAsClient`, which can be placed on either a class or a method. This annotation tells Arquillian to not run this test case in the container and instead run it in the local JVM. For use within this chapter, `@RunAsClient` is paired with `@Deployment(testable=false)`, which is used on the deployment. We may want to turn off Arquillian enrichment of the deployed resource when using `@RunAsClient` to avoid any of the Arquillian runtime being present when you run your test code.

The second piece of API is `@ArquillianResource` placed on a `URL` object. This can be done as either injection into the test method being invoked or as an injection point to the class under test. Injecting this into a calling method will ensure that the Arquillian runtime chooses the right URL to inject based on the deployment specified (for a single deployment test this is the default behavior). When injected as a parameter to a class, it will be populated based on the method being invoked.

Web service stubs

The most straightforward approach to testing your web service is to use a generated client stub and invoke it directly against your Arquillian deployment. This approach assumes that you have a WSDL available to generate a client, or are expecting a client to be provided for use within the test case. Once you have those client stubs together, you can build a test case as simple as this:

```
@Test
@RunAsClient
public void testWebServiceViaClient(@ArquillianResource URL url)
throws Exception {
  URL stubUrl = new URL(url,"/Services/MySoapService");
  WebServiceStub stub = new WebServiceStub(stubUrl);
  Result r = stub.performOperation();
  assertSomethingAboutTheResult(r);
}
```

`WebServiceStub` represents the client generated by your choice of client tools. In this test case, we use the URL of the deployment to build an instance of the web service stub. This will allow you to invoke the web service using a client that should match the way other users may invoke the web service.

However, since you're handling the client invocation yourself here, you need to ensure that Arquillian only handles deployment. As a result, your `@Deployment` method should be marked as `testable=false`, which will turn off Arquillian extending the deployment. We're only using Arquillian on the client side to perform some injection; on the server its only interaction is to do the deployment.

This approach also assumes that you are building a client. This makes more sense if you distribute a client library with your application that makes it easier to integrate.

This approach also works with REST APIs. Using client APIs such as those provided by Jersey or RESTEasy, as well as a lower-level one using Apache HTTP client, you can build out a client call to your deployed service.

Application invocation

Another approach you can take, as long as your deployment methodology supports it, is to directly call your business tier. This may work if your component behind the scenes is actually an `EJB` or a `CDI` component, but may not work as well if you are using web services over Servlet binding or a direct REST call instead of using `CDI` or `EJB` components. Think first about this `EJB`:

```
@Stateless
public class TemperatureConverter {
  public double convertToCelsius(double fahrenheit) {
    return (fahrenheit - 32)*5/9;
  }
  public double convertToFahrenheit(double celsius) {
    return (celsius*9) / 5 + 32;
  }
}
```

It's pretty straightforward if you look at it. Also, the test case for testing out this `EJB` is pretty simple:

```
@EJB
private TemperatureConverter converter;

public void testConvertToCelsius() {
  double celsius = converter.convertToCelsius(32);
  Assert.assertEquals(0,celsius);
}

public void testConvertToFahrenheit() {
  double fahrenheit = converter.convertToFahrenheit(0);
  Assert.assertEquals(32,fahrenheit);
}
```

Now, if this class were instead exposed as a web service, it would have `@WebService` on it. But what is the logical difference between that `@WebService` annotated bean and a regular `EJB`? There shouldn't be any other than the transport mechanism; all that the server does is translate the XML request in to an `EJB` invocation. This may not qualify as a complete integration test, but this should suffice for unit testing of your web services.

Using your implementations to test directly also saves a bit of work when the API changes slightly. With soapUI or stub testing, you need to regenerate code to account for the new API objects.

Compare and contrast

Both of these examples leverage some alternative ways that your application may be invoked. You do have to ask and ensure that there is agreement that this is a valid and useful way to test the code. This may not be feasible for your project for a few reasons. One may think that a web service client is not automatically generated; as a result, code changes could be out of sync with the test case. Perhaps it's decided to use a dedicated tool for the web service invocation so that it remains platform independent; soapUI is just posting XML documents and returning the output XML to the caller. If you do have a stub, feel free to use it. Inspecting the object may give you more fine-grained results.

Using soapUI

If you haven't worked with it before, here's a quick introduction to soapUI. soapUI (`http://www.soapui.org`) is a Java-based application for testing services—REST APIs, SOAP, JMS.

Once you have a WSDL, you can use soapUI to build test cases for your web service. Let's say we start with the WSDL provided with the sample project `convert.wsdl`. Go ahead and start soapUI. Once running, go to **File | New soapUI Project**. Then use soapUI to point to the WSDL in the sample code. (Alternatively, you can point to the soapUI project in `src/test/resources/soapui/TemperatureConverter-soapui-project.xml`.) You can start to poke around the service, create sample inputs, and look at the responses. Once you have a valid set of requests, as well as your web service built and deployed on your application server, go ahead and invoke the test stub. In general, this should work fine.

soapUI and Arquillian

Another way you can work with soapUI is by using it as a test-level dependency. First, let's add these dependencies to our Maven project:

```
<dependency>
   <groupId>eviware</groupId>
   <artifactId>maven-soapui-plugin</artifactId>
  <version>4.5.0</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>com.fifesoft</groupId>
  <artifactId>rsyntaxtextarea</artifactId>
  <version>1.4.1</version>
  <scope>test</scope>
</dependency>
```

Then, we can create the following test method in our code:

```
@Test
public void runViaSoapUI() throws Exception {
  SoapUITestCaseRunner runner = new SoapUITestCaseRunner();
  runner.setProjectFile("src/test/resources/soapui/
  TemperatureConverter-soapui-project.xml");
  runner.run();
}
```

This will initialize the soapUI runner and execute a specific project file. Now note that your project file, in order to run properly, must have test cases and assertions defined. Without those, results will not be provided.

Prior to executing your test case, there are several properties you should consider setting. Here they are:

```
runner.setOutputFolder("target/surefire-reports");
runner.setJUnitReport(true);
runner.setPrintReport(true);
```

The first will place your test results into the default directory for Maven tests. This will allow your CI server to pick up on them easily after running. The second property says to export JUnit test reports. This is good for monitoring your test status. The last property will add output like this to your console window. It's also good if you are running in command-line mode (or from Eclipse) and want to see the results:

```
SoapUI 4.5.0 TestCaseRunner Summary

-----------------------------

Time Taken: 1027ms

Total TestSuites: 1

Total TestCases: 2 (0 failed)

Total TestSteps: 2

Total Request Assertions: 3

Total Failed Assertions: 0

Total Exported Results: 0
```

Still, we're missing one critical piece. We had to have our web service up and running to execute the tests. This is only good if we can guarantee that the latest code was installed and these tests handle that code. Here's where Arquillian can help. Since you have built the web service using a standard Java technology (such as JAX-WS), you can create a deployment for it and deploy it to an application server prior to executing. Furthermore, you can inject the remote URL into your test case in order to dynamically choose the host/port that it runs from.

1. First, we add the Arquillian JUnit runner to our test class:

   ```
   @RunWith(Arquillian.class)
   ```

2. Next, we create a deployment method:

   ```
   @Deployment(testable = false)
   public static Archive<?> createTestArchive() {
     return ShrinkWrap.create(JavaArchive.class, "chapter6.jar")
         .addClass(TemperatureConverter.class);
   }
   ```

3. Next, we add an injection point for the URL:

   ```
   @ArquillianResource
   private URL serverUrl;
   ```

4. Then we work on our test method. We're going to change two things. First, we're going to mark the test run as client—this way it runs on the local system and can use Arquillian injection. Next, we're going to set the host/port in the test case based on the URL information. Our final test case ends up looking like this:

   ```
   @Test
   @RunAsClient
   public void testConvertUsingSoapUI() throws Exception {
     SoapUITestCaseRunner runner = new SoapUITestCaseRunner();
     runner.setProjectFile("src/test/resources/soapui/
     TemperatureConverter-soapui-project.xml");
     runner.setHost(String.format("%s:%s",serverUrl.
     getHost(),serverUrl.getPort()));
     runner.setOutputFolder("target/surefire-reports");
     runner.setJUnitReport(true);
     runner.setPrintReport(true);
     runner.run();
   }
   ```

When you run this, assuming you target something such as JBoss AS 7, you can have a fully automated test case based on your web service testing and soapUI.

soapUI and REST APIs

Using soapUI you can also test out your REST APIs. This does require that you have a **WADL** (**Web Application Description Language**) published that includes your object structure, which simply makes it easier to test out individual methods.

