



P r o f e s s i o n a l E x p e r t i s e D i s t i l l e d

Learning Apache Karaf

Develop and deploy applications using the OSGi-based runtime container, Apache Karaf

Johan Edstrom
Heath Kesler

Jamie Goodyear

[PACKT] enterprise 
PUBLISHING professional expertise distilled

Learning Apache Karaf

Develop and deploy applications using the OSGi-based runtime container, Apache Karaf

Johan Edstrom

Jamie Goodyear

Heath Kesler



BIRMINGHAM - MUMBAI

Learning Apache Karaf

Copyright © 2013 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: October 2013

Production Reference: 1221013

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

ISBN 978-1-78217-204-8

www.packtpub.com

Cover Image by Gagandeep Sharma (er.gagansharma@gmail.com)

Credits

Authors

Johan Edstrom
Jamie Goodyear
Heath Kesler

Reviewers

Ladislav Gažo
Sachin Handiekar
Achim Nierbeck
Jean-Baptiste Onofré

Acquisition Editors

Pramila Balan
Sam Birch

Lead Technical Editor

Mohammed Fahad

Technical Editors

Dennis John
Gaurav Thingalaya

Copy Editors

Roshni Banerjee
Brandt D'Mello
Sayanee Mukherji
Lavina Pareira
Laxmi Subramanian

Project Coordinator

Amigya Khurana

Proofreader

Julie Jackson

Indexer

Priya Subramani

Graphics

Abhinash Sahu
Disha Haria

Production Coordinator

Aditi Gajjar

Cover Work

Aditi Gajjar

About the Authors

Johan Edstrom is an open source software evangelist, Apache developer, and seasoned architect; he has created Java architectures for large, scalable, high-transaction monitoring, financial, and open source systems.

Johan is, by training, an electronics engineer with a penchant for fractal geometry.

He has worked as development lead, infrastructure manager, IT lead, and programmer and has guided several large companies to success in the use of open source software components. Lately, he has been helping some of the world's largest networking companies and medical startups achieve high availability and scalability and dynamically adapt SOA systems.

Johan divides his time between writing software, mentoring development teams, and teaching people how to use Apache ServiceMix, Camel, CXF, and ActiveMQ effectively and scalable to enterprise installations.

He is the co-author of the book *Instant OSGi Starter*, Packt Publishing.

Johan is a committer on Apache Camel and Apache ServiceMix and is a PMC member for Apache Camel.

I'd like to thank my wife, Connee, my daughter, Annica, and my parents, Bengt and Birgitta, for supporting me, cheering us on while writing this book, and making it possible to work through quite a few nights.

I'd like to thank the Apache Software Foundation, a fantastic place fostering open source development.

Jamie Goodyear is an open source advocate, Apache developer, and computer systems analyst with Savoir Technologies; he has designed, critiqued, and supported architectures for large organizations worldwide.

Jamie holds a Bachelor of Science degree in Computer Science from Memorial University of Newfoundland.

Jamie has worked in systems administration, software quality assurance, and senior software developer roles for businesses ranging from small startups to international corporations. He has attained committer status on Apache Karaf, ServiceMix, and Felix and is a Project Management Committee member on Apache Karaf. His first print publication was *Instant OSGi Starter*, Packt Publishing, which he co-authored with *Johan Edstrom*.

Currently, he divides his time between providing high-level reviews of architectures, mentoring developers and administrators on SOA deployments, and helping to grow the Apache community.

I'd like to thank my family and friends for all of their support over the years. I'd also like to thank all the open source communities that have made Apache Karaf possible.

Heath Kesler is an Apache developer and committer and has spoken at conferences around the world. He is a Senior SOA Architect with Savoir Technologies and has architected and developed scalable, highly available SOA systems for large corporations around the globe.

Heath currently helps corporations implement and develop enterprise integration systems using messaging and web services with a focus on maintainability and scalability. He gives training classes on complex concepts and frameworks that provide functionality to large-scale enterprise solutions. He has bootstrapped development on mission-critical systems for several Fortune 500 companies.

Heath has reached committer status on Apache Karaf and has been a contributor to Camel. He received a Bachelor of Science degree from DeVry University after his tour in the army.

I'd like to thank my wife and kids for their unending support throughout my career. Thanks also to the open source communities for making high-powered software accessible to the masses.

About the Reviewers

Ladislav Gažo is a long-time computer enthusiast and has been digging into the software world since his very youth. He has professional experience with development and software engineering of more than 12 years. While starting experiments with computer graphics and network administration, he realized the true path is towards the combination of software engineering and business. He has been developing, analyzing, and architecting Java-based, desktop-based, and finally, modern web-based solutions for several years. Application of the agile approach and advanced technology is both his hobby and daily job.

Rich experience with various technologies led Ladislav to co-found Seges – a software development company in Slovakia. He actively participates in startup events and helps building development communities – Google Developer Group and Java Group – in Slovakia. With his colleagues, he designed and spun off an interactive content management solution called Synapso, utilizing contemporary technologies combined with user experience in mind.

I would not be able to realize my knowledge as part of the review process of this book without the support of all my colleagues, friends, and family. Creating a good, long-term environment helped me to gain the experience that I can pass on further.

Sachin Handiekar is a senior software developer with over 5 years of experience in Java EE development. He graduated in Computer Science at the University of Greenwich, London, and currently works for a global consulting company developing enterprise applications using various open source technologies such as Apache Camel, ServiceMix, ActiveMQ, and ZooKeeper.

He has a lot of interest in open source projects and contributed code to Apache Camel and developed plugins for Spring Social that can be found at Github (<https://github.com/sachin-handiekar>).

He also actively writes about enterprise application development on his blog at <http://sachinhandiekar.com>.

Achim Nierbeck has more than 14 years of experience in designing and implementing Java enterprise applications. He is a committer and PMC member for Apache Karaf and is the project lead of the OPS4j Pax Web projects. Since 2010, he has enjoyed working on OSGi enterprise applications.

While not working on projects or open source development, he enjoys spending time with his family and friends.

He can be reached at <http://notizblog.nierbeck.de>.

Jean-Baptiste Onofré is an Apache Software Foundation member. He's the PMC Chair for Apache Karaf. He's also a PMC member for Apache ACE, Apache ServiceMix, Apache Syncope, and Apache Kalumet.

He's a committer for Apache Camel and Apache Archiva.

Jean-Baptiste is a software architect at Talend, where he brings over 15 years of experience in software development, system integration, enterprise system environments, and network.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<http://PacktLib.PacktPub.com>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface	1
Chapter 1: Installing Apache Karaf	7
Prerequisites	7
Obtaining Apache Karaf distribution	9
Installing Apache Karaf	10
First boot!	11
Summary	12
Chapter 2: Commanding the Runtime	13
Command Review	13
Common commands	14
Remote console access	16
Apache Karaf client – a closer look	17
Custom command creation	17
Karaf-command-archetype	18
Karaf custom command project	18
JMX console	21
Optional web console	22
How to install and start the web console	22
Summary	23
Chapter 3: System Configuration and Tuning	25
Startup properties – remote access	25
Logging properties	26
File logging	27
Console logging	28
System properties	29
Configuring Karaf	29
Setting environment variables	31
Configuring hot deployment	31

Console configuration commands	31
Web console	33
Failover configuration	34
Startup properties	35
Summary	36
Chapter 4: Provisioning	37
Apache Maven repositories	37
The Karaf system repository	41
Apache Karaf features	42
Additional "features"	45
Summary	46
Chapter 5: Deploying Applications	47
Deploying bundles	47
Building a bundle	48
Deploying the bundle using Maven	49
Deploying a bundle using the file handler	50
Deploying a bundle using HTTP	50
Deploying a bundle using hot deployments	50
Deploying feature descriptors	52
Deploying non-OSGi JARs (wrap)	56
Deploying WAR	60
Deploying Spring/Blueprint	61
Creating and deploying a Karaf Archive	62
Summary	65
Chapter 6: Deploying Production-grade Apache Karaf	67
Offline repositories	67
How to build an offline repository	68
Improving application logging	70
High availability / failover	72
Installing Karaf as a service	72
Master-slave failover	73
Child instances	75
Basic security configuration	76
Managing roles	76
Password encryption	77
Locking down JMX access	78
Summary	78

Chapter 7: Apache Karaf Cellar	79
Getting started	79
Node discovery	80
Cluster groups	85
Cloud discovery	88
Summary	88
Chapter 8: Our Final Programming Project	89
Our application	89
A Maven build	90
Java and OSGi code	92
Apache Aries Blueprint	94
Extending Apache Karaf's command system	96
Deployment descriptors and features	97
Summary	99
Appendix: Apache Karaf Commands	101
Index	107

Preface

Welcome to *Learning Apache Karaf*. This book has been created especially to provide you with all the information that you'll need to get up and running with the Karaf runtime. You will learn the basics, get started with building your first applications for deployment, and discover some tips and tricks that help expose the versatility and power of the platform. Along the way, we'll also present the best practices that we have developed over years of working with Karaf.

Before we dive into exploring Karaf, let's answer one of the most often asked questions: why the name "Karaf"?

Apache Karaf originally started as a subproject of Apache ServiceMix, where it went by the name "Kernel." It provided a basic command-line interface to manage an OSGi container. Given time, the project matured until it was moved to the Apache Felix community. At this time the project was renamed "Karaf," based upon a reflection of the project's nature:

"A carafe is a small container used for serving wine and other drinks. In similarity to the name the Kernel allows applications to be more easily handled, and improves their characteristics (much like a bottle of wine left to breathe in a decanter)."

So why the respelling? It was felt that "Karaf" would be an easier search target, and it would fill the relatively empty "K" project name listings at Apache, should it ever go top level (which it did in mid 2010). Now that we have answered this most common question, let us explore Karaf.

What this book covers

Chapter 1, Installing Apache Karaf, provides a quick installation reference for users new to Apache Karaf.

Chapter 2, Commanding the Runtime, starts with an under-the-hood discussion of how Apache Karaf commands work. We then dive into a review of the most commonly used commands and ways to access them.

Chapter 3, System Configuration and Tuning, describes an Apache Karaf installation containing several default configuration options that most users will want to alter. This chapter dives into the contents of the Apache Karaf's `etc` folder.

Chapter 4, Provisioning, explains how the applications go about gathering all of the artifacts required for runtime, before we can begin deploying them into Apache Karaf. Apache Maven repositories and feature descriptors are introduced and explained.

Chapter 5, Deploying Applications, discusses how to deploy various application assemblies into Apache Karaf. We touch upon OSGi and non-OSGi JARs, feature descriptors, WARs, and other formats.

Chapter 6, Deploying Production-grade Apache Karaf, discusses production deployment concepts for Apache Karaf, including offline repositories, improving application logging, high availability / failover, and basic security configuration.

Chapter 7, Apache Karaf Cellar, forms our primer for Apache Karaf's clustering solution.

Chapter 8, Our Final Programming Project, helps us bring together our learned concepts to build a sample application following the best practices, now that we've studied Karaf.

Appendix, *Apache Karaf Commands*, provides a quick reference for the core Apache Karaf commands.

What you need for this book

The authors have taken great care to keep the quantity of software required to explore Apache Karaf to a minimum. You will need to obtain the following before trying out the samples included in this text:

- Apache Karaf 2.3.2 or newer distribution
- Oracle Java SDK 1.6 or newer

- Apache Maven 3.0
- Git Client
- Text editor

Who this book is for

Learning Apache Karaf is for developers and system administrators who need to discover and understand how to use Apache Karaf as an operating environment in the best possible way. Developers will learn the best practices for designing applications that fully integrate into the system, while administrators gain operational experience.

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 are shown as follows: "We can include other contexts through the use of the `include` directive."

A block of code is set as follows:

```
package com.your.organization;
import org.apache.felix.gogo.commands.Command;
import org.apache.karaf.shell.console.OsgiCommandSupport;

@Command(scope = "LAK", name = "commandname", description = "This
    is a sample custom command.")
public class CommandName extends OsgiCommandSupport {
    protected Object doExecute() throws Exception {
        System.out.println("Executing command commandname");
        return null;
    }
}
```

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


```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns=http://www.osgi.org/xmlns/blueprint/v1.0.0
    xmlns:cm="http://aries.apache.org/blueprint/xmlns/blueprint-
cm/v1.0.0">


<command-bundle xmlns="http://karaf.apache.org/xmlns/shell/v1.0.0">
    <command name="LAK/commandname">
        <action class="com.your.organization.CommandName">
        </action>
        <completers>
            <ref component-id="ourCommandCompleter" />
            <null />
        </completers>
    </command>
</command-bundle>
<bean id="ourCommandCompleter"
    class="com.your.organization.CommandCompleter" />
</blueprint>
```

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

```
@echo off
REM execute setup.bat to setup environment variables.
set JAVA_HOME=C:\Program Files\Java\jdk1.6.0_31
set MAVEN_HOME=c:\x1\apache-maven-3.0.4
set PATH=%JAVA_HOME%\bin;%MAVEN_HOME%\bin;%PATH%echo %PATH%
```

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: "clicking the **Next** button moves you to the next screen."

 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

Installing Apache Karaf

Before Apache Karaf can provide you with an **OSGi**-based container runtime, we'll have to set up our environment first. The process is quick, requiring a minimum of normal Java usage integration work.

In this chapter we'll review:

- The prerequisites for Apache Karaf
- Obtaining Apache Karaf
- Installing Apache Karaf and running it for the first time

Prerequisites

As a lightweight container, Apache Karaf has sparse system requirements. You will need to check that you have all of the below specifications met or exceeded:

- **Operating System:** Apache Karaf requires recent versions of Windows, AIX, Solaris, HP-UX, and various Linux distributions (RedHat, Suse, Ubuntu, and so on).
- **Disk space:** It requires at least 20 MB free disk space. You will require more free space as additional resources are provisioned into the container. As a rule of thumb, you should plan to allocate 100 to 1000 MB of disk space for logging, bundle cache, and repository.
- **Memory:** At least 128 MB memory is required; however, more than 2 GB is recommended.
- **Java Runtime Environment (JRE):** The runtime environments such as JRE 1.6 or JRE 1.7 are required. The location of the JRE should be made available via environment setting `JAVA_HOME`. At the time of writing, Java 1.6 is "end of life".



For our demos we'll use **Apache Maven 3.0.x** and **Java SDK 1.7.x**; these tools should be obtained for future use. However, they will not be necessary to operate the **base Karaf installation**. Before attempting to build demos, please set the `MAVEN_HOME` environment variable to point towards your Apache Maven distribution.

After verifying you have the above prerequisite hardware, operating system, JVM, and other software packages, you will have to set up your environment variables for `JAVA_HOME` and `MAVEN_HOME`. Both of these will be added to the system `PATH`.



Setting up `JAVA_HOME` Environment Variable

Apache Karaf honors the setting of `JAVA_HOME` in the system environment; if this is not set, it will pick up and use Java from `PATH`.