First, you will need to add the additional goals to your pom to generate the WADL. You can pull that from the sample code; it's quite lengthy to include here. You will also need to copy the additional resources referenced in the `pom` file that are included in the same code (`chapter8-rest`). Once done, you can run the following Maven command to create the WADL for your REST resource. I used a simple REST resource based on the Temperature Converter to do that.

```
mvn compile javadoc:javadoc com.sun.jersey.contribs:maven-wadl-
plugin:generate
```

This will result in the `javadocs` as well as the WADL being generated. There is a lot of configuration to what was put in to your `pom.xml`, but here's a brief rundown:

- Configuration for the `javadoc` plugin creates a `resourcedoc.xml` that references the `javadocs` of your code
- Using local configuration plus the generated `javadocs`, creates a WADL descriptor explaining your resources as well as how to interact with them

You should see it in `target/classes` when completed. From here, you can go back in to soapUI and import this WADL when creating a new project. You can browse to the endpoints you have defined and create test cases for them, in much the same way you did for the web service. Here's what my REST API looks like:

```
@Path("/convert")
@Stateless
public class TemperatureConverter {

  @GET
  @Produces("text/plain")
  @Path("/fahrenheit/{far}")
  public double convertToCelsius(@PathParam("far") double fahrenheit)
  {
    return (fahrenheit - 32)*5/9;
  }

  @GET
  @Produces("text/plain")
  @Path("/celsius/{cel}")
  public double convertToFahrenheit(@PathParam("cel")  double celsius)
  {
    return (celsius*9) / 5 + 32;
  }
}
```

The methods are identical to the web service methods, except for annotations. Now you could actually annotate your web service with the REST API annotations to reduce coding effort, but that was not the goal here; we are looking at only the REST methods right now. When I generate the WADL for this API, I get the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<ns2:application xmlns:ns2="http://wadl.dev.java.net/2009/02">
  <ns2:doc jersey:generatedBy="Jersey: 1.16 11/28/2012 02:09 PM"
    xmlns:jersey="http://jersey.java.net/" />
  <ns2:doc title="The doc for your API" xml:lang="en"><![CDATA[
This is a paragraph that is added to the start of the generated
application.wadl]]></ns2:doc>
  <ns2:grammars />
  <ns2:resources base="http://hostname:8080/chapter6-rest">
    <ns2:resource path="/convert">
      <ns2:doc><![CDATA[The Class TemperatureConverter.
 Converts from fahrenheit to celsius, and then celsius back to
 fahrenheit]]></ns2:doc>
      <ns2:resource path="/fahrenheit/{far}">
        <ns2:param name="far" style="template" type="xs:double"
          xmlns:xs="http://www.w3.org/2001/XMLSchema">
          <ns2:doc><![CDATA[the fahrenheit]]></ns2:doc>
        </ns2:param>
        <ns2:method id="convertToCelsius" name="GET">
          <ns2:doc><![CDATA[Convert to celsius.]]></ns2:doc>
          <ns2:response>
            <ns2:representation mediaType="text/plain" />
          </ns2:response>
        </ns2:method>
      </ns2:resource>
      <ns2:resource path="/celsius/{cel}">
        <ns2:param name="cel" style="template" type="xs:double"
          xmlns:xs="http://www.w3.org/2001/XMLSchema">
          <ns2:doc><![CDATA[the celsius]]></ns2:doc>
        </ns2:param>
        <ns2:method id="convertToFahrenheit" name="GET">
          <ns2:doc><![CDATA[Convert to fahrenheit.]]></ns2:doc>
          <ns2:response>
            <ns2:representation mediaType="text/plain" />
          </ns2:response>
        </ns2:method>
      </ns2:resource>
    </ns2:resource>
  </ns2:resources>
</ns2:application>
```

This is similar to a WSDL, except it is providing the resources (rather than methods) and the types from the request/response. This does provide a full listing, so if we had concrete objects those would be listed here in the return types.

Finally, when we're ready to test this code, the calls are very much the same. In fact, the test method is the exact same. Our deployment is a little different—in order to enable JAX-RS in your archive you must deploy a WAR file.

```
@Deployment
public static Archive<?> createArchiveWithResolver() {
  return ShrinkWrap.create(WebArchive.class, "chapter8-rest.war")
      .addClasses(JaxRsActivator.class, TemperatureConverter.class);
}

@ArquillianResource
private URL serverUrl;

@Test
@RunAsClient
public void testConvertUsingSoapUI() throws Exception {
  SoapUITestCaseRunner runner = new SoapUITestCaseRunner();
  runner.setProjectFile("src/test/resources/soapui/
TemperatureConverter-soapui-project.xml");
  runner.setHost(String.format("%s:%s", serverUrl.getHost(),
      serverUrl.getPort()));
  runner.setOutputFolder("target/surefire-reports");
  runner.setJUnitReport(true);
  runner.setPrintReport(true);
  runner.run();
}
```

The `TemperatureConverter` file here is actually the REST one rather than the SOAP one. But, as you can see, the structure is the same for the test and the deployment only has the difference of being in a WAR. We also needed a second class – simply to enable JAX-RS within our application.

Other REST tests

Another way to look at REST testing is the same as we saw before with directly testing your RESTful service. If it is a CDI bean or EJB then it becomes very easy to test the underlying object.

HTTP client testing

Another way to test is using an HTTP client. This isn't unique to REST but makes the most sense using REST. While the two leading JAX-RS implementations (Jersey and RESTeasy) both provide client APIs and the JAX-RS 2.0 expert group are working to define a REST client API, I find it most appropriate to discuss this approach using Apache HTTP components.

We'll start with our same REST resource in the temperature converter. Based on its definition, it should have the following URLs:

- `http://hostname:8080/chapter8-rest/convert/fahrenheit/{far}`
- `http://hostname:8080/chapter6-rest/convert/celsius/{cel}`

Both of these URLs are also GET requests. As a result, we can simply use the `HttpGet` API. Using the same Deployment method for the other test cases, we can construct this test method to test out our resource:

```
@ArquillianResource
private URL serverUrl;

@Test
@RunAsClient
public void testFahrenheitHttpClient() throws URISyntaxException,
ClientProtocolException, IOException {
  HttpClient httpclient = new DefaultHttpClient();
  try {
    URL fUrl = new URL(serverUrl, "/chapter6-rest/convert/
    fahrenheit/32");
    HttpGet get = new HttpGet(fUrl.toURI());
    ResponseHandler<String> responseHandler = new
    BasicResponseHandler();
    String responseBody = httpclient.execute(get, responseHandler);
    double d = Double.valueOf(responseBody);
    Assert.assertEquals(0,d);
  }
  finally {
    httpclient.getConnectionManager().shutdown();
  }
}
```

This invocation will hit your REST resource and process the result of its call as a double and compare it to 0—which is what 32 Fahrenheit is in Celsius. Likewise, we can add the equivalent Celsius conversion to the tests:

```
@Test
@RunAsClient
public void testCelsiusHttpClient() throws URISyntaxException,
    ClientProtocolException, IOException {
  HttpClient httpclient = new DefaultHttpClient();
  try {
    URL fUrl = new URL(serverUrl, "/chapter6-rest/convert/celsius/0");
    HttpGet get = new HttpGet(fUrl.toURI());
    ResponseHandler<String> responseHandler = new
    BasicResponseHandler();
    String responseBody = httpclient.execute(get, responseHandler);
    double fahrenheit = Double.valueOf(responseBody);
    Assert.assertEquals(32,fahrenheit);
  }
  finally {
    httpclient.getConnectionManager().shutdown();
  }
}
```

You can also test using other HTTP methods, such as PUT, POST, and DELETE using the HTTP client. One such way would be to do an HTTP PUT to insert a record, followed by an HTTP GET to get that record by its ID to ensure that it was saved properly.

Testing JMS

JMS testing is still tricky, but still possible as long as you have the right expectations going in to the scenario. You should treat your message listener as a transport mechanism; in the same way a web service should delegate work, the JMS listener should translate the `javax.jms.Message` object in to the appropriate business object and perform work against that.

Obviously, you want to continue to separate your code for better reuse and looser coupling between components. In this use case, let's assume that you want to be able to create people. This involves synchronous and asynchronous processing. We're going to support processing XML messages that contain data for a person to be inserted or updated. First though, we need to consider the internal APIs. Instead of trying to fit this logic all within a `MessageListener` implementation, we need to have some kind of a manager that will create or update records.

First, we have our entity, similar to the entities we've seen already:

```
public class Person {
    @Id
    private int personId;
    private String name;
    private String title;
    private int age;
  // getters, setters ommitted.
}
```

And our manager class:

```
public class PersonManager {
    public int createOrUpdate(Person person) {
        // logic
        return … ;
    }
}
```

Now, in order for our testing to be complete we need to test out the manager class as well. The most basic tests for this manager class should be like this:

```
@Inject
private PersonManager mgr;
@Test
public void testCreatePerson() {
  Person p = new Person();
  // populate fields
  Assert.assertTrue(mgr.createOrUpdate(p) > 0);
}
@Test
public void testUpdatePerson() {
  Person p = mgr.getPerson(1);
  // change some fields
  Assert.assertEquals(p.getPersonId(),mgr.createOrUpdate(p));
}
```

This tests the basic functionality of the manager – creating or updating a person that is passed in. We should also have a translator that takes the String and converts it into a person. JAXB is a pretty good tool to do this simple parsing of XML into objects.

```
public class Translator {
    private JAXBContext jaxbContext;
    public Person translatePerson(String text) throws JAXBException {
        return (Person)jaxbContext.createUnmarshaller().unmarshal(new
        StringReader(text));
    }
}
```

Obviously, this class would also need its own testing. Now, that we have some reusable code behind it, we can introduce the MDB that will handle this processing:

```
public class PersonMDB implements MessageListener {
    @Inject
    private Translator translator;
    @Inject
    private PersonManager personManager;
    public void onMessage(Message message) {
        try{
            if(message instanceof TextMessage) {
                String body = ((TextMessage)message).getText();
                int personId = personManager.
                createOrUpdate(translator.translatePerson(body));
                if(message.getJMSReplyTo() != null) {
                    // reply to the message.
                }
            }
        } catch(Exception e) {
        }
    }
}
```

So the code in here is actually simple now. We just need to call the methods in the objects built to do work. If we provide testing of those components directly, we can have fairly strong belief that it will pass integration testing. We can test the MDB directly, either by mocking out a direct call to the class with a fake `TextMessage` or by using the JMS APIs to send a message to the queue that it listens to.

Testing Seam 2

Seam 2 has a dedicated extension for testing components. In addition, with the introduction of Seam 2.3, Arquillian is now the default test runtime for Seam 2 applications. Some of the Seam Test APIs are still in use and available for use within a Seam-based test. You'll want to continue to use your JBoss AS 7 container and create deployment archives as normal. You should only need the extension if you are still using Seam earlier than 2.3 in your application.

Seam 2.3 assumes that you are using a typical EAR deployment for your archive. You should have EAR, WEB, and EJB projects to cover your production code. You should also have a TEST module next to your EAR project, which comes last in the module list. This module, since its built last, will be able to refer to your EAR project to grab the primary deployment. We can manually build the archive, but we would prefer not to, due to the complexities of the content (list of libraries, list of classes required). Your deployment would then look like this:

```
@Deployment
public static Archive<?> createDeployment() {
  EnterpriseArchive er = ShrinkWrap.create(ZipImporter.class)
    .importFrom(new File("../myapp-ear/target/myapp.ear"))
    .as(EnterpriseArchive.class);
  WebArchive web = er.getAsType(WebArchive.class, "myapp-web.war");
  web.addClasses(SimpleTest.class);
  return er;
}
```

In `SimpleTest`, you just need to inject some components and work with them. Since you're deploying the entire EAR, you have access to all components within your project. If you have a similar `TemperatureConverter` like I do, your test looks just like this:

```
@RunWith(Arquillian.class)
public class SimpleTest {
  //your deployment
  @In
  private TemperatureConverter converter;
  @Test
    public void testConvertToCelsius() {
        double celsius = converter.convertToCelsius(32);
        Assert.assertEquals(0,celsius);
    }
}
```

So luckily, at this point, your test is just like any other Arquillian test.

Summary

As you can see from this chapter, it's not too difficult to work with services of various types in Arquillian—EJBs, Web Services, REST APIs are all easily testable. We do see that it becomes easier to use a tool with Arquillian to create the tests; but we also get to use Arquillian for strong use cases: the automatic startup and deployment of the application server with archives. This is the first of functional testing with Arquillian. These test cases are meant to test out functional areas of your application, use cases or user stories. They are truly end to end; they call the web service as deployed to the server and expect full results. With the EJBs we also see them running deeply through the application server including database connectivity (if used).

In the next chapter, we'll look into testing OSGi using JBoss OSGi on JBoss AS 7.1. We'll get to see some of the custom injection points provided by the OSGi enricher within JBoss and how deployments work a little differently here. We'll also see some more generic OSGi usage in GlassFish by creating regular archives that can be used with their OSGi runtime.

9 Arquillian and OSGi

In this chapter, we present a different type of deployment with Arquillian and show how you can test your OSGi bundles using Arquillian and JBoss OSGi. We will also review how you would deploy OSGi-based bundles into other containers that support OSGi, such as GlassFish, and the advanced capabilities that you receive by doing this.

While OSGi has many uses and runtimes available, using any of them requires deep integration into the application server. The primary use in this chapter is around JBoss OSGi, which provides integration to the application server for Apache Felix.

OSGi and Arquillian

OSGi support within Arquillian is primarily focused on JBoss OSGi. When you test an application using the OSGi, both an embedded and a managed application server deployment are supported. The embedded runtime is purely OSGi, though you could add support for other components manually. The embedded runtime is best for testing out plain embedded OSGi modules. The full application server deployment should be used for any deployments that leverage EJBs or CDI components. The JBoss OSGi runtime in embedded mode may help you run tests in your Java SE application that uses JBoss OSGi; this would have to be your container in the SE mode.

This chapter focuses primarily on how to use the Arquillian OSGi extension to test using JBoss OSGi in your application. We do end up using an early release of the next version of the framework, since support is very limited in the current release.

What is OSGi?

Open Services Gateway Initiative (**OSGi**) is a packaging and bundle standard that allows you to control what modules your module may depend on, what packages you export, and what packages you import into your module. Using OSGi allows you to control at a very fine-grain level what your runtime dependencies shall look like and where you are safe to make internal changes versus external changes.

While OSGi allows you to have this fine-grained control, it does not define capabilities that may compare with standard Java EE technologies such as CDI, EJB, or JMS. It does define how packages may interact with these APIs though.

Arquillian natively supports the JBoss OSGi runtime; however, you are not limited to only using OSGi here. You can also deploy modules to other containers, such as GlassFish, that do support OSGi. You will be limited on what you can and cannot inject into your test case when using other containers; you will be limited to the injection capabilities of that container when using their OSGi runtime.

JBoss OSGi

JBoss OSGi is the OSGi provider that comes with JBoss AS 7.x. It is a newer tool, making its grand entrance in the AS7 series and since then has been upgraded for the 7.1 release.

OSGi versus everything else

OSGi support in Arquillian is a bit different than the way other frameworks and containers work. OSGi is part container and part framework in nature, so it works out, but does cause some confusion in use.

When you test using JBoss OSGi, it acts as both a container and a test enricher. When you use the container, you have the option to work with the embedded OSGi container, or use a full JBoss instance. When you use OSGi embedded, you only have access to the local resources. If you want to add CDI support, you'll need to add CDI to that embedded container. This can be done using the Maven assembly plugin to add more JARs to your classpath. In most cases you'll want the full application server, since the OSGi runtime with some custom JARs isn't typically how you would run in production; so it's not a very valid test case.

At the test-case level, you have access to a number of additional injection points. They include:

- `@Inject BundleContext context;`
- `@Inject Bundle bundle;`
- `@Inject Framework framework;`

These injection points allow you to control your bundles as well as the framework from within your test cases. One of the key things the enricher does is help you to start your bundle once it has been deployed.

OSGi versus JBoss Modules

The modular architecture of JBoss AS 7 is built on top of JBoss Modules. The dynamic classloader that it has allows you to have fine-grained control over what JAR files do and do not get loaded via a configuration file. Similar to OSGi support, it allows you define dependencies (modules that you need loaded) as well as exports, parts of your JAR that can be exported for use by other applications. It removes some of the intricacies of deployment handling for you. If you prefer to work closely with your deployments then OSGi is probably the route you want to go.

If your deployment target is JBoss AS 7, there are a few points you should consider when deciding whether or not to use JBoss OSGi or JBoss Modules directly. In OSGi, you can raise events, which can do anything from being triggered by an admin condition or on bundle startup. In regular Java EE, you can do this through context listeners (servlet container) or by creating a Singleton EJB that is annotated `@Startup` to capture when your container is starting. In OSGi, you need to have a custom `MANIFEST.MF` as well as an XML document describing your services. With JBoss Modules, you only need to define a `module.xml` (outside of your archive, if you are creating an actual module) or a `jboss-deployment-structure.xml` (inside of your archive).

Working with OSGi in a test case

While definitely not a course in how to program for OSGi deployments, there are certain differences between an OSGi archive in Arquillian and a standard Java archive. To understand, it may be worth reading a bit more in-depth about ShrinkWrap and Assets.

ShrinkWrap Assets

Throughout the book, and the sample code, you'll notice a lot of archives that include an asset. Assets are typically components that are not class files but get placed into an archive. A common one used throughout the book looks like the following:

```
archive.addAsManifestResource(EmptyAsset.INSTANCE,"beans.xml");
```

`EmptyAsset.INSTANCE` in this case is an Asset implementation that represents an empty file. The filename it will be assigned is `beans.xml` and will be placed into the manifest directory: in a Java archive this is `META-INF` and in a web archive this is `WEB-INF/classes/META-INF`. Those files are what enable CDI to run against your application.

Some other assets include `FileAsset` and `StringAsset`. The `FileAsset` is the default asset created when you call `archive.addAsResource("someString")` where `someString` will point to a resource file, under `src/main/resources or src/test/resources`. The `StringAsset` on the other hand is not implicitly created but instead is always explicitly created with the contents of what your asset should have. The equivalent of using an empty asset is to use the following string asset:

```
archive.addAsManifestResource(new StringAsset(""),"beans.xml");
```

OSGi adds an important asset to the suite, the `OSGiManifestBuilder` which represents the `MANIFEST.MF` file that will contain your OSGi integration information. You add it to your archive in the following manner:

```
archive.setManifest(OSGiManifestBuilder.newInstance()
         .addBundleSymbolicName(archive.getName())
         .addBundleManifestVersion(2)
         .addImportPackages(ServiceTracker.class,
          XRequirementBuilder.class, Repository.
          class, Resource.class)
         .addManifestHeader("Service-Component", osgiServices));
```

This asset is unique, as it represents the manifest file always. Other assets are more generic in nature, so it's fine if you consider this one as more of a descriptor rather than an asset. Using this asset will be required when you are building your dynamic OSGi modules. When building your full archive, you may use Maven to generate those settings, and if you are testing this as a part of an integration test (a separate build generates the JAR file), then it may be worthwhile to simply import that JAR file.

OSGi archives

OSGi Java archives are not very different than standard Java archives. The main differences involve the manual setting of the manifest file and and ensuring that certain files are set up properly within the archive. You would need to do this anyways as long as you properly package your application for OSGi usage. Observe the following deployment method, which creates the Java archive:

```
@Deployment
public static JavaArchive createDeployment() throws
FileNotFoundException {
  String osgiServices = "OSGI-INF/sample.xml";
  String archiveName = "example-archive";
  return ShrinkWrap.create(JavaArchive.class, archiveName)
.addClasses(SampleComparator.class, DeclarativeServicesSupport.class,
RepositorySupport.class)
.addAsResource(osgiServices)
.addAsManifestResource(RepositorySupport.BUNDLE_VERSIONS_FILE)
.setManifest(OSGiManifestBuilder.newInstance()
        .addBundleSymbolicName(archiveName)
        .addBundleManifestVersion(2)
        .addImportPackages(ServiceTracker.class,
         XRequirementBuilder.class, Repository.
         class, Resource.class)
        .addManifestHeader("Service-Component", osgiServices));
}
```

First, we have a couple of constant values – the service XML file that describes our bundle and the name of the bundle, separated so that we can refer to it while building the archive. We need to add a few classes to our archive, namely our service (`SampleComparator`) and the support classes that help publish it into OSGi framework. These all need to be packaged together into the archive so that the archive as a whole can be published. Arquillian does not enrich the classes used for deployment and testing, only the archive itself to include Arquillian for testing the support.

OSGi web archives are similar. You will need to add the same manifest entries to import the options into the modules. Similar to the problems you might see with Maven, it is not possible to export packages from a WAR file since the compilation path cannot be resolved externally through a WAR file. The WAR files are also unique, as they contain other JAR deployments which may expose importable services. In fact, this is a likely way that you will consume those services.

Testing a deployed service

Let's say we build a simple welcome service. The service can be as simple as follows:

```
public interface WelcomeService {
  public String welcome(String person);
}

public class WelcomeServiceImpl implements WelcomeService {

  public String welcome(String person) {
    return String.format("Welcome, %s!",person);
  }
}
```

The component deployed can be declared as a service in our XML document as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<component name="sample.component" immediate="true">
  <implementation class="com.tad.arquillian.chapter9.osgi.
  WelcomeServiceImpl" />
  <property name="service.description"
  value="A Welcoming Service" />
  <property name="service.vendor" value="TAD" />
  <service>
    <provide interface="com.tad.arquillian.chapter9.
    osgi.WelcomeService" />
  </service>
</component>
```

This document binds the implementation of the service to the interface and makes it available for use. We can then test the service using the following test case:

```
@Test
public void testUsingWelcomeService() {
        ServiceReference sref = context.getServiceReference
        (WelcomeService.class.getName());
        WelcomeService service = (WelcomeService)
        context.getService(sref);
        Assert.assertEquals("Welcome,
        Harry!",service.welcome("Harry"));
}
```

This assumes that we have injected the bundle context into the test case as well. Using OSGi's standard APIs we are able to get references to our services in our test case to validate how they work, similar to how another OSGi may call our service.

Life outside of OSGi embedded

So far, we've looked at JBoss OSGi only in the embedded sense. We have only deployed to the internal container, not to a full JBoss AS 7.1 instance. While it's not currently possible to run test cases this way, this is an expected feature in JBoss AS 7.2. The next few sections will focus on some of the capabilities you can gain once those features are available.

Testing Java EE using OSGi

[Once we are working with a full container, it's actually quite easy to do things such as test standard Java EE services. In order to run the test suite, you must have downloaded and built JBoss AS 7.2. Set `JBOSS_HOME` to this path. In my case it was `/apps/src/jboss-as/build/target/jboss-as-7.2.0.Alpha1-SNAPSHOT/`.]

If we start with the same `WelcomeService` example, first we redefine our deployment method as follows:

```
@Deployment
public static JavaArchive createDeployment() {
  return ShrinkWrap.create(JavaArchive.class, "mytest.jar")
          .addClasses(WelcomeServiceActivator.
           class,WelcomeService.class,WelcomeServiceImpl.
           class,CDIServiceListener.class)
          .setManifest(OSGiManifestBuilder.newInstance()
                  .addBundleSymbolicName("mytest.jar")
                  .addBundleManifestVersion(2)
                  .addBundleActivator
                  (WelcomeServiceActivator.class)
                  .addImportPackages(PackageAdmin.
                  class, BundleContext.class,
                      ServiceTracker.class, ManagedBean.class,
                      ManagementClient.class));
}
```

Some new things came up in this deployment. First is an activator, for starting up the bundle.

```
@ArquillianResource
PackageAdmin packageAdmin;

@ArquillianResource
ManagementClient managementClient;

@ArquillianResource
BundleContext context;
```

There are some additional enrichments that we can access. `BundleContext` is available for looking up services, and `PackageAdmin` is available to get bundles and start or stop them. There are other injection points available, but these are the important ones.

Finally, we have our test case, which is pretty similar to other existing test cases:

```
@Test
public void testNothing() throws BundleException {
  ServiceReference sref = context.getServiceReference(WelcomeService.
  class.getName());
  WelcomeService service = (WelcomeService)
  context.getService(sref);
  String w = service.welcome("Larry");
  Assert.assertEquals("Welcome, Larry!",w);
}
```

In the last of our basic examples, we can also show off how to create an EJB test case where the EJB delegates work to the OSGi container service. First, our deployment method:

```
@Deployment
public static JavaArchive createDeployment() {
  return ShrinkWrap.create(JavaArchive.class, "mytest.jar")
          .addClasses(WelcomeService.
          class, WelcomeServiceImpl.class,
          WelcomeServiceActivator.class, WelcomeProxyBean.class)
          .addAsManifestResource(EmptyAsset.INSTANCE,"beans.xml")
          .setManifest(OSGiManifestBuilder.newInstance()
                  .addBundleSymbolicName("mytest.jar")
                  .addBundleManifestVersion(2)
                  .addBundleActivator
                  (WelcomeServiceActivator.class)
                  .addImportPackages(BundleActivator
                  .class, Logger.class, Module.class,
                  InitialContext.class)
        .addExportPackages(WelcomeProxyBean.class));
}
```

The structure is very similar to the last deployment. We do include an EJB which simply does what our old test case does.

```
@Stateless
@LocalBean
public class WelcomeProxyBean {
  @Resource
```

```
    BundleContext context;

    public String greet(String name) {
      ServiceReference sref = context.getServiceReference
     (WelcomeService.class.getName());
          WelcomeService service = (WelcomeService) context.
          getService(sref);
          return service.welcome(name);
    }
  }
```

This EJB simply uses an injected `BundleContext` to load the service and run the greeter. The injection and test case then look as follows:

```
  @EJB
  WelcomeProxyBean proxy;

  @Test
  public void testViaProxyEJB() {
    String name = "Larry";
    Assert.assertEquals("Welcome, Larry!",proxy.greet(name));
  }
```

So as you can see, the code is not much different and we don't have any new dependencies as a result.

Multiple deployment scenarios

In our final case, we'll look at creating multiple implementations of a service and how to look them up, as well as the packaging nuances of working with this structure.

```
  public interface StringProvider {
    public String getString();
  }

  public class StringOne implements StringProvider {
    public String getString() {
      return getClass().getCanonicalName();
    }
  }

  public class StringTwo implements StringProvider {
    public String getString() {
      return getClass().getSimpleName();
    }
  }
```

First we have our service and the implementations for it. Pretty straightforward, we have a single `getString` method and two implementations, one returning the canonical class name (also known as the **FQCN**, **Fully-Qualified Class Name**) and the other returning the class's simple name. We'll want to make both available via rest calls as options.

Next, we have two activators. Each implementation will end up in its own OSGi bundle, so to handle that we need to have unique activators per class. They do essentially the same thing, but one per implementation that we are using.

```
public class StringOneActivator implements BundleActivator {
  private ServiceRegistration registration;
  public void start(BundleContext context) throws Exception {
    context.registerService(StringProvider.class.getName(),
        new StringOne(), null);
  }
  public void stop(BundleContext context) throws Exception {
    registration.unregister();
  }
}

public class StringTwoActivator implements BundleActivator{
  private ServiceRegistration registration;
  public void start(BundleContext context) throws Exception {
    context.registerService(StringProvider.class.getName(),
        new StringTwo(), null);
  }
  public void stop(BundleContext context) throws Exception {
    registration.unregister();
  }
}
```

Finally, we'll look at the UI to see how we can expose these services easily. First we have a JAX-RS application, to turn on JAX-RS support in our deployment:

```
@ApplicationPath("/rest")
public class RestApplication extends Application { }
```

Then we have our JAX-RS resource. It's going to simply return a string representing the various `getString` methods provided. It's dynamic enough so that if we add more implementations, their feedback will be used as well.

```
@Path("/resource")
public class RestResource {
  @GET
  @Produces("text/plain")
  public String listServices() throws InvalidSyntaxException {
    BundleContext ctx = getBundleContextFromClass();
    ServiceReference[] refs= ctx.getServiceReferences
    (StringProvider.class.getCanonicalName(), null);
        StringBuilder sb = new StringBuilder();
        List<String> strings = new ArrayList<String>();
        for(ServiceReference sr : refs) {
          StringProvider sp = (StringProvider)ctx.getService(sr);
          strings.add(sp.getString());
        }
        Collections.sort(strings);
        for(String s : strings) {
          sb.append(s).append("\n");
        }
        return sb.toString();
  }

  private BundleContext getBundleContextFromClass() {
        BundleReference bref = (BundleReference)
        getClass().getClassLoader();
        Bundle bundle = bref.getBundle();
        if (bundle.getState() != Bundle.ACTIVE) {
            try {
                bundle.start();
            } catch (BundleException ex) {
                throw new RuntimeException(ex);
            }
        }
        return bundle.getBundleContext();
    }
}
```

The REST resource is pretty straightforward. You'll notice that we have to manually load `BundleContext`; this is because injection of `BundleContext` does not work in JAX-RS resources (we previously injected it using `@Inject` in a test method and `@Resource` in an EJB).

The produce method is pretty straightforward: we get a reference to the bundle context, find all `ServiceReferences` to our `StringProvider`, and build a String that contains the results in a sorted fashion. Now that our UI is done we can build a simple test case for this setup. We'll use Arquillian but run it in client mode so that we can make direct HTTP requests. Before looking at the deployment structure (what I consider the more interesting piece), let's take a look at the test method. You'll notice some limitations in it.

```
@ArquillianResource
private URL baseUrl;

@Test
@RunAsClient
public void testGetResource() throws Exception {
  URL url = new URL(baseUrl,"/war-test/rest/resource");
  InputStream is = url.openStream();
  StringBuilder sb = new StringBuilder();
  BufferedReader br = new BufferedReader(new
  InputStreamReader(is));
  String line = null;
  while((line = br.readLine()) != null) {
    sb.append(line).append("\n");
  }
  Assert.assertEquals(StringTwo.
  class.getSimpleName()+"\n"+StringOne.class.
  getCanonicalName()+"\n",sb.toString());
}
```

Similar to other test cases, we use an injected `ArquillianResource` to get the base URL and then build the expected path from there. That's pretty clean and easy. Then we just need to read the URL contents, something we've done before.