For users unfamiliar with setting environment variables, the following batch setup script will set up your windows environment:

```
@echo off
REM execute setup.bat to setup environment variables.
set JAVA_HOME=C:\Program Files\Java\jdk1.6.0_31
set MAVEN_HOME=c:\x1\apache-maven-3.0.4
set PATH=%JAVA_HOME%\bin;%MAVEN_HOME%\bin;%PATH%echo %PATH%
```

The script creates and sets the `JAVA_HOME` and `MAVEN_HOME` variables to point to their local installation directories, and then adds their values to the system `PATH`. The initial `echo off` directive reduces console output as the script executes; the final `echo` command prints the value of `PATH`.



Managing Windows System Environment Variables

Windows environment settings can be managed via the **Systems Properties** control panel. Access to these controls varies according to the Windows release.

Conversely, in a Unix-like environment, a script similar to the following one will set up your environment:

```
# execute setup.sh to setup environment variables.
JAVA_HOME=/path/to/jdk1.6.0_31
MAVEN_HOME=/path/to/apache-maven-3.0.4
PATH=$JAVA_HOME/bin:$MAVEN_HOME/bin:$PATH
```

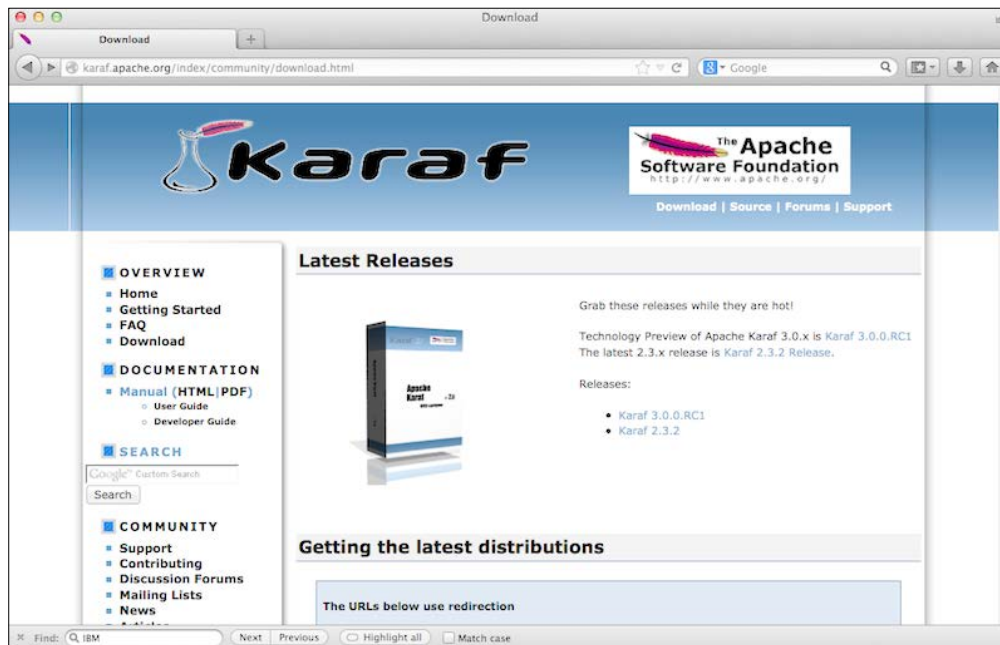
```
export PATH JAVA_HOME MAVEN_HOME
echo $PATH
```

The first two directives create and set the `JAVA_HOME` and `MAVEN_HOME` environment variables, respectively. These values are added to the `PATH` setting, and then made available to the environment via the `export` command.

Obtaining Apache Karaf distribution

As an Apache open source project, Apache Karaf is made available in both binary and source distributions. The binary distribution comes in a Linux-friendly, GNU-compressed archive and in Windows ZIP format. Your selection of distribution kit will affect which set of scripts are available in Karaf's bin folder. So, if you're using Windows, select the ZIP file; on Unix-like systems choose the `tar.gz` file.

Apache Karaf distributions may be obtained from <http://karaf.apache.org/index/community/download.html>. The following screenshot shows this link:



The primary download site for Apache Karaf provides a list of available mirror sites; it is advisable that you select a server nearer to your location for faster downloads.

For the purposes of this book, we will be focusing on Apache Karaf 2.3.x with notes upon the 3.0.x release series.

Apache Karaf 2.3.x versus 3.0.x series



The major difference between Apache Karaf 2.3 and 3.0 lines is the core OSGi specification supported. Karaf 2.3 utilizes OSGi rev4.3, while Karaf 3.0 uses rev5.0. Karaf 3 also introduces several command name changes. There are a multitude of other internal differences between the code bases, and wherever appropriate, we'll highlight those changes that impact users throughout this text.

Installing Apache Karaf

The installation of Apache Karaf only requires you to extract the `tar.gz` or `.zip` file in your desired target folder destination.

The following command is used in Windows:

```
unzip apache-karaf-<version>.zip
```

The following command is used in Unix:

```
tar -zxvf apache-karaf-<version>.tar.gz
```

```
icbts:LAK jgoodyear$ tar -zxvf apache-karaf-2.3.2.tar.gz
icbts:LAK jgoodyear$ cd apache-karaf-2.3.2
icbts:apache-karaf-2.3.2 jgoodyear$ ls -lah
total 1808
drwxr-xr-x@ 15 jgoodyear  wheel   510B Aug  7 20:05 .
drwxr-xr-x@  4 jgoodyear  wheel   136B Aug  7 20:05 ..
-rw-r--r--@  1 jgoodyear  wheel   27K Jul  7 10:31 LICENSE
-rw-r--r--@  1 jgoodyear  wheel    1.8K Jul  7 10:31 NOTICE
-rw-r--r--@  1 jgoodyear  wheel    3.8K Jul  7 10:31 README
-rw-r--r--@  1 jgoodyear  wheel   123K Jul  7 10:31 RELEASE-NOTES
drwxr-xr-x@  9 jgoodyear  wheel   306B Aug  7 20:05 bin
drwxr-xr-x@  3 jgoodyear  wheel   102B Jul  7 10:31 data
drwxr-xr-x@  8 jgoodyear  wheel   272B Jul  7 10:31 demos
drwxr-xr-x@  3 jgoodyear  wheel   102B Jul  7 10:31 deploy
drwxr-xr-x@ 21 jgoodyear  wheel   714B Jul  7 10:31 etc
-rw-r--r--@  1 jgoodyear  wheel   263K Jul  7 10:30 karaf-manual-2.3.2.html
-rw-r--r--@  1 jgoodyear  wheel   480K Jul  7 10:30 karaf-manual-2.3.2.pdf
drwxr-xr-x@ 10 jgoodyear  wheel   340B Aug  7 20:05 lib
drwxr-xr-x@  3 jgoodyear  wheel   102B Jul  7 10:30 system
icbts:apache-karaf-2.3.2 jgoodyear$
```

After extraction, the following folder structure will be present:

- The `LICENSE`, `NOTICE`, `README`, and `RELEASE-NOTES` files are plain text artifacts contained in each Karaf distribution. The `RELEASE-NOTES` files are of particular interest, as upon each major and minor release of Karaf, this file is updated with a list of changes.

- The `bin` folder contains the Karaf scripts for the **interactive shell** (Karaf), starting and stopping background Karaf service, a client for connecting to running Karaf instances, and additional utilities. These scripts will be described in later chapters.
- The `data` folder is home to Karaf's logfiles, bundle cache, and various other persistent data.
- The `demos` folder contains an assortment of sample projects for Karaf. It is advisable that new users explore these examples to gain familiarity with the system. For the purposes of this book we strived to create new sample projects to augment those existing in the distribution.
- The `instances` folder will be created when you use Karaf child instances. It stores the child instance folders and files.
- The `deploy` folder is monitored for hot deployment of artifacts into the running container. In *Chapter 5, Deploying Applications*, we will explore this feature in depth.
- The `etc` folder contains the base configuration files of Karaf; it is also monitored for dynamic configuration updates to the configuration admin service in the running container. Karaf configuration details will be explored in *Chapter 3, System Configuration and Tuning*.
- An HTML and PDF format copy of the Karaf manual is included in each kit.
- The `lib` folder contains the core libraries required for Karaf to boot upon a JVM.
- The `system` folder contains a simple repository of dependencies Karaf requires for operating at runtime. This repository has each library jar saved under a Maven-style directory structure, consisting of the library Maven group ID, artifact ID, version, artifact ID-version, any classifier, and extension.

First boot!

After extracting the Apache Karaf distribution kit and setting our environment variables, we are now ready to start up the container. The container can be started by invoking the Karaf script provided in the `bin` directory:

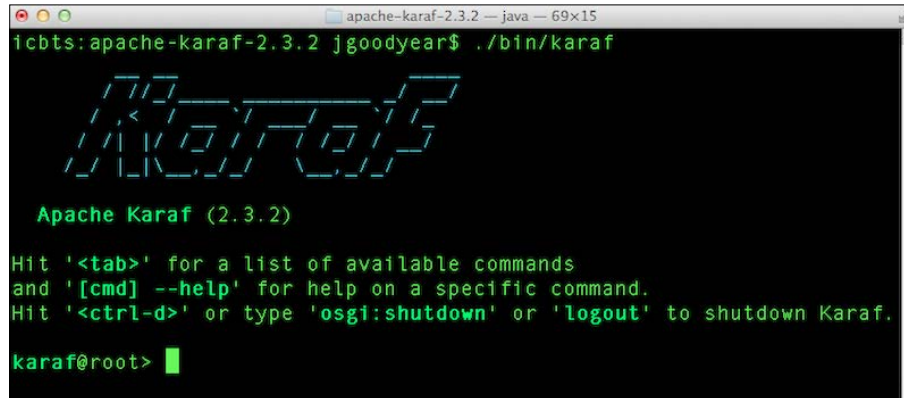
On Windows, use the following command:

```
bin\karaf.bat
```

On Unix, use the following command:

```
./bin/karaf
```


The following image shows the first boot screen:



Congratulations, you have successfully booted Apache Karaf! To stop the container, issue the following command in the console:

```
karaf@root> shutdown -f
```

The inclusion of the `-f` or `--force` flag to the shutdown command instructs Karaf to skip asking for confirmation of container shutdown.

[ Pressing **Ctrl + D** will shut down Karaf when you are on the shell; however, if you are connected remotely (using SSH), this action will just log off the SSH session, it won't shut down Karaf.]

Summary

We have discovered the prerequisites for installing Karaf, which distribution to obtain, how to install the container, and finally how to start it running.

In the next chapter we will introduce the Karaf command line, review some of the most commonly used commands, and ways to access them.

2

Commanding the Runtime

We have successfully installed and started Apache Karaf. We're presented with an interactive console; what do we do now? Have fun, of course! Apache Karaf provides a bevy of ways to interact with the container running on your local machine or remotely in datacenter.

In this chapter we'll review:

- Common Karaf commands
- Remotely connecting to Karaf
- Making your own custom commands
- Using Karaf's JMX management
- Karaf's optional web console

Command Review

When we first boot into Apache Karaf, we are presented with an interactive command shell that resembles a DOS or Unix prompt. This resemblance is intentional, and more importantly, it's harmonized to provide the same user experience regardless of the underlying host operating system.

Apache Karaf shell commands generally take the syntax `Scope:CommandName [options]`. The scope provides a namespace for a set of complementary commands to be gathered, and allows for disambiguation for common command names. The command name for a given scope must be unique. Each command may operate upon several optional inputs. For more information on a given command, run `--help` command to have it list its description, options, and syntax.

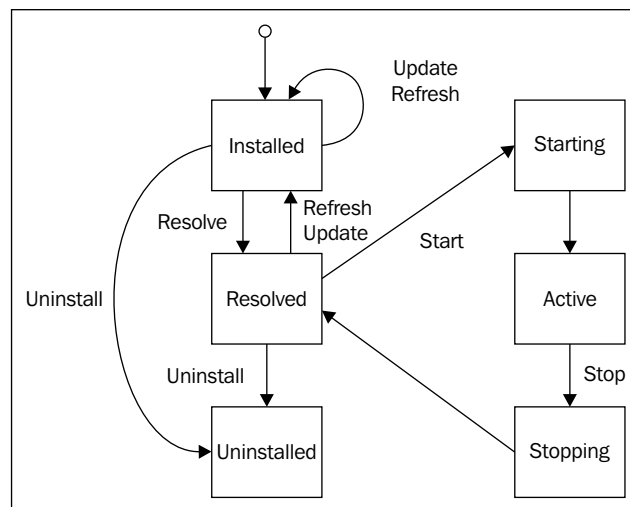


Discovering Commands

Pressing the *Tab* key will ask Karaf to complete the current command line entry similar to other well-known consoles (for example, bash).

Common commands

Apache Karaf roots, as an OSGi runtime container, permeates how applications are operated. All deployed applications follow the OSGi lifecycle. The following diagram represents the various bundle life cycle states:



Each state transition is initiated by invoking the command of the same name in accordance to the OSGi specification:

- **Install**: This installs one or more bundles into the container. Passing `-s` to the `install` command instructs the container to attempt starting the bundle if possible. When a bundle is successfully installed, its bundle ID will be presented – this identifies the bundle in the running container.
- **Uninstall**: This uninstalls a bundle via **bundle ID (remove from container)**.
- **Start**: This starts a bundle. If the bundle is in the resolved state, the container will call the bundle's `start` method, otherwise the container will attempt to resolve the bundle then start if successful.
- **Stop**: This stops a bundle via bundle ID (calls `BundleActivator`'s `stop` method).

- **Resolve:** This instructs the container to attempt to wire all required dependencies to the bundle.
- **Refresh:** This refreshes a bundle's wiring; this causes the container to recalculate dependencies.
- **Update:** This updates the bundle as per its updated location; bundle is reinstalled and rewired.

Refresh versus update



The subtle difference between refreshing and updating a bundle can lead to unexpected results. Issuing the `update` command may change the version of the bundle you're working with, while `refresh` just performs a rewiring to dependencies.

- **List:** This displays bundles/artifacts deployed into the container and their status. Passing `-t 0` will display all the installed bundles instead of just those above the default list threshold of start level 50.



Use the `la` alias to quickly list all bundles.

- **Info:** This displays system information of the Karaf instance. Versions, JVM information, thread metrics, memory usage, and host system information are provided.
- **Headers:** This views the headers in the bundle's manifest file.
- **Imports:** This displays imported packages. If you specify a bundle ID, the imports of that bundle are displayed.
- **Exports:** This displays exported packages. If you specify a bundle ID, the exports of that bundle are displayed.
- **Start-Level:** This obtains or sets the start level of the container. By convention, levels range in Karaf from 0 to 100; in which at level 0 the OSGi framework is not launched, and at level 100 all bundles may start. In the Karaf core system, bundles have run levels below 50 and user bundles generally start at level 60.
- **Bundle-Level:** This obtains or sets the start level of a particular bundle. Bundles with start levels above the current run level cannot start.
- **Framework:** Select OSGi framework for the container, and optionally set OSGi core debugging. By default, Apache Karaf uses Apache Felix, however it is common to switch to Equinox.