Finally, we have the deployment structure. To deploy this we need to use an EAR with multiple modules. One module is the API tier, what is publically exposed, two of the JARs are the implementations for the services, and then finally we have the WAR file that will contain the UI.

```
@Deployment(testable=false)
public static EnterpriseArchive createDeployment() {
  JavaArchive api = ShrinkWrap.create(JavaArchive.class,"api.jar")
      .addClass(StringProvider.class)
      .setManifest(OSGiManifestBuilder.newInstance()
                  .addBundleSymbolicName("api.jar")
                  .addBundleManifestVersion(2)
                  .addExportPackages(StringProvider.class));
  JavaArchive one = ShrinkWrap.create(JavaArchive.class,"one.jar")
      .addClasses(StringOne.class,StringOneActivator.class)
      .setManifest(OSGiManifestBuilder.newInstance()
                  .addBundleSymbolicName("one.jar")
                  .addBundleManifestVersion(2)
                  .addBundleActivator(StringOneActivator.class)
                  .addExportPackages(StringOne.class)
                  .addImportPackages(BundleActivator.
                  class,StringProvider.class));
  JavaArchive two = ShrinkWrap.create(JavaArchive.class,"two.jar")
      .addClasses(StringTwo.class,StringTwoActivator.class)
      .setManifest(OSGiManifestBuilder.newInstance()
                  .addBundleSymbolicName("two.jar")
                  .addBundleManifestVersion(2)
                  .addBundleActivator(StringTwoActivator.class)
                  .addExportPackages(StringTwo.class)
                  .addImportPackages(BundleActivator.
                  class,StringProvider.class));

  WebArchive wa = ShrinkWrap.create
  (WebArchive.class, "war-test.war")
          .addClasses(RestApplication.class,RestResource.class)
          .setManifest(OSGiManifestBuilder.newInstance()
                  .addBundleSymbolicName("war-test.war")
                  .addBundleManifestVersion(2)
                  .addImportPackages(BundleActivator.
                  class,StringProvider.class,ServiceTracker.
                  class,Application.class)
                  .addBundleClasspath("WEB-INF/classes"));
  EnterpriseArchive ea = ShrinkWrap.create
  (EnterpriseArchive.class,"test.ear")
      .addAsModule(api).addAsModule(one).
      addAsModule(two).addAsModule(wa);
  return ea;
}
```

The final deployment becomes easy to build once we have the modules created. Each module is made up of one or two classes plus a manifest file with the OSGi header information embedded. We even have to add the web archive as an OSGi bundle so that we can enable JAX-RS support and the references to the other bundles as OSGi dependencies; otherwise, the services will not be found on a REST call.

OSGi support in GlassFish

GlassFish supports OSGi as well. Unlike the JBoss OSGi support there is no distinct container for GlassFish to work with OSGi. You would simply point to the appropriate container you want to work with; for example, your managed GlassFish or embedded GlassFish instance should work fine. If you wanted to create your bundle, your test setup may look something like the following:

```
@Deployment(testable=false)
public static JavaArchive createDeployment() {
  return ShrinkWrap.create(JavaArchive.class, "mytest.jar")
          .addClasses(Installer.class)
          .setManifest(createManifest());
}

private static StringAsset createManifest() {
  StringBuilder sb = new StringBuilder();
  sb.append("Manifest-Version: 1.0").append("\n");
  sb.append("Export-Package:
  com.tad.arquillian.chapter9.gf.test").append("\n");
  sb.append("Bundle-Version: 1.0.0.SNAPSHOT").append("\n");
  sb.append("Build-Jdk: 1.6.0_17").append("\n");
  sb.append("Built-By: manual").append("\n");
  sb.append("Tool: Bnd-0.0.357").append("\n");
  sb.append("Bnd-LastModified: ")
  .append(System.currentTimeMillis()).append("\n");
  sb.append("Bundle-Name: MavenHelloServiceApi OSGi
  Bundle").append("\n");
  sb.append("Bundle-ManifestVersion: 2").append("\n");
  sb.append("Created-By: Manual").append("\n");
  sb.append("Bundle-SymbolicName:
  com.tad.arquillian.chapter9").append("\n");
  sb.append("Bundle-Activator: ")
  .append(Installer.class.getCanonicalName()).append("\n");
  return new StringAsset(sb.toString());
}
```

This deployment assumes you're using Bundle-Activator to launch your bundle. If you're using some other means to create your bundles during deployment, you would simply need to recreate that process in ShrinkWrap to develop the archive.

OSGi support in GlassFish is a little bit more mature than in JBoss. As a result, you can use your CDI components more fluidly, especially in your test cases (they should be deployable as first class members as long as you have `beans.xml` deployed). (Better support for OSGi is slated with Weld 1.2.)

Summary

Arquillian support for OSGi is still in its infant stages. As support for the standard grows within JBoss, I believe its usage will become more stable and more robust. We can go ahead and develop and deploy usable OSGi bundles, but until there is strong integration to the Java EE platform it may not be an overly useful tool for your arsenal.

When it comes to most other application servers, we can actually use their default container adapters to do work. This logically makes more sense; however, we lose some features such as bundle or bundle-context injection. We could manually lookup the service if we had access to a managed object to provide the bundle or bundle context.

One of the benefits of using GlassFish for your OSGi runtime in your application server is that your application likely has little to change to work with it. Obviously, you lose some injection capabilities for bundle resources, but that may not be a needed capability for your testing. The main use case right now for JBoss OSGi is around testing the actual OSGi runtime, not around application testing.

In the final chapter of this book, we look indepth at using ShrinkWrap to build archives for use in your test deployments. We start to look at creating and manipulating descriptors in archives that can be used to tweak your test time deployments without needing to rebuild your entire archive with the change. You will be introduced to some recipes for importing prebuilt archives and we touch again on bringing libraries into your archive using the resolver APIs for ShrinkWrap.

10
ShrinkWrap in Action

ShrinkWrap is a powerful tool for building and manipulating Java archives. In fact, it is used internally to JBoss AS 7 to read archives. The goal of ShrinkWrap is to be able to create archives on the fly based on class and file structure. If you're familiar with ant and how it creates archives, ShrinkWrap is very similar except that it works based on classes declared in the JVM. This sets it apart greatly from other tools.

What is ShrinkWrap?

ShrinkWrap is a **domain specific language** (**DSL**) for accessing, creating, and manipulating archives. It is instantiated using a static call to the `ShrinkWrap` class, and takes a class as an argument to know what kind of archive we're working with. It can start as simple as `ShrinkWrap.create(JavaArchive.class)` to make a new `Archive`. You can also import a file easily, using `ShrinkWrap.createFromZipFile(JavaArchive.class,new File("myfile.jar"))`.

ShrinkWrap is meant to be easy to use, simplistic, and extensible. When you create an archive it will be deployed to your container. While Arquillian supports manual creation and deployment, it is much easier to allow Arquillian to deploy your archive for you. ShrinkWrap will make this process as simple or complex as you need, allowing you to bring in external dependencies, build complex archive structures, or even just importing a file from the file system; it's your choice.

Creating archives

Primarily you're going to work with three main archive types. `JavaArchives` are used for plain JAR files, `WebArchives` are used for WAR files, and `EnterpriseArchives` are used for EAR files. These are typically what you can deploy to an application server, but you may also want to create a `ResourceAdapterArchive` if you are developing that type of component. These are all accessed when you do `ShrinkWrap.create(Archive.class)`, where `Archive.class` would be one of those listed previously.

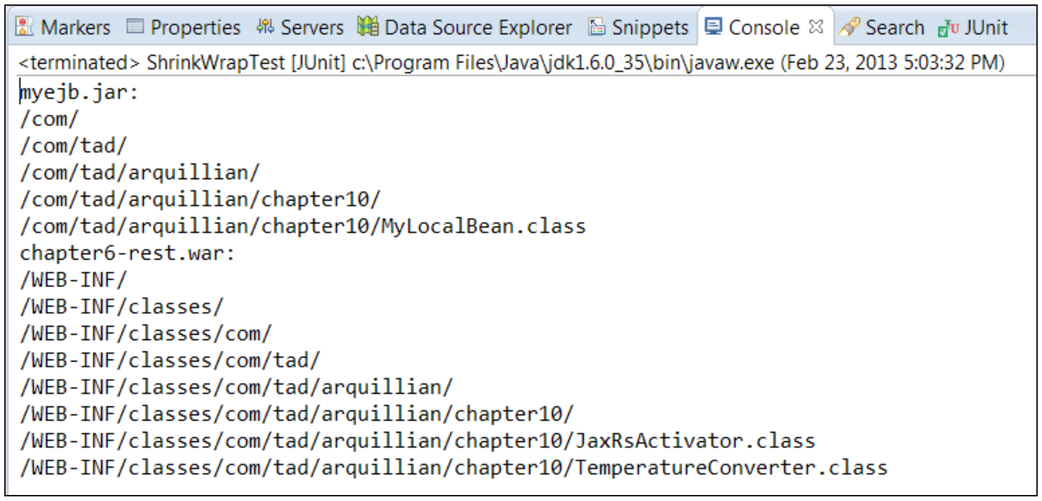
So the subject of this book is Arquillian. ShrinkWrap's first and foremost goal is generation of deployments. Arquillian leverages this completely to build its archives. Throughout the book we've had deployment methods, most of them looking something like this:

```
@Deployment
public static Archive<?> createEJBJar() {
  return ShrinkWrap.create(JavaArchive.class, "myejb.jar")
      .addClasses(MyLocalBean.class);
}
```

This is a very simple JAR file. I know it's a JAR file since I told ShrinkWrap to create a `JavaArchive` and the name is `myejb.jar`. After the `create` method is called, we can add classes or other types of assets to the archive. Assets are how ShrinkWrap generally refers to a construct that may exist in an archive. Similarly, we've had this type of deployment already shown:

```
@Deployment
public static Archive<?> createArchiveWithResolver() {
  return ShrinkWrap.create(WebArchive.class, "chapter6-rest.war")
      .addClasses(JaxRsActivator.class, TemperatureConverter.class);
}
```

We've spoken a little bit about what this does – but here's a little refresher. First, we create a WAR archive named `chapter6-rest.war`. We add two classes to the archive—`JaxRsActivator` and `TemperatureConverter`. Since we're working on a `WebArchive` class it knows exactly where to place the class files to match the type. To troubleshoot problems, we can output the contents of our archives using `System.out.println(archive.toString(true))` (or a logger). We can then see the differences in how these files are structured.



```
<terminated> ShrinkWrapTest [JUnit] c:\Program Files\Java\jdk1.6.0_35\bin\javaw.exe (Feb 23, 2013 5:03:32 PM)
myejb.jar:
/com/
/com/tad/
/com/tad/arquillian/
/com/tad/arquillian/chapter10/
/com/tad/arquillian/chapter10/MyLocalBean.class
chapter6-rest.war:
/WEB-INF/
/WEB-INF/classes/
/WEB-INF/classes/com/
/WEB-INF/classes/com/tad/
/WEB-INF/classes/com/tad/arquillian/
/WEB-INF/classes/com/tad/arquillian/chapter10/
/WEB-INF/classes/com/tad/arquillian/chapter10/JaxRsActivator.class
/WEB-INF/classes/com/tad/arquillian/chapter10/TemperatureConverter.class
```

As we can see, when we add classes to the `WebArchive` it knows to put them under `WEB-INF/classes` and when creating a `JavaArchive` they go in the root. This is simple so far, right?

There's also a shortcut we can use: instead of adding classes directly, we can add a `package` to our archive. Adding packages supports both `String` and `Package` objects; we can handle a package and all child packages from there as well.

Adding resources

Adding resources to archives—such as a `META-INF` entry or `WEB-INF` entry—will differ for each archive type. This is meant to align with how each would need to be positioned in the file. In Maven, for example, any files under `src/main/resources` end up as uncompiled code in `WEB-INF/classes`, as long as you're using a WAR file. Likewise, JAR files will put these files under the root directory. This is how the class loader expects to find files within your application server. The archive's `addAsResource` method is the right one to start using here.

Assets

Assets, as mentioned earlier represent different types of files that can be added. Your class files are assets (internally these are referenced as `ClassAssets`). There are five main asset types that you will use in your calls. There's no reason to use a `ClassAsset` directly; you can add classes more fluently. There are other asset types as well, but you would primarily use these assets when building your typical archives:

- `ByteArrayAsset`: This type is useful for any binary files that you may have read as a byte array.
- `FileAsset`: This type is most implicit of assets. Adding files is typically done via String calls; it assumes that a local path is given to find the file and will try to convert it automatically.
- `StringAsset`: This type is useful if you have simple String contents that are going into a file—perhaps a `ServiceProvider` implementation or a short properties file?
- `UrlAsset`: You can also refer to URL assets, contents as identified by a URL (external web page that has contents to be loaded).
- `EmptyAsset`: This type represents an empty file. This is the equivalent of using an empty `StringAsset`, but makes more sense from a fluent standpoint.

As you can see, the main use for an asset is for a file that is to be added, not a class, to an archive. Assets are also easy to work with. The `Asset` interface only defines one method, `openStream`, which returns an `InputStream` for the data. If you wanted to you could add additional asset implementations of your own for your use, but you probably wouldn't need to. In general, they all come with constructors to handle the type they wrap.

Assets get added directly or indirectly to an archive. When we add `ja.addAsResource("myfile.txt")` to the test case, we'll see that the file is loaded as `/myfile.txt`; however, adding it to the `WebArchive` puts it in `/WEB-INF/classes wa.addAsResource("myfile.txt")`. This is the same as if we added it as a `FileAsset`. The difference is that when we specify it as an asset we need to provide a name or location to put the file. This can be done with the second argument to the method or by using a `NamedAsset`. This approach works with all of the asset-related methods—`add`, `addAsResource`, `addAsManifestResource` (places the file in `META-INF`), and many others. The methods are overloaded in the archive class to allow many ways to add these assets.

Managing libraries and modules

Usually, if you're building a web application, you'll have some kind of library that you use. Whether it's a large framework such as Spring or Hibernate, or something more refined such as PrimeFaces or Apache Commons Email, you'll need to bring in some set of JAR files to your web application.

One way to add these libraries to your application is to use the relative path. Take a look at this code snippet:

```
public File[] getWebInfLibs() {
  File dir = new File("target/my-app/WEB-INF/lib");
  return dir.listFiles();
}
```

Assuming that you are using Maven, this will give you all files that were in your exploded directory's WEB-INF/lib directory. These files do get copied in place early enough in the build that they should be available during testing. Although, there may be reasons you only want certain files. You can manually choose each file from here and only install the files you want; however, ShrinkWrap has a more fluent API that better matches to your Maven build style. It can actually consume your `pom.xml` and resolve dependencies for you as defined within your project.

More on building EARs

Assuming Maven is our build tool, it makes sense to use the Maven Resolver to look up dependencies that we build ourselves to place into our EAR or WAR files. Typically, our deployable WAR file is made up of the following entries:

- One EJB jar, `com.mycompany:foo-ejb`
- One Library jar, `com.mycompany:foo-lib`
- One WAR file, `com.mycompany:foo-ui`

Our EAR file is the last part built in this setup. We'll use the resolver three times to find the three files and only import those three files.

```
EnterpriseArchive ea = ShrinkWrap.create(EnterpriseArchive.class,"foo-
ear.ear");
PomEquippedResolveStage mavenResolver = Maven.resolver().offline().
loadPomFromFile("pom.xml");
JavaArchive ejbJar = mavenResolver.resolve("com.mycompany:foo-ejb").
withoutTransitivity().as(JavaArchive.class)[0];
WebArchive warFile = mavenResolver.resolve("com.mycompany:foo-ui").
withoutTransitivity().as(WebArchive.class)[0];
```

```
JavaArchive lib = mavenResolver.resolve("com.mycompany:foo-lib").
withoutTransitivity().as(JavaArchive.class)[0];
ea.addAsLibraries(lib);
ea.addAsModule(ejbJar);
ea.addAsModule(warFile);
```

One difference here with each resolver call is that we choose `withoutTransitivity`—this tells the resolver to not bring in the transitive dependencies in Maven; we only want the artifact we requested. Enterprise archives have special methods for adding libraries and modules to their structure, to match how an EAR should be defined. One special thing to remember is that you cannot add classes directly to an EAR file. If you want to add classes to your EAR, you will need to add them manually to the appropriate module in the EAR to test them out.

ShrinkWrap resolver

ShrinkWrap provides a resolver API for retrieving artifacts in a generic fashion. Currently the only implementation is Maven based, which leverages your local `pom.xml` to retrieve artifacts. There is a certain amount of configuration you can do as needed, but in general you're going to want to set up your resolver to look at your `pom.xml` (or you can point to another project XML file).

```
Maven.resolver().offline().loadPomFromFile("pom.xml")
```

This snippet tells ShrinkWrap to create a Maven Resolver, go offline, and load structure from the local `pom.xml`. We can take it a step further. Using this same setup, we can resolve an artifact and bring in its transitive dependencies just like this:

```
Maven.resolver().offline().loadPomFromFile("pom.xml").
resolve("commons-email:commons-email").withTransitivity().asFile();
```

This will give you a `File[]` representing the files that are needed for this dependency. In the `WebArchive` that you're building, you can then call `wa.addAsLibraries(libs)` to add these files to your archive. You can also resolve multiple dependencies; the resolve method accepts variable arguments to parse multiple dependencies at once. Once you've loaded your pom file above, you'll have access to a `PomEquippedResolveStage` object to perform additional look up with additional dependencies as needed.

You can also use the Maven Resolver to look up resources for a multi-module build, as long as it's used in the last steps of the build. Let's suppose you're going to create an EAR file and deploy that EAR file in your tests. You want to use some of your modules in their entirety, maybe add a test to one of the JAR files and then deploy your entire JAR file in a more integration-test type of setup.

Importing modules

Another place where ShrinkWrap has some easy utilities is when we want to import a module directly after a build to use as our deployment. This works assuming that you have a target directory with your deployable archive available directly. Since it goes against your already-built JAR file, this assumes that any unit tests have already passed and instead you are more likely doing functional or integration testing.

If you know the file's path, importing it as an archive is as simple as using `JavaArchive ja = ShrinkWrap.createFromZipFile(JavaArchive.class, new File("target/myfile.jar"))`, which will read the file and create a `JavaArchive` from it. This object can be modified, so if you want to change entries, add or remove contents you're free to. They will be applied to the given `JavaArchive`. Of course, this approach assumes that the target file is already created. Based on the standard Maven lifecycles, the final JAR file is not available until right before it's copied to your local repository. Of course you can change the lifecycle to create your JAR earlier, but we have to think whether this is the right way or not. If we are working with the entire archive, this is more of an integration test – we need to ensure that all parts of our deployment are working to be able to test this out. You probably wouldn't run this with the build, but perhaps in a post-build setup, so maybe:

```
mvn clean install –Punit-test && mvn test –Pintegration-test
```

To call these tests right after your build execution, you could set up your build to act like this, import your completed archive, and run the tests.

Deployment descriptors

Another useful feature in ShrinkWrap is its ability to create deployment descriptors. This is provided in the ShrinkWrap descriptors module, the ability to create and import descriptors, then export them for inclusion in an archive.