- **Show-tree:** This displays bundle wiring as a tree structure. This command allows you to observe which bundles are providing services to your bundle.
- **Create-dump:** This creates a directory containing diagnostic information.
- **Dynamic-import:** This enables/disables dynamic import for a given bundle.
- **Watch:** This watches for an updated version of a bundle; upon a new bundle becoming available it'll update to that version.
- **Print-stack-traces:** This prints the full stack trace in the console when the execution of a command throws an exception.
- **Restart:** This restarts the container (reboot).
- **Grep, Cat, Tail, and Pipes:** The Karaf console provides several utilities that Linux users will be familiar with, including `grep` for regular expression matching, `cat` for printing file contents to screen, `tail` for streaming file content to the screen, and `pipes` (we use the `|` character to denote a pipe) that feed output from one command to the input of another.
- **Shutdown:** This stops the Karaf instance.

This list is, by no means, exhaustive; please see *Appendix, Apache Karaf Commands*, for more commands and greater details of their options and inputs.



Command names across versions?

In the process of updating from Karaf 2.x to 3.0, a lot of commands were renamed (for example, `osgi:list` became `bundle:list`). In addition, some of the more commonly used commands had aliases created so that less typing would be required. Some examples of these include `la` for listing bundles, and `ld` for display log.

Remote console access

Karaf's interactive shell provides a powerful interface to administer the container runtime; however, most users require their application server to operate on a remote host—this is where Karaf's built-in **SSH** daemon steps in. By default, when a Karaf instance is started, it will bind upon port 8101 a **SSHD** service, which can be connected to using the provided Karaf client utility (in the Karaf's `bin` folder)—its embedded SSH client, or a third-party SSH client. The default credentials are username `karaf`, password `karaf`. The default port can be edited in `etc/org.apache.karaf.shell.cfg`, and the credentials edited in `etc/users.properties`. Once connected to an instance, you may type `exit` to log out of the system.

You do not need to run the interactive command line console to operate Karaf; you may simply start Karaf using the following invocations to start a headless session:

On Windows: `#bin\start.bat`

On Linux: `#bin\start`

Connecting to Karaf, in this case, only requires using any one of the remote clients we discussed before. Stopping one of these instances can be accomplished via the provided stop scripts, or by issuing the `shutdown` command.

Apache Karaf client – a closer look

The Apache Karaf client is found in the `bin` folder of your installation. It has two intended use modes; if a command is passed to the client, it executes then exits, otherwise without a command it starts an interactive shell session. Let's look at the client's optional input, then an actual invocation:

```
bin# client [-a port] [-h hostname] [-u username] [-p password] [-r
retries] [-d retry-delay] [commands]
```

The optional inputs are mostly self-explanatory; however, let's examine them using an actual invocation of the client script:

```
bin# client -a 8101 -h localhost -u karaf -p karaf -r 3 -d 5 info
```

The above invocation sample specifies to connect to port (8101), hostname (localhost, you may also use IP format 127.0.0.1), username (karaf), password (karaf), number of connection retries to attempt (-r 3, three retries), delay between connection attempt in seconds (-d 5, five seconds), and finally an optional command to be issued.

Custom command creation

The set of commands that come prebuilt into Apache Karaf are meant for the basic administration and monitoring of the runtime container (see *Appendix, Apache Karaf Commands*, for more in-depth command coverage). Some users, however, need to embed additional commands to augment their own applications. Apache Karaf provides a convenient methodology for expanding its collection of commands using Apache Maven archetypes. An archetype can be thought of as a maven-based project skeleton generator, in this case, all the setup is created to allow you to write your own Karaf command.

Karaf-command-archetype

When you invoke Apache Maven with the `Karaf-command-archetype`, a maven-based project will be generated for building a custom Karaf command. Use the following invocation to start the process:

```
# mvn archetype:generate \
-DarchetypeGroupId=org.apache.karaf.archetypes \
-DarchetypeArtifactId=karaf-command-archetype \
-DarchetypeVersion=2.3.2 \
-DgroupId=com.your.organization \
-DartifactId=com.your.organization.command \
-Dversion=1.0.0-SNAPSHOT \
-Dpackage=com.your.organization
```

During project generation, you will be prompted to provide the `command name`, its `description`, and define in which `scope` it exists. Once the project is generated, you may then import it into your IDE for further development.

Karaf custom command project

Using the generated project as our base, let's review the project contents:

- `pom.xml`: POM stands for `project object model`. This file provides Maven with the directives to obtain required dependencies, compile code, assemble binaries, and blueprint descriptor into a bundle.
- `src/main/java/com/your/organization/CommandName.java`: The command logic extends `OsgiCommandSupport`. This is where your custom command logic is implemented.
- `src/test/java/com/your/organization/CommandNameTest.java`: This is a unit test skeleton class.
- `src/main/resources/OSGI-INF/blueprint/shell-log.xml`: A blueprint descriptor file that wires the custom command logic into the Karaf shell. Blueprint is very similar to wiring Spring XML, however it has been designed for use with OSGi.

Let's take a closer look at the custom command code:

```
package com.your.organization;
import org.apache.felix.gogo.commands.Command;
import org.apache.karaf.shell.console.OsgiCommandSupport;
```

```

@Command(scope = "LAK", name = "commandname", description = "This
is a sample custom command.")
public class CommandName extends OsgiCommandSupport {
    protected Object doExecute() throws Exception {
        System.out.println("Executing command commandname");
        return null;
    }
}

```



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.

Karaf's heritage from the Apache Felix Gogo shell becomes apparent as we utilize the Gogo Command and extend Karaf's `OsgiCommandSupport`. The command annotation sets our custom command's scope, command name, and its description (this description is returned when you run the `help` command). Internally to our `CommandName` logic, we implement the `doExecute()` method; any custom operations are carried out here. We return `null` to signal the end of our command.

Let's take a closer look at the blueprint wiring code:

```

<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns=http://www.osgi.org/xmlns/blueprint/v1.0.0
  http://www.osgi.org/xmlns/blueprint/v1.0.0 xmlns:cm="http://aries.
  apache.org/blueprint/xmlns/blueprint-cm/v1.0.0">

  <command-bundle xmlns="http://karaf.apache.org/xmlns/shell/v1.0.0">
    <command name="LAK/commandname">
      <action class="com.your.organization.CommandName">
      </action>
    </command>
  </command-bundle>
</blueprint>

```

The blueprint wiring for our command picks up the common namespaces for blueprint, configuration management, and Karaf's shell. Of particular interest to us is the command and action elements inside the `command-bundle` block. The `command` element sets a command scope and name, using a slash to denote the separation between scope and name. The `action` element allows us to wire the `CommandName` implementation class to the command name.

For Karaf's tab completion mechanism to work, we'll have to add a shell completer.

In the path `src/main/java/com/your/organization/`, create a file `CommandCompleter.java`, and include the following:

```
package com.your.organization;
import org.apache.karaf.shell.console.completer.StringsCompleter;
import org.apache.karaf.shell.console.Completer;
import java.util.List;

public class CommandCompleter implements Completer {
    public int complete(String buffer, int cursor, List
        candidates) {
        StringsCompleter delegate = new StringsCompleter();
        delegate.getStrings().add("option1");
        delegate.getStrings().add("option2");
        delegate.getStrings().add("etc");
        return delegate.complete(buffer, cursor, candidates);
    }
}
```

Our `CommandCompleter` class extends the Karaf's `Completer`, implementing the `complete` method. When we try to tab complete the command, our list of candidate completions will be returned.

To make this completer available to the Karaf shell, we must update our blueprint XML file as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns=http://www.osgi.org/xmlns/blueprint/v1.0.0
    xmlns:cm="http://aries.apache.org/blueprint/xmlns/blueprint-
cm/v1.0.0">

    <command-bundle xmlns="http://karaf.apache.org/xmlns/shell/v1.0.0">
        <command name="LAK/commandname">
            <action class="com.your.organization.CommandName">
            </action>
            <completers>
                <ref component-id="ourCommandCompleter" />
                <null />
            </completers>
        </command>
    </command-bundle>
    <bean id="ourCommandCompleter" class="com.your.organization.
CommandCompleter" />
</blueprint>
```

We add to our `command-bundle` block, the `completers` element, in which we specify a reference to our `completers` component implementation. We take care to add a bean reference to our implementation class outside the `command-bundle` scope. The `completer` element is our custom code for handling the case where a user attempts to tab complete one of our custom commands.

Building the project only requires invoking:

```
# mvn install
```

To deploy the project into your Karaf container, issue the following command:


```
karaf@root> install -s mvn:com.your.organization/com.your.organization.
command/1.0.0-SNAPSHOT
```

Your command will now be available on the Karaf shell; to test it out try the following command:

```
karaf@root> lak:commandname
Executing command commandname
karaf@root>
```

Experiment with using tab completion to see additional options to the command:

```
karaf@root> commandname <tab>
etc option1 option2
karaf@root> commandname
```

[ The LAK scope will be in lowercase]

JMX console

In addition to Karaf's interactive shell, administrative access to the container is also provided through the **Java Management Extension (JMX)** facility. Access is granted via port `1099` by default, however this can be reconfigured in `etc/org.apache.karaf.management.cfg`.

In general, users typically connect to Karaf's JMX port using `JConsole`. Once connected, you may access MBeans to monitor and administer the following types:

- **Admin:** This administrates child Karaf instances
- **Bundles:** This manipulates OSGi bundles

- **Config:** This manipulates Karaf configuration files (etc folder), which affects ConfigAdmin
- **Dev:** This retrieves information from and manipulates the OSGi framework
- **Diagnostic:** This creates an information file containing Karaf's activities (dump file)
- **Features:** This operates upon Karaf features
- **Log:** This manipulates the logging mechanism
- **Packages:** This manipulates PackageAdmin and retrieves information on imported and exported packages in the container
- **Services:** It retrieves information on OSGi services
- **System:** It accesses to shut down a Karaf instance
- **Web:** This retrieves information on web bundles (install the WAR feature first – a feature is a sort of script for instructing Karaf to install everything we require to run a particular application)
- **HTTP:** This retrieves information on an HTTP servlet
- **OBR:** This manipulates the OSGi bundle repository (optional feature must be installed first)

The sensitive nature of the capabilities offered by MBeans means that you should restrict access to the JMX port. More information on locking down Karaf will be presented in *Chapter 6, Deploying Production-grade Apache Karaf*.

Optional web console

Apache Karaf strives to remain a lightweight application container runtime, as such, Karaf's **web console** is available as an optional capability. The web console provides a graphical interface to system status, deployed artifacts, configuration, and a web-based shell for issuing commands. This web console provides a convenient interface when using a secure shell terminal isn't practical.

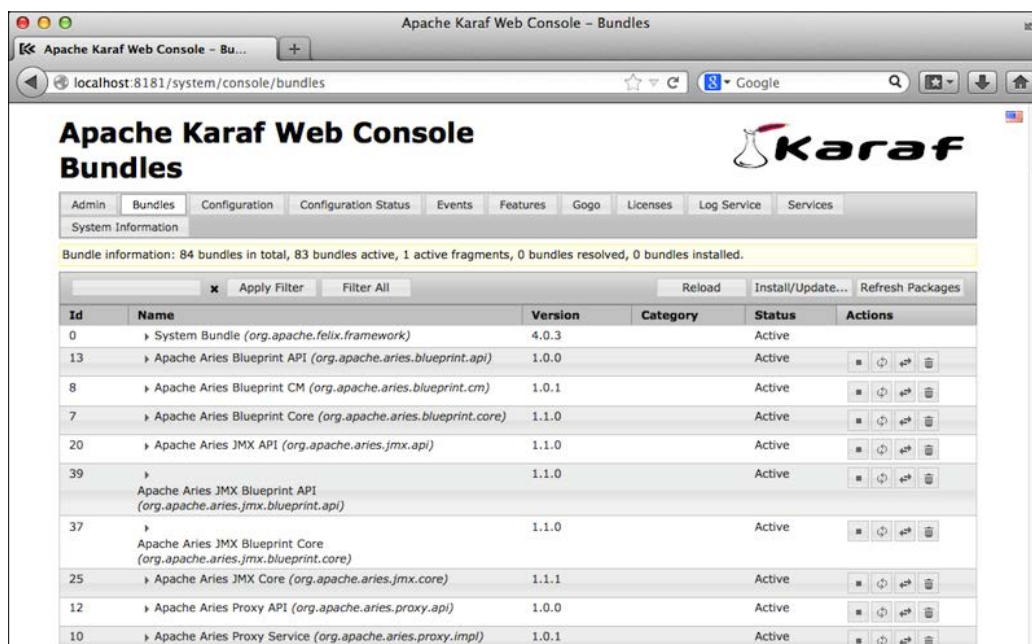
How to install and start the web console

The installation and starting of the web console requires an internet connection, and the issuing of two commands:

```
karaf@root> feature:install webconsole
```

The command instructs Karaf to install and start the web console (more on Karaf feature descriptors can be found in *Chapter 4, Provisioning*). All dependencies required to operate the web console will be obtained by Karaf. Once the web console is started, you can open your web browser to <http://localhost:8181/system/console>.

The following screenshot shows the opened screen:



Upon opening the above URL, you will be asked to provide credentials; use the default credentials: `karaf/karaf` (username/password). Once you've authenticated, you'll be presented the bundles view; you can change views by selecting the various tabs.

Summary

We've just begun diving into using Apache Karaf, becoming familiar with some of the most common commands you'll use, and how we can create our own custom commands if required. Finally, we explored the various ways in which Karaf makes itself accessible to users.

In our next chapter we'll dive into tuning your Apache Karaf system configuration.

3

System Configuration and Tuning

Getting Karaf installed and running is an exciting first step, but learning how to navigate the commands will make you dangerous. Now we need to get the system set up with a solid and stable runtime environment. For this we need to look into the configuration files. An Apache Karaf installation has several default configurations that most users will want to alter to meet their needs. We will dive into the content of the Karaf `etc` directory, learn how to change the default OSGi container/settings, and how to configure your system for remote access.

In this chapter we'll review the following:

- Remote access
- Logging properties
- System properties

Startup properties – remote access

The first thing to do is to look at the configuration file that houses the settings for SSH remote access. Most configuration files are held in the `etc` directory of the of the Karaf installation directory.

As mentioned in the previous chapter, `etc/org.apache.karaf.shell.cfg` can be used to change the defaults for remote access. The default configuration will look like the following:

```
sshPort=8101
sshHost=0.0.0.0
sshIdleTimeout=1800000
sshRealm=karaf
hostKey=${karaf.base}/etc/host.key
```


`sshHost` and `sshPort` defines the host name and the port to use when connecting to this system from a remote client.

`sshIdleTimeout` is used for defining the amount of time the client can remain idle before getting logged out (default is 30 minutes).

`sshRealm` is the JAAS domain used for password authentication.

`hostKey` is used to locate the private/public key location; this setting is ignored if no file exists.

The optional settings are used to give more control over authentication (defaults are as defined):

```
sshRole=admin
```

```
keySize=1024
```

```
algorithm=DSA
```

Use the `user.properties` file under `etc` to configure the login credentials for the Karaf instance. You should see the following in the file:

```
karaf=karaf,admin
```

This will use the `[user=password,role..roleN]` format with any roles comma-delimited after the password. The default definition has `karaf` as the username and `karaf` as the password with the `admin` role associated with it.

The `admin` role can be defined in `etc/system.properties` and the default will look like the following:

```
karaf.admin.role=admin
```

This will allow you to connect to the remote system using the following command as defined in *Chapter 2, Commanding the Runtime*:

```
%KARAF_HOME%/bin/client -a 8101 -h localhost -u karaf -p karaf -r 3 -d 5
```

Logging properties

The logging system for Karaf is based on OPS4J Pax Logging and the backend uses `log4j`, which allows for many of the additional features such as nested filters, appenders, and error handlers. The OPS4J Pax Logging system is a standard OSGi logging framework that supports API's and provides OSGi-friendly implementations of the following:

- Apache `log4j`
- Apache Commons Logging

- SLF4J
- Java Util Logging

File logging

The configuration file `etc/org.ops4j.pax.logging.cfg` is used to define appenders, log levels, and so on. Just take a quick look at one of the appenders that comes predefined in the logging configuration file:

```
# File appender
log4j.appender.out=org.apache.log4j.RollingFileAppender
log4j.appender.out.layout=org.apache.log4j.PatternLayout
log4j.appender.out.layout.ConversionPattern=%d{ISO8601} | %-5.5p |
%-16.16t | %-32.32c{1} | %-32.32C %4L | %X{bundle.id} - %X{bundle.
name} - %X{bundle.version} | %m%n
log4j.appender.out.file=${karaf.data}/log/karaf.log
log4j.appender.out.append=true
log4j.appender.out.maxFileSize=1MB
log4j.appender.out.maxBackupIndex=10
Configuration using the command line
```

The above appender is what Karaf uses to write to the `log/karaf.log` file. It uses an `org.apache.log4j.RollingFileAppender` to allow the system to roll off files that get filled up to the `log4j.appender.out.maxFileSize`.



It is recommended to increase your file size to avoid rolling the files too often and to avoid filling up the logs during debug level logging.

Changing the file location can be done by updating `log4j.appender.out.file` to a preferred file location.

In order to change the log level to get information, you will need to change the root-level logger in the configuration file as seen in the following line of code:

```
log4j.rootLogger=INFO, out
```

Change the `rootLogger` to the desired logging level (for example, `ERROR`, `WARN`, `INFO`, and `DEBUG`). By default, the Karaf logger is using the `out` appender defined previously.

Console logging

All this is great, but what if you want to change the log settings while the application is running? Well, you are in luck; the guys who built Karaf made it so that you can change the log settings as well as view the log from the console.

Let's start with viewing the log from the console. This is a functionality that is extremely helpful when debugging issues such as bundle startup failures.

Once you have the console up and running, view the log from the console using the following command:

```
karaf@root> log:display
```

The preceding command will present the last lines of the log. But in many cases, what you are looking for is an exception that is not always presented in the last few lines of the log

```
karaf@root> log:display-exception
```

The preceding command will present the last exception in the log. Both the `display` and `display-exception` commands can be configured from the `org.apache.karaf.log.cfg` file under `etc`. Change the number of lines displayed by default, or override that from the command line using the `-n` parameter. The log statement pattern can also be changed in this file or overridden from the command line using the `-p` parameter:

```
karaf@root> log:clear
```

The preceding command will clear the log; it is very useful when trying to debug specific bundle installation problems:

```
karaf@root> log:get
```

```
Level: INFO
```

Use the `log:get` command in order to get the current log level settings. If you decide this not the right setting, you have the ability to change it from the command line using the following:

```
karaf@root> log:set LEVEL [package]
```

Set the log level at the package level by adding the package at the end of the command as a parameter. So a full package-level logging command might look like the following:

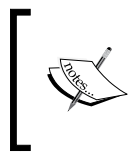
```
karaf@root> log:set DEBUG org.myproject.subproject
```

Also, logging per bundle can be activated by setting the following property in `org.ops4j.pax.logging.cfg`:

```
log4j.rootLogger=INFO, out, sift, osgi:VmLogAppender
```

After restarting, we can see that each `BundleName` bundle now has its own logfile, located at `data/log/BundleName.log`.

One of the best ways to monitor a Karaf instance is to tail a log. Tailing will actively monitor the log and display the updates in real time to the console. This is very handy when trying to debug installation issues on bundles or features.



On Windows, a program such as WinTail might be helpful.

On Linux-based systems, simply use `tail -f data/karaf.log` from the command prompt.

We can also tail from the Karaf console using the following command:

```
karaf@root> log:tail
```

It is recommended that you tail from a separate command window when monitoring, because you will need the console for starting and stopping bundles, and running basic administration commands.

System properties

The system properties are values that can alter how the Karaf instance is set up for runtime and are available as Java system properties. These values are different from the startup, in that, they define how the Karaf instance should be configured. The startup properties define what bundles will be activated and in what order we are using the start levels. This section we will show some of the default system properties that you will want to look at for your configuration.

Configuring Karaf

The first file to be aware of is the `etc/config.properties` file. There is no need to alter this file, any changes to the default configurations can be added to the `etc/custom.properties` file. The `custom.properties` file is referenced from the `config.properties` file so that users can override default configurations without having to alter system default properties. This also makes it easier for users to upgrade to newer versions of Karaf without having to piece together the `config.properties` files while figuring out what changed. The user can just move the `custom.properties` file to the new instance and it will be ready to go.

Clear as mud right? Let's illustrate with an example so you can see what we mean. Open the `etc/config.properties` file, and locate the following `karaf.framework` setting:


```
karaf.framework=felix
```

Now change this to `equinox`. But don't just jump the gun and change it here. This is a default setting, so it is best to leave this file alone. Go ahead and open the `etc/custom.properties` file. Now add the following at the end:

```
karaf.framework=equinox
```

If you have already started Karaf once, you will need to restart from a clean slate for the changes to take effect. That means shut down any Karaf instance you have open, delete the data directory, and then start the Karaf instance.

The preceding paragraph will apply to any configuration changes made. This is needed because Karaf will cache configuration settings, bundle installations, and feature deployments across startups.

 It is important to remember that Karaf caches bundle and configuration settings. When developing for Karaf, it is a good habit to always delete the data directory when restarting Karaf in order to avoid cached settings or bundles that might cause confusion. This can also be achieved using the `reboot -c` command from the command line.

Add custom property files into the mix by updating one of the following lines in the `etc/config.properties` file:

```
${includes} = jre.properties custom.properties my.properties
```

Or uncomment the following:

```
${optionals} = my.properties
```

Where the properties are included, it will change how Karaf will function if the file is missing. By putting the properties file in the `includes`, the Karaf instance will fail to start if that properties file is missing. By adding the properties file to the `optionals`, Karaf will just display a warning if not found and would start up regardless. But once this is done, make sure to create the file `etc/my.properties`.

Setting environment variables

In most cases you will want to update or change the memory setting for Karaf. To do this, edit the `bin/setEnv` file running from the `bin` directory:

```
export JAVA_MIN_MEM=128M # Minimum memory for the JVM
export JAVA_MAX_MEM=1024M # Maximum memory for the JVM
export JAVA_PERM_MEM=128M # Minimum perm memory for the JVM
export JAVA_MAX_PERM_MEM=256M # Maximum perm memory for the JVM
```

Configuring hot deployment

Another totally awesome feature in the OSGi container is the ability to hot deploy. This allows the user to simply drop in a JAR file or even a Spring or Blueprint-based XML file into the defined deploy directory and have it deployed without having to restart the Karaf container. Define where that folder is located in the `etc/org.apache.felix.fileinstall-deploy.cfg` file:

```
felix.fileinstall.dir      = ${karaf.base}/deploy
felix.fileinstall.tmpdir   = ${karaf.data}/generated-bundles
felix.fileinstall.poll     = 1000
```

First you have the location of the `deploy` directory defined by `felix.fileinstall.dir`. The default is `${karaf.base}/deploy`. This means that we can drop files in that location and they will be picked up by the framework and deployed into the runtime.

Next is the temporary storage area for the generated bundles, which is defined by `felix.fileinstall.tmpdir`. This is where the bundles are cached when generated for runtime.

Finally, there is the `felix.fileinstall.poll` property. This will change the time interval for scans of the hot deployment folder. The default is 1000 milliseconds.

Console configuration commands

All this is great, but how do we manage it after the Karaf container has started? I am glad you asked! This is where your handy-dandy command console comes in. Remember that list of common commands you learned in *Chapter 2, Commanding the Runtime*? That was just the tip of the iceberg. We saw earlier in this chapter that logging had a list of subcommands specific to configuring logging. Well, configuration has a similar set that helps manage the Karaf container during runtime.

The first thing to try is the `list` command that will display all the configurations that are manageable from the console:

`config:list`

This will give a list of properties printed to the console. The following is an example of what will be seen in the console for the hot deploy configuration talked about in the previous section:

```
Pid: org.apache.felix.fileinstall.d3c0a5d7-af3b-40f0-b1b0-29059a0fb6ef
FactoryPid: org.apache.felix.fileinstall
BundleLocation: mvn:org.apache.felix/org.apache.felix.fileinstall/3.2.6
Properties:
    felix.fileinstall.poll = 1000
    service.pid = org.apache.felix.fileinstall.d3c0a5d7-af3b-40f0-b1b0-29059a0fb6ef
    felix.fileinstall.dir = [installDir]/deploy
    service.factoryPid = org.apache.felix.fileinstall
    felix.fileinstall.filename = file: [installDir]/etc/org.apache.felix.fileinstall-deploy.cfg
    felix.fileinstall.tmpdir = [installDir]/data/generated-bundles
```

Now that the current property settings can be reviewed, making changes via the console is simple.

First get into the edit mode. We will continue using the hot deploy configuration for this example:

`config:edit [PID]`

Or, in this case, the following:

`config:edit org.apache.felix.fileinstall.d3c0a5d7-af3b-40f0-b1b0-29059a0fb6ef`

This tells the container that we are editing the configuration for the `fileinstall` implementation. Now edit any property available via this bundle using the following command:

`config:propset [property] [value]`


Change the value of the poller to 2000 milliseconds:

`config:propset felix.fileinstall.poll 2000`

Now that we have made the change, we need to apply it (or update it):

config:update

After doing the update, rerun `config:list` and ensure that the changes have been made. Reopen the configuration file and verify that it has also been updated.

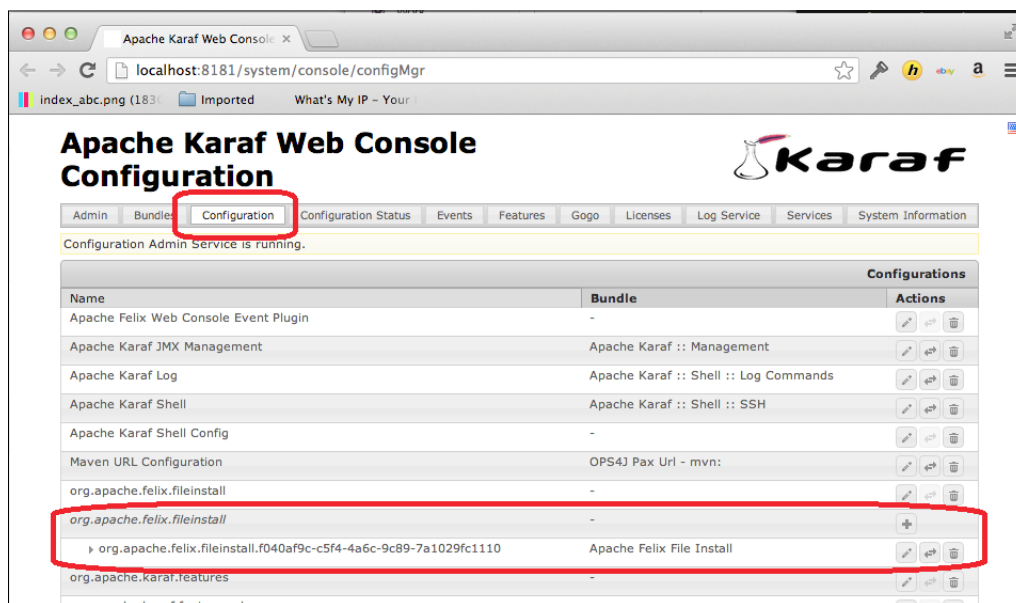
 Karaf uses several Apache Felix subprojects, so don't be alarmed when you see Felix configuration in the files. We can still use the Equinox container.

Changes can be rolled back or canceled using the following from the command line:

Config:cancel

Web console

As though configuration files and command consoles weren't enough, there is a third way to update and change configurations. Use the web console as described in *Chapter 2, Commanding the Runtime*. To stay consistent with the previous example, we will stick with altering the hot deployment configurations. The following screenshot demonstrates what the web console should look like and highlights the tabs and bundles to focus on:



Select the **fileinstall** row to get the **Configuration** properties panel to display the following screenshot:

org.apache.felix.fileinstall

FileInstaller	
Poll directory	/apache-karaf-2.3.1/deploy
Poll interval	3000
Log level	0
Start new bundles?	true
File name filter	
Temp directory	/apache-karaf-2.3.1/data/generated-bundles
No initial delay	true
Start bundles transiently	false
Use the bundle activation policy when starting	true
Bundles start level	0

Configuration Information

Persistent Identity (PID)	org.apache.felix.fileinstall.f040af9c-c5f4-4a6c-9c89-7a1029fc1110
Factory Persistent Identifier (Factory PID)	org.apache.felix.fileinstall
Configuration Binding	Apache Felix File Install (org.apache.felix.fileinstall), Version 3.2.6

Save Unbind Delete Reset Cancel

All the properties for the `fileinstall` are presented in a table, with the available functions displayed as buttons. Change the value of **Poll interval** to 3000 and click on **Save**. Open the configuration file `etc/org.apache.felix.fileinstall-deploy.cfg`; you'll notice that the change has been persisted back to the file.

Failover configuration

Karaf can also be set up in a master/slave configuration for high availability using either file locking or JDBC locking. To set up file locking on a Karaf instance, alter the `etc/system.properties` file by adding the following:

```
karaf.lock=true (this will indicate that locking should be used)
karaf.lock.class=org.apache.karaf.main.SimpleFileLock (the type of lock)
karaf.lock.dir=[PathToDirectory] (the location of the lock)
karaf.lock.delay=10 (indicates how often to check for lock)
```

Karaf has support for many different JDBC-compliant databases. They are listed as follows:

- **Oracle:** `org.apache.karaf.main.OracleJDBCLOCK`
- **MySQL:** `org.apache.karaf.main.MySQLJDBCLOCK`
- **PostgreSQL:** `org.apache.karaf.main.PostgreSQLJDBCLOCK`
- **Derby:** `org.apache.karaf.main.DerbyJDBCLOCK`
- **Default** (shown previously): `org.apache.karaf.main.DefaultJDBCLOCK`

Remember that the JDBC driver should be present in the Karaf classpath. This can be done by using the following two steps:

1. Copy the JDBC driver in `KARAF_HOME/lib`.
2. Add the `JDBC_DRIVER` path in startup script in `bin/karaf`:

```
CLASSPATH="$CLASSPATH:$KARAF_HOME/lib/JDBCJarFile.jar"
```

But for the purposes of this example, just use the file locking configuration. Now start up Karaf, and tail the logfile, we will see something like the following show up:

```
Aug 13, 2013 1:52:27 PM org.apache.karaf.main.SimpleFileLock lock
INFO: locking
```

This allows us to start a second instance of Karaf pointed at the same file location as the master, and it will only initialize to the point of acquiring a file lock and will wait (the level of initialization is configurable via container-level locking). Once the master instance fails/shuts down, the slave instance will get lock and continue to initialize. This gives the system a failover from one instance to another without any manual intervention.

Startup properties

The `etc/startup.properties` file is used to define the core services and their start levels on Karaf.

Take a look at the first couple of lines in the file:

```
org.ops4j/pax/url/pax-url-mvn/1.3.5/pax-url-mvn-1.3.5.jar=5
org.ops4j/pax/url/pax-url-wrap/1.3.5/pax-url-wrap-1.3.5.jar=5
```

All of the lines in this configuration define core service bundles and the start level for each.

These are not the only properties that can impact the Karaf startup. There is often the need to add features to the default installation. This can be done by altering the `etc/org.apache.karaf.features.cfg` file to include custom or optional features.

`featuresRepositories` is used to define the location of a features definition file that will get referenced during startup or make it available on a default installation.

`featuresBoot` lists the features that are to be loaded on startup. This is a comma-separated list. Adding features to this list will make them available and started immediately after initialization.



This configuration is cached and only accessed on a clean-slate startup, so any changes to this file requires the user to delete the `data/cache` directory in order for the changes to take effect.

Summary

There are many other ways to configure Karaf on startup, this chapter only gave a review of the most commonly used configurations. Karaf can be configured in an almost infinite number of ways, but that means care must be taken when altering or changing the configuration. Understand what is being changed and your chances for success are much greater. In this chapter, we covered the basics from memory settings to configuration changes at runtime. In the next chapter we will discuss how to set up the Maven repositories and feature descriptors.

4 Provisioning

One of the most important parts of developing and deploying dynamic systems is to have a flexible and accessible provisioning model. Such a system will allow you to build modern container-based applications that are easily updated, simple to use, and easy to document.

In this chapter we'll review the following topics:

- Apache Maven repositories
- The Karaf system repository
- Feature descriptors and building your own features

Apache Maven repositories

Apache Maven is named after a Yiddish word (<http://en.wikipedia.org/wiki/Maven>) meaning accumulator of knowledge.

Maven came about as an attempt to organize various Ant build files in the Jakarta Turbine project. Since its inception, it has grown to be a very commonly used build standard for Java projects with the following objectives, according to the Maven website:

- Making the build process easy
- Providing a uniform build system
- Providing quality project information
- Providing guidelines for best practices' development
- Allowing transparent migration to new features

Maven uses a fairly simple layout with a directory structure reflecting group, artifact, and versioning. This is also combined with a set of concepts such as `RELEASE` and `SNAPSHOT` allowing you to distinguish where in the development lifecycle you are, what you are building against, and how you control deployment and testing:

```
~/ .m2/repository/org/apache/camel/camel-core/2.11.0
~/ .m2/repository/org/apache/camel/camel-core/2.9.2
~/ .m2/repository/org/apache/camel/camel-core/2.10.4
```

This shows an example layout of the directory `~/ .m2/repository/org/apache/camel/camel-core`.

The `~/` notation is shorthand for your home directory in Unix-like shells.

Under a specific version you'll find the actual JAR files for the artifact as well as signature and metadata information:

```
~/ .m2/repository/org/apache/camel/camel-core/2.10.4
/camel-core-2.10.4.pom.sha1
~/ .m2/repository/org/apache/camel/camel-core/2.10.4
/camel-core-2.10.4.pom
~/ .m2/repository/org/apache/camel/camel-core/2.10.4
/camel-core-2.10.4.jar
~/ .m2/repository/org/apache/camel/camel-core/2.10.4
/camel-core-2.10.4.jar.sha1
```

Here you'll also find repository metadata information, error information if you are missing the ability to resolve an artifact, as well as the information about the last attempt that was made to download a particular artifact. When you are using Maven as a build tool, executing a clean installation of a JAR file will also populate your Maven repository. If you are doing that with a `SNAPSHOT` build, a timestamp-based JAR file will be deployed. This allows you to have projects that are dependent on each other as well as the ability to independently update the dependencies needed to build and deploy those projects.

If we look at a `SNAPSHOT` release directory of a different project, the layout is a little different. These artifacts are from an OpenNMS code branch. We are looking at a specific artifact in a larger project:

```
~/ .m2/repository/org/opennms/ng/opennms-distributed-ipc/1.0-SNAPSHOT

2383 Jul 19 14:23 opennms-distributed-ipc-1.0-SNAPSHOT.pom
11568 Aug 14 17:12 opennms-distributed-ipc-1.0-SNAPSHOT.jar
4673 Aug 14 17:12 opennms-distributed-ipc-1.0-SNAPSHOT-sources.jar
920 Aug 14 17:12 maven-metadata-local.xml
263 Aug 14 17:12 _maven.repositories
```

The interesting part here is the local metadata file:

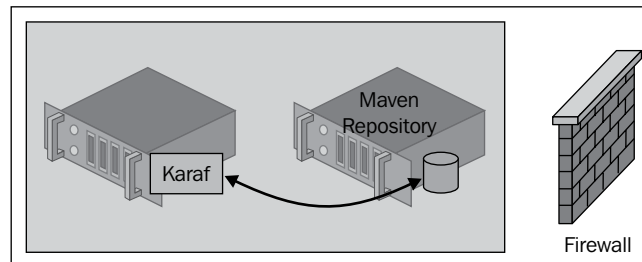
```
<?xml version="1.0" encoding="UTF-8"?>
<metadata modelVersion="1.1.0">
  <groupId>org.opennms.ng</groupId>
  <artifactId>opennms-distributed-ipc</artifactId>
  <version>1.0-SNAPSHOT</version>
  <versioning>
    <snapshot>
      <localCopy>true</localCopy>
    </snapshot>
    <lastUpdated>20130814231249</lastUpdated>
    <snapshotVersions>
      <snapshotVersion>
        <classifier>sources</classifier>
        <extension>jar</extension>
        <value>1.0-SNAPSHOT</value>
        <updated>20130814231249</updated>
      </snapshotVersion>
      <snapshotVersion>
        <extension>jar</extension>
        <value>1.0-SNAPSHOT</value>
        <updated>20130814231249</updated>
      </snapshotVersion>
      <snapshotVersion>
        <extension>pom</extension>
        <value>1.0-SNAPSHOT</value>
        <updated>20130814231249</updated>
      </snapshotVersion>
    </snapshotVersions>
  </versioning>
</metadata>
```

This file contains the full descriptor for the local status of this particular `SNAPSHOT` JAR; it is the file that Maven will consult as soon as you depend on this artifact.

To accomplish this Maven resolution, Karaf utilizes a subproject from OPS4J Pax URL. Pax URL provides a set of URL handlers that can be used in both OSGi and regular Java environments. They facilitate the building and resolution of URLs such as `mvn:`, `wrap:`, and `classpath:`, so that Karaf's deployment system can quickly and easily find the artifacts necessary for deployment.

These URL handlers are used in several places, such as the OSGi and features commands, that allow you to transparently load new OSGi bundles without having to learn the actual framework syntax and API structure for loading artifacts into the runtime.

Since the URL handlers respect all the Maven conventions as well as the settings that you provide of where to load, a common deployment for Apache Karaf is kept behind a firewall that utilizes a local proxy repository, as shown in the following figure:



Examples of such repositories are **Nexus** or **Artifactory**. They allow you to configure a corporate- or project-designated repository that is used for all artifact resolutions. Not only will this reduce network traffic in large networks, it also allows you to control what is actually resolved when pointing to a particular artifact.

This is controlled by the `org.ops4j.pax.url.mvn.cfg` configuration file; here you can list the repositories you wish to search when invoking Maven commands.

Two settings in particular are of interest, they control where Karaf looks for Maven artifacts and are shown in the following commands with their default values on a new download:

```
org.ops4j.pax.url.mvn.defaultRepositories=file:${karaf.home}/${karaf.default.repository}@snapshots@id=karaf.${karaf.default.repository}
org.ops4j.pax.url.mvn.repositories= \
    http://repo1.maven.org/maven2@id=central, \
    http://svn.apache.org/repos/asf/servicemix/m2-repo@id=servicemix, \
    http://repository.springsource.com/maven/bundles/release@id=springsource.release, \
    http://repository.springsource.com/maven/bundles/external@id=springsource.external, \
    http://oss.sonatype.org/content/repositories/releases/@id=sonatype
```

The first of the two defines a **SYSTEM** repository and the second provides us with a set of remote repositories that can be overridden by the end user to introduce a local Nexus proxy or similar.

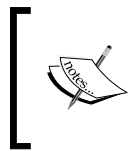
The Karaf system repository

Karaf also provides a second "system" repository; it is a simplified Maven layout directory that will be consulted by Pax URL prior to looking at your home directory repository and lastly resorting to utilizing preconfigured remote Maven repositories.

The system repository is the fastest way of loading artifacts into Karaf. It is also a very convenient place to put third-party dependencies in custom containers.

Correctly populating the system repository allows you to build a custom container that can be preconfigured with every single artifact necessary for offline usage.

The ability to accommodate offline usage is quite important in secure environments such as those requiring PCI or HIPPA compliance.



HIPPA and PCI are compliance programs, the latter is a program managed by the industry and the prior a government-regulated program. While sometimes producing similar results in terms of data audit and security, they are far from being the same.

The system directory is also utilized by a default Apache Karaf build for things such as Apache Aries Blueprint, command JARs, features JARs, Apache commons JARs, and so on, that are necessary to load in a specific order early on in the lifecycle of the container.

One file is used to control the startup of the core bundles needed to run Apache Karaf; these bundles are statically defined in a configuration file called `startup.properties`.

The following is a list of rules that the startup levels defined in Apache Karaf follow:

- Hot standby container (you need to enable failover and a true JDBC lock)
- A normal or a standby container
- Above this, user bundles are beginning to execute

From `etc/startup.properties` where the startup level is indicated as a property value:

```
#
# Startup core services like logging
#
org/ops4j/pax/url/pax-url-mvn/1.3.6/pax-url-mvn-1.3.6.jar=5
org/ops4j/pax/url/pax-url-wrap/1.3.6/pax-url-wrap-1.3.6.jar=5
org/ops4j/pax/logging/pax-logging-api/1.7.0/pax-logging-api-1.7.0.jar=8
```



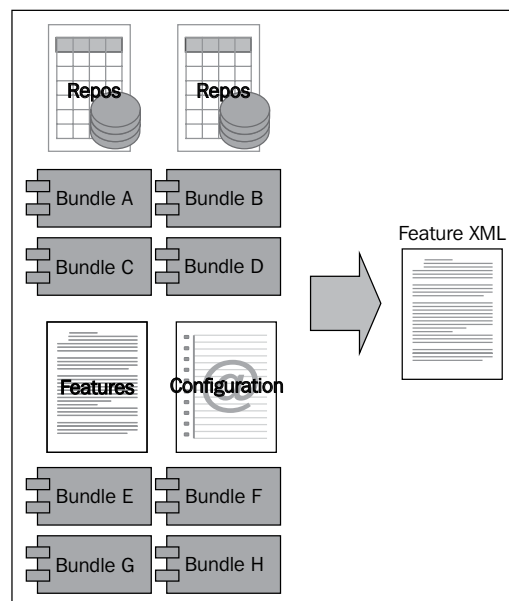
```
org/ops4j/pax/logging/pax-logging-service/1.7.0/pax-logging-service-1.7.0.jar=8
org/apache/felix/org.apache.felix.configadmin/1.6.0/org.apache.felix.configadmin-1.6.0.jar=10
org/apache/felix/org.apache.felix.fileinstall/3.2.6/org.apache.felix.fileinstall-3.2.6.jar=11
```

This will start the bundles for pax-url, pax-url-wrap (more about that in the next chapter), logging, and configuration admin. These are the core Karaf dependency JARs necessary for the operation of Apache Karaf. This file is consulted very early on and as you can see follows a "Maven-like" layout, where these files will be loaded as regular bundles via an OSGi bundle context. They will be prepended with a `${karaf.home}` environment variable so that the `system` folder is easily discovered.

Apache Karaf features

Apache Karaf provides you with a mechanism to build applications or more complex deployment descriptors. It is a complete solution for building and deploying "Applications". **Applications** here are defined as a set or several sets of JARs that together define what is necessary to have on the classpath in a running system. In the Apache Karaf world, this concept is called **features**. The features command and its related structure came about, as this is an area that is yet to be standardized in OSGi.

The features facility allows you to combine Maven, URL handlers, and other features into a features XML file, as illustrated in the following figure:



The complete schema for the feature descriptor can be found at <http://karaf.apache.org/xmlns/features/v1.1.0>, the biggest schema changes between Karaf 2.x and 3.x are around OBR resolution and forced naming changes.

A feature is an XML snippet containing a collection of bundles, features, configurations, and repository definitions. The features command suite in Apache Karaf manages the loaded feature descriptors. These commands allow you to add, remove, install, uninstall, and refresh features.

Several other projects such as Apache Camel, Apache CXF, and Apache ActiveMQ have adopted this structure and provide feature descriptors for easy deployment into Karaf. This is also the recommended way of developing your own applications.

A sample set of features (these are from the Apache Camel project) is as follows:

```
<feature name='camel-core' version='${project.version}'
  resolver='(obr)' start-level='50'>
  <feature version='${servicemix-specs-version}'>xml-specs-api</
feature>
  <bundle>mvn:org.apache.camel/camel-core/${project.version}</
bundle>
  <bundle>mvn:org.apache.camel.karaf/camel-karaf-commands/${project.
version}</bundle>
</feature>

<feature name='camel-spring' version='${project.version}'
  resolver='(obr)' start-level='50'>
  <bundle dependency='true'>mvn:org.apache.geronimo.specs/geronimo-
jta_1.1_spec/${geronimo-jta-spec-version}</bundle>
  <feature version='${spring-version-range}'>spring</feature>
  <feature version='[1.2,2) '>spring-dm</feature>
  <feature version='${spring-version-range}'>spring-tx</feature>
  <feature version='${project.version}'>camel-core</feature>
  <bundle>mvn:org.apache.camel/camel-spring/${project.version}</
bundle>
</feature>
```

These two features describe the camel-core and the camel-spring deployment—the two frequently used deployments in Karaf and Camel environments. The camel-core feature loads in the necessary XML tools, the base Camel JAR, and the Camel Karaf commands.

They also mark the spring-dm feature as allowed within a range; it has to be from Version 1.2 and no greater than Version 2.0 because using these ranges allow you to stay with a features dependency for a longer time, as long as you know they are binary compatible.

The Camel Karaf commands are Camel-specific extensions to the Apache Karaf command-line interface, allowing you to visualize and monitor Camel routes as though they were native Karaf commands. Later on in this book, **we will show you how to build and deploy your own commands.**

Once a Karaf feature is added with `features:addUrl`, you can use the `features` command set to manipulate them.

The second feature, `camel-spring`, is an interesting feature, as it is an aggregate of multiple features: Spring, Spring dynamic modules, Spring transaction, and finally `camel-core`. Both of these features also denote that they allow for "OBR" resolution (OBR is an emerging standard for resolution) and "OSGi bundle repository," which allows for partial resolution from an OBR bundle repository on a class instead of JAR. This leads to more finely tuned deployments, but as this isn't a deployment technology that is widely available, we'll just mention it as a side note here.



The aggregation shows how you can build very complex applications rapidly. The one thing to look out for is that dependency chains and features should not be cyclic, because this will lead to deployment loops. Because we don't want cyclic features, it is also a good time to point out that one of the key tenants of "best practice" modular development and OSGi deployment is that you maintain a good package structure, avoid split packages (packages that cross multiple jars), and refactor code that cross-import across JARs into independent JARs that can resolve on their own or in a simple chain.

This can be slightly cumbersome in refactoring existing legacy code, but Apache Karaf tries to provide tools to effectively illustrate dependency trees, header information, and the details of the origin of a particular import or export.

All URLs in the mentioned features utilize the `pax-url` MVN handler. Technically, you could write all your bundle entries with a complete qualified file or regular HTTP URLs in the following manner:

```
<bundle>http://repo1.maven.org/maven2/org/apache/servicemix/nmr/org.apache.servicemix.nmr.api/1.0.0-m2/org.apache.servicemix.nmr.api-1.0.0-m2.jar</bundle>
```

While doable, the recommended way of doing this is to use the MVN URL; it allows you to abstract the resolution mechanism away from the `feature` descriptor so that you can use the same features behind a firewall, completely connected to the Internet or completely offline. It simply becomes a task of providing the correct features resolution mechanism to your Karaf container.

Additional "features"

There are several other settings you can control in your feature descriptors. The following is an example of a feature that controls the `start-level` for all bundles deployed, this allows you to force and ensure ordering via `start-level`:

```
<feature name='my-project' version='1.1.0'>
  <feature version='2.4.0'>camel-spring</feature>
  <bundle start-level='80'>mvn:com/myproject-dao</bundle>

  <bundle start-level='85'>mvn:com/myproject-service</bundle>
  <bundle start-level='85'>mvn:com/myproject-camel-routing</bundle>
</feature>
```

In the preceding feature, the DAO JAR is started ahead of the service layer and camel routes. We can also nest features in features, thereby allowing us to build dependency chains.

Just like bundle imports, the version ranges too follow OSGi conventions, so we can say, for instance, that we want to be able to use versions from 1.x to 2.x but not above or below that feature range.

Perhaps the most useful construct is the ability to rely on other features repositories; this allows you to rely on existing projects without having to rewrite or copy over any complex features, it also helps you in not having to redo the work of larger existing projects:

```
<repository>mvn:org.apache.cxf.karaf/apache-cxf/${cxf-version}/xml/
features</repository>
<repository>mvn:org.apache.jclouds.karaf/jclouds-karaf/${jclouds-
version}/xml/features</repository>
```

Here we are reusing the features of Apache CXF and jclouds.

We can also inject configuration admin properties for use with our deployments:

```
<config name="com.packt.book">  myProperty = myValue </config>
```

We can inject configuration files too if necessary:

```
<configfile finalname="/etc/jetty.xml">mvn:org.apache.karaf/apache-
karaf/2.3.2/xml/jettyconfig</configfile>
```

This allows us to inject complex configurations as well. This is one of several ways of introducing configuration to your bundles as well as managing the configuration during deployment. It will interact with the Config Admin services, just as doing a file deploy into `${karaf.home}/etc` or setting Aries Blueprint or Spring Properties defaults will.

Summary

Apache Karaf allows you to provision artifacts in several ways, most importantly via Maven repositories and the `system` repository. Apache Karaf also provides you with an "application-building" facility named **features**.

Karaf uses Maven natively, you can start deploying straight out of the box from remote, local, or proxied Maven repositories. All metadata is respected and followed so that Karaf can be a part of your regular development cycle.

Apache Karaf also has a `system` repository – a simplified Maven structure that is prepopulated with the resources necessary to start Karaf. It is also a resource that you yourself as a developer can customize if needed. This will help with offline deployments, as no remote repositories need to be consulted.

Features are XML descriptors that allow you to combine JARs, other features, and configuration data into applications.

In the next chapter we'll see how features can be used in deploying complete applications.

5

Deploying Applications

By this point, Karaf should be installed using a stable configuration and we have a basic understanding on how to use the commands from the console. Now we get to be creative; here we learn about the numerous ways to deploy applications into the Karaf container. In this chapter, we will review six of the most common ways, which are listed as follows:

- Deploying bundles
- Deploying feature descriptors
- Deploying non-OSGi jars
- Deploying WAR
- Deploying Spring/Blueprint
- Creating and deploying a Karaf archive

Deploying bundles

In order to deploy a bundle, we first have to understand what a bundle is and how we create one. A **bundle** is a JAR file with an OSGi-compatible manifest file. The manifest file is a regular manifest file with additional OSGi-specific headers. Some of the more common headers you will notice in it are as follows:

- Bundle-Vendor
- Bundle-Name
- Bundle-DocURL
- Bundle-Description
- Bundle-SymbolicName
- Bundle-Version

- Bundle-License
- Bundle-ManifestVersion
- Import-Package
- Export-Package

The most common approach to creating an application bundle is to use Apache Maven, which allows you to automate the build process and generate a bundle with all required dependencies. Maven uses the **Apache Felix Bundle Plugin** to assemble and define the bundle based on XML configurations. The Felix Bundle Plugin utilizes the BND tool, which can also be used from the command line.

This can be configured by the developer to assemble the bundle in the best way possible for this deployment. What is the best way to deploy? That, of course, depends on what you are trying to achieve. It was recognized early on that forcing developers and admins to deploy in certain ways was not very flexible. In this section we will focus on deploying the bundle using the Maven repository.

Building a bundle

In the example discussed in *Chapter 2, Commanding the Runtime*, the `pom.xml` file demonstrated several important features of building a bundle.

First, let's look at the packaging type defined near the top of the file. It will read `bundle` as the packaging. This means that the build tool will create an OSGi-compliant JAR by adding the required headers to the manifest file:

```
<packaging>bundle</packaging>
```

The next part to be aware of is the `dependency` section. This defines all of the dependency requirements for this application. The following is an example of one of the dependencies required for the custom command bundle:

```
<dependency>
<groupId>org.apache.karaf.shell</groupId>
<artifactId>org.apache.karaf.shell.console</artifactId>
<version>2.3.2</version>
</dependency>
```

Finally, look at the bundle plugin. This defines and assembles the bundle for deployment:

```
<groupId>org.apache.felix</groupId>
<artifactId>maven-bundle-plugin</artifactId>
<configuration>
<instructions>
```

```

<Bundle-SymbolicName>${project.artifactId}</Bundle-SymbolicName>
<Export-Package>
    com.your.organization*;version=${project.version}
</Export-Package>
<Import-Package>!${project.artifactId}*,
    org.apache.felix.service.command,
    org.apache.felix.gogo.commands,
    org.apache.karaf.shell.console,
    *
</Import-Package>
<Private-Package>!*</Private-Package>
</instructions>
</configuration>

```

The configuration section tells the bundle plugin how to define the manifest file for the JAR. The imports and exports will be handled automatically by the plugin if they are not specified. It is important to note that if you do not add the required import packages, an exception of type `ClassNotFoundException` may possibly occur when deployed. Once this is deployed, they can easily be found using the `headers` command in the command console.

Build the bundle using the Maven command `mvn clean install` in order to get the bundle into the Maven repository. The bundle is *not* in the Karaf instance yet, it is just available to be deployed.

Deploying the bundle using Maven

Now that the bundle is in the Maven repository, the bundle can be deployed into the Karaf container using the Maven `url` handler command. To deploy the bundle, you need to know the `groupid` and `artifactid` for the bundle, which can be found in the `pom.xml` file:

```

<groupId>com.your.organization</groupId>
<artifactId>custom-command</artifactId>

```

A simple deployment command would look like the following using the `mvn` handler; the version on the end of the command is optional (Maven will use the latest version found if not specified):

```

> install mvn:com.your.organization/custom-command/1.0.0-SNAPSHOT
Bundle ID: 85.

```


That's it! The custom command application is deployed into the Karaf container. This can be confirmed by using the `list` command:

```
[85] [Installed ] [ ] [80] Apache Karaf :: Shell test Commands  
(1.0.0.SNAPSHOT)
```

Deploying a bundle using the file handler

There is always a need to deploy without using Maven, which is where the file handler comes in. The file handler can be used to deploy any OSGi JAR file without having to reach into the Maven repo. The path can be either an absolute or a relative path to the installation directory of Karaf:

```
install file://[PathToFile]/custom-command-1.0.0-SNAPSHOT.jar
```

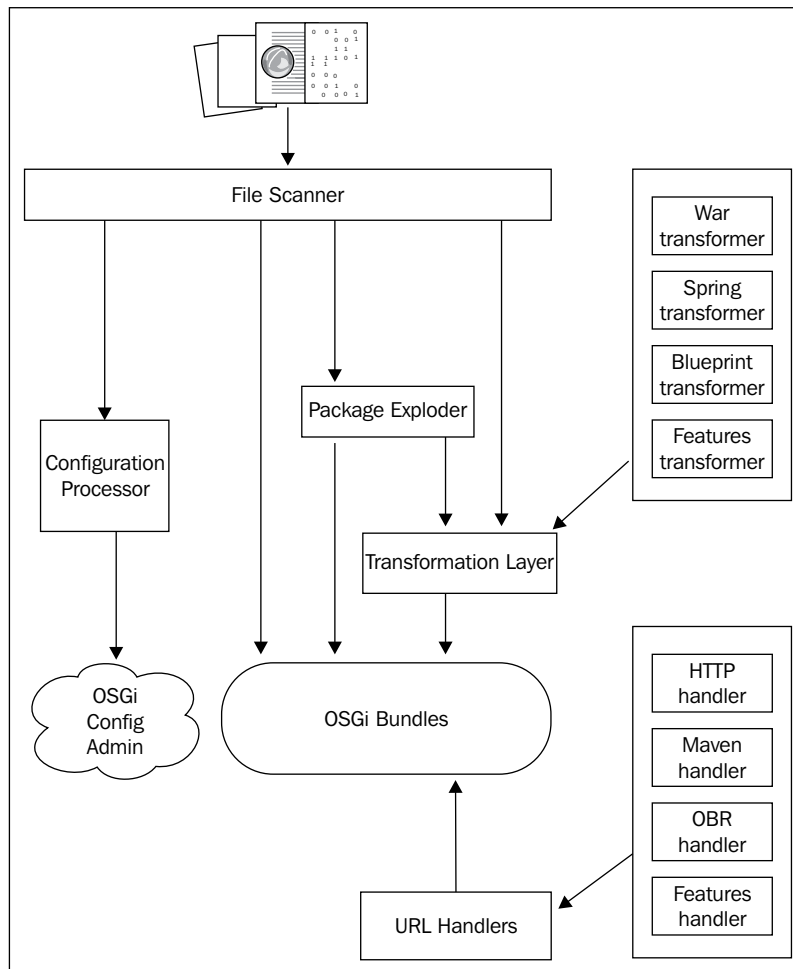
Deploying a bundle using HTTP

In some cases, the bundle file is not local and is only available via HTTP. The file could be downloaded manually and installed using the file handler. The HTTP handler provides the ability to call the URL directly using the `install` command, as shown in the following lines of command:

```
install http  
://[URLToFile]/custom-command-1.0.0-SNAPSHOT.jar
```

Deploying a bundle using hot deployments

The preceding command is pretty long and becomes cumbersome to type in if you are developing and often need to deploy bundles for testing. The `[karaf]/deploy` directory is used to quickly and easily deploy bundles into the Karaf container. Karaf monitors the `deploy` directory for changes; when a new file is put in that directory, it gets installed into the runtime and started. This means that updating or deleting a file from the `deploy` directory will also cause Karaf to either update or remove a bundle from the container:



The preceding figure shows how the hot deploy directory handles files are put into the `deploy` folder. When the scanner detects a change in a file, it will inspect that file and determine whether or not it is an OSGi bundle, a descriptor file (Spring or Blueprint), or some other type that can be transformed. If the file is an OSGi bundle, it will be processed in the following standard lifecycle for deployment:

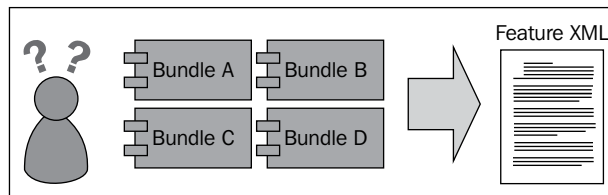
- **Install**
- **Resolve**
- **Starting**
- **Active**



While the hot deploy functionality is really cool, avoid using it for production deployments.

Deploying feature descriptors

feature descriptors are XML files that describe how to deploy a group of bundles or other features:



It can be thought of as a macro for deployments. It will detail what bundles need to be installed as well as the start level for deployment.

A feature file can be deployed in the following ways:

- Deployed as a bundle
- Dropped in the Hot Deploy directory
- Referenced on startup

The feature definitions may use any of the handlers available from the command line in order to install and start the required bundles. By now you should be able to see the power of features. This means that you can have a complex application with a very large number of bundles to install with dependencies on other features, and you can install it using a single command:

```
>features:install [feature-name]
```

An easy way to get started with features is to use a Maven archetype to build an example project:

```
mvn archetype:generate \
  -DarchetypeGroupId=org.apache.karaf.archetypes \
  -DarchetypeArtifactId=karaf-feature-archetype \
  -DarchetypeVersion=2.3.1 \
  -DgroupId=org.myorg \
  -DartifactId=myFeature \
  -Dversion=1.0.0-SNAPSHOT \
  -Dpackage=org.myorg.package
```

Once this runs successfully, go into the directory that was just created using the artifact ID as the name, and run `mvn clean install`. On completion, you will see a new folder structure `target/classes`, and in there will be your new `features.xml` file. This is a basic structure to start with:

```
<?xml version="1.0" encoding="UTF-8"?>
<features>
</features>
```

From here you can add in whatever bundles or features you like by altering the file to add in the custom command and the feature WAR. This example is just using the bundle from our previous examples and deploying the "WAR" feature as part of our application:

```
<?xml version="1.0" encoding="UTF-8"?>
<features>
<feature name="custom-command">
<bundle>
    mvn:com.your.organization/custom-command
</bundle>
<feature>war</feature>
</feature>
</features>
```

After altering the `target/classes/feature.xml` file, start up Karaf and execute the following command:

```
karaf@root> features:list
```

If this is a start from a clean slate, you will see that the `war` feature is available by default in the Karaf container, but is not installed:

```
[uninstalled] [2.3.1] war karaf-2.3.1 Turn Karaf as a full WebContainer
```

Now we can discuss the multiple ways of deploying this new feature into the Karaf container. First, let's look at the simplest way. Copy the file into the `[karaf]/deploy` directory, and that's it! You have deployed the custom command and have installed the war feature. To prove it, go to the console and use the `features:list` command again; this time you will see it in the list:

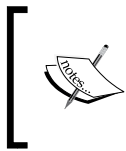
```
[installed ] [0.0.0 ] custom-command repo-0*
[installed ] [2.3.1 ] war          karaf-2.3.1  Turn Karaf as ...
```



* Installed via deploy directory - indicates that a feature was deployed via the deploy directory.

Uninstalling the feature is just as simple. Remove the `feature.xml` file from the `deploy` directory. This will uninstall the feature and remove the URL from the features list.

If that is just too simple, or you are deploying into production and need a scriptable command line for deploying, we can use the console.



When following along in Karaf, you will need to remove the `feature.xml` file from the `deploy` directory, otherwise you will continue to see the `war` feature installed and the `custom-command` in the `features:list`, even after deleting the `data` directory.

To deploy a feature via the console we first need to add the feature file. Karaf will parse the XML file and resolve the feature names for display in the console and show it as available to install:

```
karaf@root>features:addurlfile:/// [PathToFile] /feature.xml
```

Now you can do a `features:list` again and see that the `custom-command` is listed but not installed:

```
[uninstalled] [0.0.0          ] custom-command      repo-0
```

Adding the URL will process the `feature.xml` file but will not install the feature, leaving it in an uninstalled state. Now you can install the feature:

```
karaf@root> features:install custom-command
```

Uninstalling the feature is just as easy:

```
karaf@root> features:uninstall custom-command
```

In order to remove the feature from the list of features, you will need to remove the URL:

```
karaf@root>features:removeurlfile:/// [PathToFile] /feature.xml
```

Maven can also be used to create and deploy features. This means setting up a pom.xml file to assemble the required components to make the bundle. Let's look at the example pom.xml file:

```
<build>
  <resources>
    <resource>
      <directory>src/main/resources</directory>
      <filtering>true</filtering>
    </resource>
  </resources>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-resources-plugin</artifactId>
      <executions>
        <execution>
          <id>filter</id>
          <phase>generate-resources</phase>
          <goals>
            <goal>resources</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
    <plugin>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>build-helper-maven-plugin</artifactId>
      <executions>
        <execution>
          <id>attach-artifacts</id>
          <phase>package</phase>
          <goals>
            <goal>attach-artifact</goal>
          </goals>
          <configuration>
            <artifacts>
              <artifact>
                <file>target/classes/features.xml</file>
                <type>xml</type>
                <classifier>features</classifier>
              </artifact>
            </artifacts>
          </configuration>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

```
</configuration>
</execution>
</executions>
</plugin>
</plugins>
</build>
```

There are two parts to this build, so let's break it up in the explanation.

The first part is `maven-resources-plugin`, which allows you to filter resources and use Maven properties in your `features.xml` file. An example of a Maven property when used in the feature descriptor would be as follows:

```
<feature name='custom-command' version='${project.version}'>
```

This makes it possible to externalize property values for use while building the feature file.

The second plugin is configured to attach the filtered `features.xml` file. Look at the artifact elements, notice that there are three subelements defined that help to further define the file that is being attached:

- The `file` element defines the location of the filtered `feature.xml` file
- The `type` defines the file type, in this case it is `xml`
- The optional `classifier` feature helps to further define the configuration, and is more readable

Running `mvn clean install` on this pom will deploy the bundle into the Maven repository. Now we can use the Maven handler to load the URL:

```
karaf@root> features:addUrl mvn: org.myorg /myFeature/1.0/xml/features
```

This command makes Karaf aware of the feature definition but does not install it. To verify that the feature descriptor was added, we can use `features:list`.

Deploying non-OSGi JARs (wrap)

There are times when there is no OSGi-compliant bundle available for a third-party JAR that you need. Karaf supports the `wrap` protocol that provides the ability to alter a JAR to be OSGi-compliant. A good example of this is the `hsqldb.jar` file (Version 1.8.0.10):


```
Manifest-Version: 1.0
Ant-Version: Apache Ant 1.7.0
Created-By: 1.4.2_04-b05 (Sun Microsystems Inc.)
```

```

Specification-Title: HSQLDB
Specification-Version: 1.8.0.10
Specification-Vendor: The HSQLDB Development Group
Implementation-Title: Standard runtime
Implementation-Version: private-2008/06/01-10:22:29
Implementation-Vendor: ft
Main-Class: org.hsqldb.util.SqlTool

```

Dropping the JAR in the `deploy` directory will still install the JAR into the OSGi container even though the JAR is not OSGi compliant. What happens is the `wrap` protocol is automatically invoked. It will open the JAR and rewrite the manifest file.

 This will write the `manifest.mf` file to `import *`, which is not the best solution, but will work in a pinch.

There are several new entries when looking at the rewritten manifest file. The `wrap` protocol takes some liberties as to how to define many of the headers. Manifest headers can be displayed in the console using `osgi:headers` or `headers:`

```

Bundle-Name = hsqldb.jar
Bundle-SymbolicName = hsqldb.jar
Bundle-Version = 0.0.0
Bundle-ManifestVersion = 2

Private-Package =
Import-Package = javax.naming;resolution:=optional,
    javax.naming.spi;resolution:=optional,
    javax.net;resolution:=optional,
    .....
Export-Package = org.hsqldb.store;uses:=org.hsqldb.lib,
    org.hsqldb.types;uses:="org.hsqldb.lib,org.hsqldb",
    .....

```

Another way to deploy this JAR is to add it to a `features` file. It just so happens that we have one to work from. Adding the following to the `feature.xml` file defined previously will demonstrate this functionality:

```

<bundle>
    file:/// [PathToFile] /hsqldb/hsqldb.jar
</bundle>

```


Dropping the new `feature.xml` file in the `deploy` directory results in an error. What? An error? When defining the bundle in a feature, the `wrap` protocol is not automatically invoked. Checking the `[karaf]/data/log/karaf.log` file will expose the exception that was thrown:

```
Unable to install features
org.osgi.framework.BundleException: Jar is not a bundle, no Bundle-
SymbolicName file:/// [PathToFile]/hsqldb/hsqldb.jar
```

There was no indication on the console that there was a failure. The `log:display` command identified the issue. In the case of using the feature file, a `wrap` protocol must be specified as follows:

```
<bundle>
  wrap:file:/// [PathToFile]/hsqldb.jar
</bundle>
```

Deploying the new feature will result in the successful deployment of the wrapped `hsqldb.jar` file. This can be verified by using the `osgi:list` or `list` command from the console:

```
[86] [Active ] [      ] [ 80] wrap_file__PathToFile _hsqldb.jar (0)
```

The same file can be installed directly using the command line from the Karaf console. The `-s` parameter is used to automatically start the bundle:

```
karaf@root>install -s wrap:file:/// [PathToFile]/hsqldb.jar
```



The `wrap` handler can be used with the other URL handlers as well (`mvn`, `http`, and so on).

At this point, we have identified several ways to get a non-OSGi-compliant JAR deployed into the runtime. But what do we do if we need more control over how the manifest is written? The `shade` plugin provides the capability to package an artifact in a special JAR, including its dependencies, and to shade (rename) the packages of some of the dependencies. To achieve fine-grained control over the classes from which the selected dependencies are included, `artifact` filters can be used:

```
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-shade-plugin</artifactId>
<version>2.1</version>
<executions>
<execution>
<phase>package</phase>
<goals>
```

```

<goal>shade</goal>
</goals>
<configuration>
<transformers>
<transformer implementation="org.apache.maven.plugins.shade.resource.
ManifestResourceTransformer">
<manifestEntries>
<Bundle-Name>hsqldb-shade</Bundle-Name>
<Bundle-Vendor>hsqldb</Bundle-Vendor>
<Bundle-Version>1.8.0.10</Bundle-Version>
<Bundle-ManifestVersion>2</Bundle-ManifestVersion>
<Bundle-Description>
    shaded version of the hsqldb jar
</Bundle-Description>
<Bundle-DocURL>myProject.com</Bundle-DocURL>
<Export-Package>
    org.hsqldb.index;uses:=org.hsqldb
</Export-Package>
<Import-Package>
    javax.naming;resolution:=optional
</Import-Package>
<Bundle-SymbolicName>
    org.hsqldb-shaded
</Bundle-SymbolicName>
</manifestEntries>
</transformer>
</transformers>
</configuration>
</execution>
</executions>
</plugin>

```

The above pom entry will take the JAR and rewrite the manifest.mf file to be OSGi-compliant.



Many of the third-party JARs are available in OSGi compliant bundles, so look for them before shading.

Deploying WAR

With OSGi becoming the runtime container of choice, there needs to be ways to deploy different types of archives. Deploying a WAR file is also supported within the Karaf container. In order to get a WAR file to deploy correctly and start, the war feature needs to be installed first. Luckily, we have already started the war feature several times throughout this chapter. In case the Karaf instance has been reset to a clean slate, let's install the war feature again:

```
karaf@root> features:install war
```

There, simple enough, now we are ready to install a WAR file of choice.

For this example I am using the non-OSGi-compliant `sample.war` file from Apache Tomcat available at <http://tomcat.apache.org/tomcat-5.5-doc/appdev/sample/sample.war>.

Just like the JAR file, the WAR file can be dropped in the `[karaf]/deploy` directory and it will be picked up, transformed to OSGi-compliant, installed, and started:

```
[ 88] [Active      ] [          ] [ 80] sample (0.0.0)
```

Once it is installed and is shown as active, you can go to <http://localhost:8181/sample/hello.jsp> and see the application:



Even though this WAR is not OSGi-compliant, the Karaf container is transforming it using the PAX Web Extender. This is fine for development, but in a production environment, you should control the transformation using Maven or the command line, as we do with the JAR. The WAR has a few other manifest headers that need to be addressed:

```
Manifest-Version = 1.0
Bnd-LastModified = 1377097343444
Tool = Bnd-0.0.357
Originally-Created-By = 1.5.0_06-b05 (Sun Microsystems Inc.)
Ant-Version = Apache Ant 1.6.5
```

```

WAR-URL = file:[PathToFile]/sample.war
Generated-By-Ops4j-Pax-From = file:[PathToFile]/ sample.war
Web-ContextPath = /sample
Created-By = 1.6.0_51 (Apple Inc.)

Bundle-Name = sample
Bundle-SymbolicName = sample
Bundle-Version = 0.0.0
Bundle-ManifestVersion = 2
Bundle-ClassPath = WEB-INF/classes

Import-Package =
    javax.servlet,
    javax.servlet.http,
    .....

```

The headers were automatically generated by PAX Web, and in most cases, these were not what you would want to use. Take `Web-ContextPath` for example; this is the path to your web application, which will now be as follows:

```
http://localhost:8080/sample
```

Let's look at a more controlled way to deploy the WAR file, so that we can change the context path and symbolic name to something more meaningful by using the `war` or `webbundle` protocol provided by Pax Web:

```

karaf@root> osgi:install -s webbundle:file:/// [PathToFile]/sample.
war$Bundle-SymbolicName=tomcat-sample&Webapp-Context=/mySampleApp

```

Notice the header definitions after `$`. This provides a way to change the header values from the command line. Now you would access the web page through the following URL:

```
http://localhost:8080/mySampleApp
```

Deploying Spring/Blueprint

Blueprint and Spring DM configuration files can also be deployed using the `[karaf]/deploy` directory. The Blueprint or Spring transformer will parse the name of the configurations file to get the `bundle-symbolicName` and the `bundle-version` values. The import packages are generated from the configurations and will include the referenced classes. The manifest, after deploying a file named `blueprint-config-1.2.3.xml`, would have the following headers:

```

Manifest-Version: 2
Bundle-SymbolicName: blueprint-config

```

```
Bundle-Version: 1.2.3
Import-Package: [referenced packages]
DynamicImport-Package: *
```

The transformers for both Spring and Blueprint have the ability to process manifest header definitions from the XML file. So if you need to customize the header values, just add the `manifest` tag to the configuration file in the following manner:

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0">
  <manifest xmlns=
    "http://karaf.apache.org/xmlns/deployer/blueprint/v1.0.0">
    Bundle-Description="my awesome bundle"
  </manifest>
```

The transformers can also be utilized at the command line in the console using the individual protocols `spring` and `blueprint`:

```
install -s spring:xxx:/// [PathToFile]
```

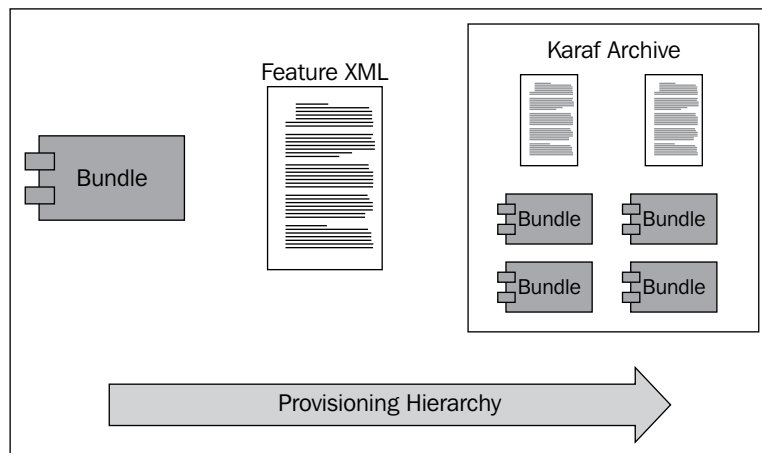
Or:

```
install -s blueprint:xxx:/// [PathToFile]
```

Where `xxx` is the location protocol (for example, `mvn`, `file`, `http`, and so on).

Creating and deploying a Karaf Archive

A Karaf Archive (KAR) file is basically a JAR file that contains feature descriptors and bundle JARs that can represent a full or partial deployment:



This makes it so you no longer have to copy individual JARs into local repositories. It also avoids the need to have Maven-based resolution for production environments. Allow the production environment to be completely isolated if need be, no need to have a Nexus repo or Maven repo local to the production system. KARs can also be referenced as dependencies from Maven builds, making it easier to reference a project and its dependencies.

Maven can be used to assemble the KAR just as easily as creating a bundle. The purpose of the KAR archive is to perform the following:

- Read the feature descriptors for all feature definitions
- Resolve the bundles defined by the features
- Package all bundles, configurations, and dependencies required by the features

When using Maven to build the KAR, there are two parts of the process that need to be addressed. First is the `pom.xml` file that will utilize `features-maven-plugin`. Use the `create-kar` goal to create the KAR archive and list the `features.xml` files that it utilizes to assemble the bundles:

```
<groupId>my.groupId</groupId>
<artifactId>my-kar</artifactId>
<version>1.0</version>
<packaging>pom</packaging>

<dependencies>
<dependency>
<groupId>com.your.organization</groupId>
  <artifactId>custom-command</artifactId>
<version>1.0.0-SNAPSHOT</version>
<scope>runtime</scope>
</dependency>
</dependencies>
<build>
<plugins>
<plugin>
<groupId>org.apache.karaf.tooling</groupId>
<artifactId>features-maven-plugin</artifactId>
<version>2.2.5</version>
<executions>
<execution>
<id>create-kar</id>
<goals>
```

```
<goal>create-kar</goal>
</goals>
<configuration>
<featuresFile>
${project.base}/[PathToFile]/features.xml
</featuresFile>
</configuration>
</execution>
</executions>
</plugin>
</plugins>
</build>
```

The other piece to this puzzle is the reference to the `features.xml` file. The following is a simple `feature` file that will package the `custom-command` bundle:

```
<?xml version="1.0" encoding="UTF-8"?>
<features>
<feature name="my-kar" version="1.0">
<bundle>
mvn:com.your.organization/custom-command/1.0.0-SNAPSHOT
</bundle>
</feature>
</features>
```

Now you can simply run the `mvn clean install` command against the `pom` file and your KAR file will be accessible in the target directory. Copy that KAR file into the `[karaf]/deploy` directory, then list features on the command console.