First, we'll tinker a bit with the `beans.xml` descriptor. Typically, this file is empty and is used purely as a marker to indicate that CDI should be enabled in this archive.

```
BeansDescriptor beans = Descriptors.create(BeansDescriptor.class);
StringAsset beansXml = new StringAsset(beans.exportAsString());
JavaArchive ja = ShrinkWrap.create(JavaArchive.class);
ja.addAsManifestResource(beansXml, "beans.xml");
```

`Descriptors` is the entry point to this DSL, similar to the `ShrinkWrap` entry point. This allows you to create a new descriptor instance of importing existing descriptor content.

```
Descriptors.importAs(BeansDescriptor.class).fromString(beansXml.
getSource());
```

Once we have the `Descriptor` object available we can begin manipulating the content of the descriptor. At this point, the content should look like this:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<beans 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/beans_1_0.
xsd"/>
```

This is the standard header of the file. We can add some more content to the file in a type-safe manner just like this:

```
BeansDescriptor beans = Descriptors.create(BeansDescriptor.class);
beans.getOrCreateInterceptors().clazz("com.mycompany.cdi.
MyInterceptor");
.. more calls
StringAsset beansXml = new StringAsset(beans.exportAsString());
```

You do need to ensure that you complete adding contents before you export the descriptor.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<beans 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/beans_1_0.
xsd">
  <interceptors>
    <class>com.mycompany.cdi.MyInterceptor</class>
  </interceptors>
</beans>
```

The resulting descriptor will correctly have the class listed as defined previously. Unfortunately, there is no class argument for the method but instead we can call `MyInterceptor.getClass().getCanonicalName()` to add this class directly.

Importing descriptors

It's also possible to import a descriptor and manipulate it before adding it to your archive. We'll start with this `persistence.xml`, which we'll change to match the use within a single test case.

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://java.sun.com/xml/ns/
             persistence http://java.sun.com/xml/ns/persistence/
             persistence_2_0.xsd"
```

```
                version="2.0">
     <persistence-unit name="ShrinkWrapUnit">
        <jta-data-source>java:/jdbc/MyDataSource</jta-data-source>
        <properties>
           <property name="hibernate.hbm2ddl.auto" value="none"/>
           <property name="hibernate.show_sql" value="true"/>
        </properties>
     </persistence-unit>
  </persistence>
```

There's one persistence unit and we'll change the property for `hibernate.hbm2ddl.auto` to be `create-drop` instead of none. To do this we need to iterate through all persistence units and all properties to find the right one.

```
PersistenceDescriptor pd = Descriptors.importAs(PersistenceDescriptor.
class).fromFile("src/test/resources/META-INF/persistence.xml");
for(PersistenceUnit<PersistenceDescriptor> unit :
pd.getAllPersistenceUnit()) {
  for(Property<Properties<PersistenceUnit<PersistenceDescriptor>>>
prop : unit.getOrCreateProperties().getAllProperty()) {
    if(prop.getName().equalsIgnoreCase("hibernate.hbm2ddl.auto")) {
      prop.value("create-drop");
    }
  }
}
```

As you can probably guess, the resulting output of the descriptor looks like this:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="2.0"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://
java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
<persistence-unit name="ShrinkWrapUnit">
   <jta-data-source>java:/jdbc/MyDataSource</jta-data-source>
    <properties>
      <property name="hibernate.hbm2ddl.auto" value="create-drop"/>
      <property name="hibernate.show_sql" value="true"/>
    </properties>
  </persistence-unit>
</persistence>
```

So now that our `persistence.xml` has been modified to be `create-drop`, we can add this descriptor to our `JavaArchive`.

```
jar.addAsManifestResource(new StringAsset(pd.exportAsString()),
"persistence.xml");
```

This is a pretty straightforward way to manipulate descriptors, right? While you can build the entire descriptor from scratch as well, typically you should have the base file to work from, and extend what's in there. If you do decide to make the descriptor entirely programmatically, you should ensure that it provides methods to be reused wherever possible.

Deploying descriptors

The other type of descriptor supported is a deployable descriptor. Arquillian allows you to deploy certain descriptors to certain application servers. This has to be a descriptor that you would typically deploy directly—so a data source for example would apply. You simply need to add it to your deployment chain, for example:

```
@Deployment(order=1)
public static Descriptor createDataSource(){
  return Descriptors.create(DataSourceDescriptor.class).
jndiName("java:/jdbc/MyDataSource").provider()...;
}
@Deployment(order=2)
public static Archive createApplication() {
  return ShrinkWrap.createFromZipFile(JavaArchive.class, new
File("target/myfile.jar"));
}
```

This will create the two deployments, first the data source and then the archive. This is the same order as the deployment will occur. You would need to deploy the data source first anyways, so that can be covered here.

A complex use case

Now that we've gone through all of the general features of ShrinkWrap, we're going to bring it all together for our final case, a complex deployment archive. The goals are as follows:

- Create and deploy a web archive
- It should contain two of the libraries that we use in our application, commons-email and commons-lang

- Add our own internal library that is made up of a single package
- Add two of our own web resources—jQuery and a single static HTML page with content

```
WebArchive war = ShrinkWrap.create(WebArchive.class,
    "ComplexExample.war");
```

We create our initial Archive—this is what we want to work with. Using the Maven Resolver we can find dependencies to declare in our `pom.xml`.

```
PomEquippedResolveStage mavenResolver = Maven.resolver().offline()
    .loadPomFromFile("pom.xml");
File[] libs = mavenResolver
    .resolve("org.apache.commons:commons-email",
        "commons-lang:commons-lang").withTransitivity()
    .asFile();
```

In addition, we can build a custom `JavaArchive`.

```
JavaArchive jar = ShrinkWrap.create(JavaArchive.class,"foobar.jar")
    .addPackage(FooBar.class.getPackage());
war.addAsLibraries(libs);
war.addAsLibraries(jar);
```

Both of those types of external dependencies can be added directly to our WAR file.

```
war.addAsWebResource("web/index.html");
war.addAsWebResource("web/jquery.js");
```

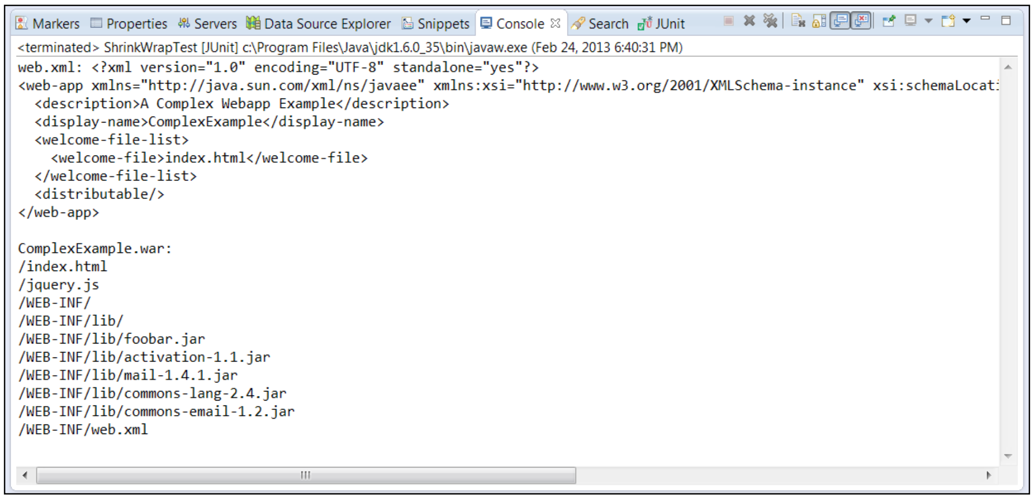
Next, we grab the two files sitting in `src/test/resources/web` and place them as `WebResources` in our WAR file. Since these are already on our classpath, we can just add them directly without needing a `FileAsset`.

```
WebAppDescriptor web = Descriptors.create(WebAppDescriptor.class)
    .description("A Complex Webapp Example")
    .displayName("ComplexExample").getOrCreateWelcomeFileList()
    .welcomeFile("index.html").up().distributable();
```

We also create the `WebAppDescriptor` which represents our `web.xml`. We set up the name, description, the welcome file, and mark that it's distributable. We also show off a little bit of navigation. The `WelcomeFileList` will be created dynamically but always has a reference back to its parent.

```
String webXml = web.exportAsString();
war.addAsWebInfResource(new StringAsset(webXml), "web.xml");
```

Finally, we export the descriptor and add it as the `web.xml`. Here's what the end result looks like:



```
<terminated> ShrinkWrapTest [JUnit] c:\Program Files\Java\jdk1.6.0_35\bin\javaw.exe (Feb 24, 2013 6:40:31 PM)
web.xml: <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocat
  <description>A Complex Webapp Example</description>
  <display-name>ComplexExample</display-name>
  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
  </welcome-file-list>
  <distributable/>
</web-app>

ComplexExample.war:
/index.html
/jquery.js
/WEB-INF/
/WEB-INF/lib/
/WEB-INF/lib/foobar.jar
/WEB-INF/lib/activation-1.1.jar
/WEB-INF/lib/mail-1.4.1.jar
/WEB-INF/lib/commons-lang-2.4.jar
/WEB-INF/lib/commons-email-1.2.jar
/WEB-INF/web.xml
```

The code used to create this is pretty clear and easy to read; what you would expect from a DSL type of framework.

Summary

As we've seen, ShrinkWrap has a lot of useful classes for creating and manipulating archives. It also has advanced features for managing your Maven dependencies to add to your test cases dynamically. ShrinkWrap is meant to be fluent. The lead developer of the project designs this with an API-first mentality—does this work well for a developer who may be using it? This is how the DSL approach for ShrinkWrap came to be.

Thank you for reading this book! Hopefully you got as much out of reading it as I did from writing it. My goal was to bring you top to bottom on an introductory level to Arquillian to start writing test cases that may deploy to a container.

Index

Symbols

@ArquillianResource 176
@ArquillianResource object 102
@Before method 127, 128
@ConversationScoped bean 102
@Deployment annotation 13
@Deployment method 42, 176
@EJB
  about **20, 96**
  and @Resource 96
@Inject 20, 95
@Inject Bundle bundle; 193
@Inject BundleContext context; 193
@Inject Framework framework; 193
@Inject method 122
@InSequence annotation 125
@RequestScoped method 43
@RequestScoped object 97
@Resource 19
@RunAsClient annotation 175
@RunWith annotation 13
@ShouldMatchDataSet annotation 132

A

activeByDefault 17
addPackage method 22
addPackages method 22
AfterServlet 161
Apache Tomcat. *See* **Tomcat**
Apache TomEE. *See* **TomEE**
application
  invocation 177
applicationContext.xml file 120
ApplicationScoped resource 99
archives
  creating 208, 209
Arquillian
  about 9, 25, 26
  and OSGi 191
  and soapUI 178-180
  BOM 15
  differences 10, 11
  extending 125
  extensions 26
  OSGi support 192
  Persistence Extension 129
  profiles, testing 16, 17
  test case, fundamentals 12, 13
  test cases, categorizing 17-19
  tests, enriching 19
Arquillian documentation
  URL 16
Arquillian extensions
  about 118
  framework extensions 118
  Spring 118-123
  versus Spring test 123
Arquillian Reference Guide
  URL 41
ArquillianResource annotation 20
Arquillian Weld EE 15
assets
  about 210
  ByteArrayAsset type 210
  EmptyAsset type 210
  FileAsset type 210
  StringAsset type 210
  UrlAsset type 210

B

BeforeServlet 161
Behavior-driven development (BDD) 145
BundleContext 201
ByteArrayAsset type 210

C

calculateSum method 12
CalculatorDelegate class 68
captures 172
CDI bean 67
CDI injection
  about 97-102
  extension, building 104-106
  external extensions, testing 102, 103
CloudBees 64, 74
cloud containers 73
cloud deployment 64
Configuration class 120
container
  about 41, 86, 87
  and deployment 41, 42
  comparing 70, 71
container, comparing
  Apache TomEE 71
  CloudBees 74
  cloud containers 73
  GlassFish 73
  JBoss 72
  JBoss AS 6 72
  JBoss AS 7 73
  OpenShift 74
  Pre-JBoss 6 72
  Tomcat 71
  WebLogic 74
container-specific descriptor 80
convertToCelsius method 30
create method 208

D

DAO 127
database
  cleaning up 132, 134
  testing against, JPA used 125-129
  validating 132, 134
DatabaseService CDI bean 112
dependencies
  deploying 141-145
dependency injection 95, 96
deployment 41, 42
descriptors
  deploying 216
  deployment descriptors 213, 214
  importing 214, 215
domain specific language. *See* **DSL**
DSL 207

E

EARs
  building 211, 212
  creating 137-140
EJB and resource injection
  about 106
  EJB deployments, considerations 107
  persistence, testing 107-111
  remote session beans 114, 115
  session beans, testing 112, 113
embedded application 11, 75
embedded container
  about 41, 44, **92**
  Apache Tomcat 49, 50
  Apache TomEE 52-54
  Arquillian Daemon Server 55
  Glassfish 3.1 54
  JBoss AS 6.x 55
  Jetty 48
  JPA/Hibernate 56
  mixing 50, 52
  OpenEJB 47
  OpenWebBeans 46
  running 44
  Spring 56
  summary 56
  Weld SE and EE 45, 46
EmptyAsset.INSTANCE 194
EmptyAsset type 210
Enterprise Archive (EAR files) 23
expect block 147, 148
extensions
  building 104
  testing 102, 103

F

FileAsset 194
FileAsset type 210
FQCN 200
Fully-Qualified Class Name. *See* **FQCN**
functional testing
  about 151
  page delegate 157, 158
  Selenium used 152-155
  test case, simplifying 156, 157
  Test Driven Development (TDD) 156
  Warp 158

G

GlassFish
  about 62, 73
  OSGi support 204
Glassfish 3.1 54, 59
GlasshFish 3.x 62
Graphene
  captures 172, 173
  rich application testing with 168-171
  selectors and interactions 173, 174

H

Hibernate 56

I

Infrastructure as a Service (IaaS) 64
Inspection 160
Instance object 116
int value 12
inversion of control. *See* **IoC**
IoC 12

J

J2EE 30
JaCoCo 140, 141
Java 2, Enterprise Edition. *See* **J2EE**
Java Archive (plain JAR files) 23
javadoc plugin 181
Java EE
  testing, OSGi used 197-199
JAXB 187
JAX-RS 99
JBoss 61, 72
JBoss AS 6 72
JBoss AS 6.x 55
JBoss AS 7 62, 73, 193
JBoss AS 7.1 58
JBoss containers
  about **58**
  AS 5.1 and AS 6 59
  JBoss AS 7.1 58
JBossJBoss AS 7 107
JBoss Modules
  versus OSGi 193
JBoss OSGi 192
Jetty 48
JMS
  testing 185, 187
JMX protocol 42
JPA 56
JUnit
  and Arquillian JUnit Container 15
JUnit deployment
  about 39
  managed (Boolean) attribute 39
  name (string) attribute 39
  order (int) attribute 39
  testable (true) attribute 39

L

libraries
  managing 211
local protocol 42

M

managed application server 11
managed (Boolean) attribute 39
managed container
  about 41, 57, **92**
  Glassfish 3.1 59
  JBoss containers 58, 59
  summary 60
  Tomcat 60
MANIFEST.MF 193
manual testing 30, 31

META-INF/context.xml file 51
mocking 33, 34
modules
  importing 213
  managing 211

N

name (string) attribute 39
NullPointerExceptions 118

O

OpenEJB 34, 47, 48
Open Services Gateway Initiative. *See* **OSGi**
OpenShift 74
OpenWebBeans 46
order (int) attribute 39
OSGi
  about 192
  and Arquillian 191
  JBoss OSGi 192
  service multiple implementations, creating 199-203
  support, in GlassFish 204
  used, for testing Java EE 197-199
  versus everything else 192
  versus JBoss Modules 193
  working, in test case 193
OSGi, in test case
  deployed service, testing 196
  OSGi Java archives 195
  ShrinkWrap Assets 194
OSGi Java archives 195

P

PackageAdmin 198
Page delegate pattern 157
permgen space 83
Persistence Extension 130, 131
phases 161
Platform as a Service (PaaS) 64
POJO 32
PomEquippedResolveStage object 212
Pre-JBoss 6 72
profiles
  overlapping 91
  setting up 91
protocols
  about 42
  JMX 42
  local 42
  servlet 2.5 42
  servlet 3.0 42

Q

QTP 37
QuickTest Professional. *See* **QTP**

R

remote application server 11
remote container
  about 41, 60, **92**
  Apache TomEE 63
  cloud deployments 64
  GlassFish 62
  GlasshFish 3.x 62
  JBoss 61
  JBoss AS 7 62
  Legacy JBoss 61
  summary 70
  Tomcat 61
  Tomcat 6 61
  TomEE 1.0/1.5 63
  WebLogic 62
  WebLogic 10g 63
  WebLogic 12c 63
remote session beans 114, 115
remote testing
  about 65
  non testable deployments 69
  other cases 68
  power 69
  structuring 66-68
  tools 68
  use cases 65
Resource Adapters (RARs) 23
resources
  adding, to archives 209

REST APIs
and soapUI 181, 183
REST tests
about 183
HTTP client testing 184, 185
RuntimeException 118

S

Seam 2
testing 188
Selenium
about **36**
used, for functional testing 152-155
service testing
application innovation 177
approaches 175
HTTP client testing 184, 185
JMS testing 185, 187
Seam 2 testing 188
soapUI, using 178
web service stubs 176
Servlet 2.5 protocol 42
Servlet 3.0 protocol 42
servlet containers 35
setup block 147
ShrinkWrap
about 15-207
archives, creating 208, 209
archive types 23
assets 210
resolver 212
resources, adding 209
ShrinkWrap API 22
ShrinkWrap Assets 194
soapUI
about 38
and Arquillian 178-183
and REST APIs 181, 183
using 178
spock 145, 146
Spring 36, 56, 118, 119
Spring MVC 163-168
Spring test
versus Arquillian 123
Stateful 145
StateProcessor class 145
StringAsset type 210
sum method 13

T

testable (true) attribute 39
Test Driven Development (TDD) 156
testing
progression 8
testing models, container
as client 69
in container 69
mixed mode 69
tests
archive types 79, 80
bugs 84
container 86, 87
container-specific descriptor 80-82
embedded container, dos and donts 91
managed container, dos and donts 92
package version, incorrect 82
packaging structure 78, 79
port, changing 84, 85
profiles, overlapping 91
profiles, setting up 90
remote container, dos and donts 92
remote server 88
resources, sharing 82, 83
running 77
runtimes, managing 89
start-up issue, handling 84
Tomcat
about 49, 60, 61, 71
Weld, activating 51
Tomcat 6 61, 78
TomEE 52, 53, 63
TomEE 1.0/1.5 63
toString method 25
Transaction Extension 135, 136

U

unittest profile 18
UrlAsset type 210

W

WADL 181
Warp
  about 158, 160
  and spring MVC 163-167
  and spring warp 168
  grouping requests 161, 163
  phases 161
Web Application Description Language. *See* **WADL**
Web Archive (WAR files) 23
WEB-INF/web.xml file 51
WebLogic 62, 74
WebLogic 10g 63
WebLogic 12c 63
Web service stubs 176
Weld
  activating, in Tomcat 51
Weld SE and EE 45, 46
when block 148

X

XDoclet 32