```
[installed] [1.0] my-kar repo-0
```

You can see the information on that deployed KAR by using the `features:info my-kar`:

```
karaf@root> feature:info my-kar

Feature my-kar 1.0
Feature has no configuration
Feature has no configuration files
Feature has no dependencies.
Feature contains followed bundles:
mvn: com.your.organization/custom-command/1.0.0-SNAPSHOT
```

Summary

We've just discovered that there are many ways to get our applications defined and deployed. All of these options are available to provide flexibility. There is no one correct way to deploy, it will all depend on what you are trying to achieve and what fits best into your software and hardware requirements.

6

Deploying Production-grade Apache Karaf

We have touched upon many of Karaf's capabilities and features in the preceding chapters. Now, we'll focus on ruggedizing our container for production environments.

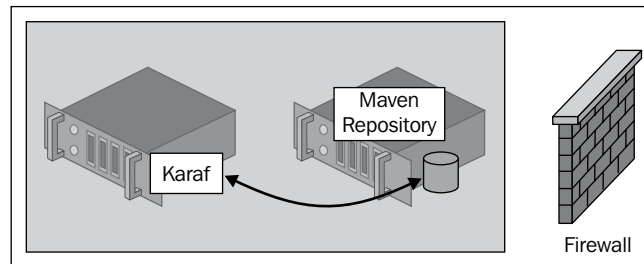
In this chapter we'll review:

- Offline repositories
- Improving application logging
- High availability / failover
- Basic security configuration

Offline repositories

Core to Apache Karaf's ability to provision bundles, features, configuration, configuration files, and other artifacts is Apache Maven. By default, Maven will seek out resources available locally in your local `m2` repository or in Karaf's built-in system folder (the system folder has the highest priority by default). If it fails to find a resource, it will venture onto the internet to obtain the artifact through a number of public repositories, as defined in `etc/org.ops4j.pax.url.mvn.cfg` file.

This functionality in practice works great; however, in production systems, this behavior of going to the internet to obtain resources could lead to problems (for example, resources may no longer be publically available or your deployment may not have access to the internet). Our best practice is to set up an offline repository that contains all the resources your application(s) require for deployment:



The offline Maven repository is set up on a machine behind your firewall (as shown in the previous figure) such that only selected members of your organization have rights to populate or access the repository.

How to build an offline repository

The system folder is an **OSGi Bundle Repository (OBR)** that uses a Maven folder structure: `groupId/artifactId/version/artifactId-version[-classifier].extension`.

You can copy artifacts in the system folder as soon as you use this folder structure.

To simplify the population of the system folder, you can use a Maven plugin provided by Karaf.

Creating your own offline repository consists of the following steps:

1. **Installing Java and Maven on a target server:** Follow the installation instructions for JDK 1.7 as provided by the vendor. Download and install Apache Maven 3.0.x; this requires unzipping the archive in its desired home directory. You may add `JAVA_HOME` and `MAVEN_HOME` to the system environment variables and `PATH`.
2. **Creating a Maven POM-based project structure:** Create a project folder, `/offlineRepo`, and skeleton POM file, `/offlineRepo/pom.xml`. This skeleton POM file will contain your Maven project coordinates (`groupId`, `artifactId`, and `version`).

3. **Adding Karaf's features-maven-plugin to the build:** Add `features-maven-plugin` to the project's `build` element. The plugin contains two configuration elements: `descriptors` and `features`. The `descriptors` configuration element is where you make the plugin aware of available descriptor files. Since descriptor files commonly reference other descriptor files, it is advisable to add all known files to this list. The `features` configuration element is where you make the plugin aware of the specific features you want to have populated into your offline repository:

```
<project>

<build>
  <plugins>
    <plugin>
      <groupId>org.apache.karaf.tooling</groupId>
      <artifactId>features-maven-plugin</artifactId>
      <version>2.2.1</version>

      <executions>
        <execution>
          <id>add-features-to-repo</id>
          <phase>generate-resources</phase>
          <goals>
            <goal>add-features-to-repo</goal>
          </goals>
          <configuration>
            <descriptors>
              <!-- Feature Descriptor Files -->
              <descriptor>
mvn:org.apache.karaf.assemblies.features/standard/2.3.2/xml/
features
              </descriptor>
              <descriptor>
mvn:org.apache.karaf.assemblies.features/enterprise/2.3.2/xml/
features
              </descriptor>
            </descriptors>
            <features>
              <!-- Features to add to offline repository -->

              <feature>webconsole</feature>
              <feature>config</feature>
            </features>
            <repository>target/offline-
repository</repository>
```

```
        </configuration>
      </execution>
    </executions>
  </plugin>
</plugins>
</build>
</project>
```

The preceding `features-maven-plugin` configuration will pick up the `webconsole` and `config` feature descriptor resources and construct an offline repository in `target/offline-repository`.

4. **Build the offline repository based upon the configuration:** After completing our configuration of the POM file, we may build the offline repository by invoking the following:

```
# mvn generate-resources
```

This will create our offline repository in `/offlineRepo/target/offline-repository`. Exploring the generated folder, you will find all of the dependencies cited in your feature descriptor projects.

5. **Install the generated repository:** Finally, we install the generated repository into Karaf by configuring `etc/org.ops4j.pax.url.mvn.cfg` to include our repository as follows:

```
org.ops4j.pax.url.mvn.defaultRepositories=file:${karaf.
home}/${karaf.default.repositories}@snapshots,/offlineRepo/target/
offline-repository@snapshots
```

We add `@snapshot` to indicate that some resources may be snapshots. Generally, we recommend only deploying release versions of code; however, in some circumstances, snapshot artifacts may be required.

Improving application logging

As discussed in *Chapter 3, System Configuration and Tuning*, Apache Karaf uses Pax Logging to manage its log output. This significantly simplifies administration by unifying the configuration of multiple appenders in one location (`etc/org.ops4j.pax.logging.cfg`). The default configuration, however, is not fully optimized for production environments; the file appender configuration can be tweaked:

```
# File appender
log4j.appender.out=org.apache.log4j.RollingFileAppender
log4j.appender.out.layout=org.apache.log4j.PatternLayout
log4j.appender.out.layout.ConversionPattern=%d{ISO8601} | %-5.5p |
%-16.16t | %-32.32c{1} | %-32.32C %4L | %X{bundle.id} - %X{bundle.name} -
%X{bundle.version} | %m%n
log4j.appender.out.file=${karaf.data}/log/karaf.log
log4j.appender.out.append=true
log4j.appender.out.maxFileSize=50MB
log4j.appender.out.maxBackupIndex=100
```

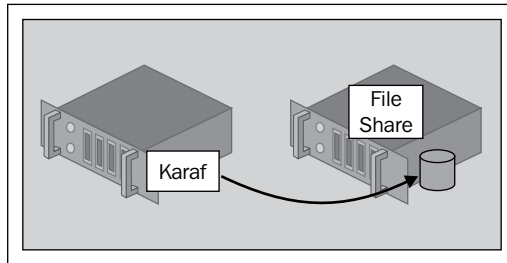
In the preceding configuration we have increased the maximum logfile size to 50 MB and the number of retained logfiles to 100. These settings will require up to 5 GB of disk space to be available just for the logfiles. This may sound unreasonable at first, but keep in mind that at debug-level output Karaf and its deployed applications can fill your logfiles at an impressive pace.

Our next alteration will be to relocate the log folder to outside the KARAF_HOME/data directory:

```
# File appender
log4j.appender.out=org.apache.log4j.RollingFileAppender
log4j.appender.out.layout=org.apache.log4j.PatternLayout
log4j.appender.out.layout.ConversionPattern=%d{ISO8601} | %-5.5p |
%-16.16t | %-32.32c{1} | %-32.32C %4L | %X{bundle.id} - %X{bundle.name} -
%X{bundle.version} | %m%n
log4j.appender.out.file=${karaf.base}/log/karaf.log
log4j.appender.out.append=true
log4j.appender.out.maxFileSize=50MB
log4j.appender.out.maxBackupIndex=100
```

We relocate the logfiles to be outside the data folder for administrative convenience. The content of the Karaf data folder are by-products of the OSGi container environment; from time to time, administrators may need to delete this content. Having the log folder outside this location removes the opportunity for an overzealous deletion taking out the logs.

Lastly, after reconfiguring the file appender thusly, we should also consider the impact this will have on our host system. If substantial logging is expected to be maintained in production, we should consider offloading the disk I/O impact:



For high performance logging, we point to a mounted file share (as shown in the previous figure). This is useful when Karaf is located on a server blade with poor I/O characteristics, as depicted in the preceding diagram.

It's also possible to add additional appenders (or custom appenders).

High availability / failover

Now that we've tweaked our Karaf configuration into a position to run for an extended period, let's discuss how we can help ensure continued service in the event of a fault.

Installing Karaf as a service

The first step towards a high availability / failover topology is to install Karaf as a service on your host operating system.

Let's review the service setup process.

Install the wrapper feature:

```
karaf@root> features:install wrapper
```

Instruct the wrapper to install Karaf as a service, using the following command:

```
karaf@root> wrapper:install -s AUTO_START -n KARAF -d Karaf -D "Karaf Service"
```

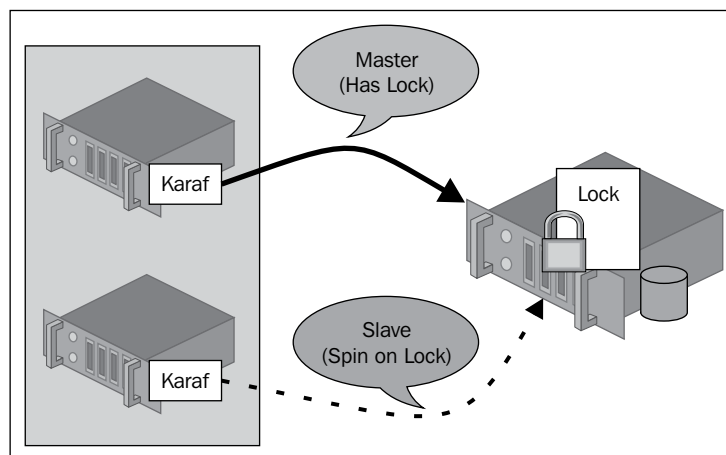
The first flag option (`-s`) tells the service to automatically start, the second (`-n`) sets the service name, the third (`-d`) the display name, and finally, the last flag (`-D`) sets the service description.

Now we can register the Karaf service. The method for performing this will vary by host operating system. For example, on Red Hat-based systems, the generated Karaf service file may be linked to `/etc/init.d`; in that case, use the `chkconfig` command to add the service and the `start` command to launch the service.

Now when we reboot the host machine, Karaf will automatically be started. At this point, it is safe to uninstall the wrapper feature, as the service wrapper is itself installed and configured.

Master-slave failover

To make the services offered by applications deployed in Karaf more resilient, we'll now configure and deploy Karaf in a master-slave (active-passive) topology, as shown in the following figure:



The master-slave topology depicted in the preceding screenshot utilizes a simple locking mechanism in which only one Karaf instance may control a `lock` resource and thereby achieve full runtime. All other Karaf instances attempting to gain the lock will spin, retrying to gain control. Until they gain control, they will not reach full runtime.

The simplest locking mechanism is a shared file on an **NFSv4** mount. We use NFSv4 as it properly implements `flock`, allowing just one process to hold the file. The configuration for this deployment has two or more Karaf instances with the following entries in their `etc/system.properties` file:

```
#append to system.properties file.
karaf.lock=true
karaf.lock.class=org.apache.karaf.main.SimpleFileLock
karaf.lock.dir=<PathToLockFileDirectory>
karaf.lock.delay=10
```

Alternatively, JDBC may be used for the locking mechanism. In this setup, a database provides a lockable resource over which Karaf instances vie for control. Each Karaf instance will contain the following entries:

```
#append to system.properties file.
karaf.lock=true
karaf.lock.class=org.apache.karaf.main.DefaultJDBCLock
karaf.lock.level=50
karaf.lock.delay=10
karaf.lock.jdbc.url=jdbc:derby://dbserver:1527/sample
karaf.lock.jdbc.driver=org.apache.derby.jdbc.ClientDriver
karaf.lock.jdbc.user=user
karaf.lock.jdbc.password=password
karaf.lock.jdbc.table=KARAF_LOCK
karaf.lock.jdbc.clustername=karaf
karaf.lock.jdbc.timeout=30
```

The appropriate lock class, URL, and JDBC driver must be provided to Karaf for the instance to be able to attempt to obtain the lock (see the Karaf manual for currently supported databases and their lock implementation class names).



Karaf Lock Level

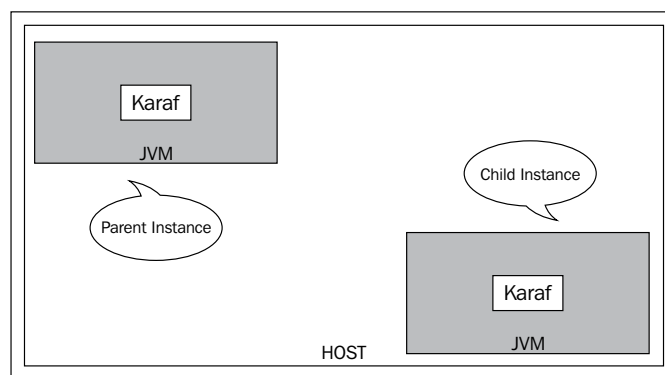
Apache Karaf supports container-level locking; this allows Karaf to let the OSGi environment to start bundles up to a defined run level before gaining the lock and fully starting. The advantage this provides is that a passive Karaf instance will have less work to do to become an active container – all the core bundles will be already loaded.

Child instances

Another form of high availability offered by Karaf is the use of **child instances**.

Child instances come in two varieties: bare environments and clones. The bare environment is essentially a second Karaf installation that has had all of its ports automatically selected to avoid bind exceptions at runtime on the same host. A clone instance is a copy of the parent container, including its deployed applications and configuration (its ports have also been autoselected to avoid port conflicts). Each instance runs in its own JVM, thereby providing process-level isolation.

A pragmatic approach to high availability using child instances will have you deploying applications that are not vetted to your child instance(s) before migrating them into your parent container:



In the preceding diagram we can see the relationship of the parent Karaf instances with its child. If the application that is not vetted and deployed in the child instance was to crash the container, the remaining applications in the parent would continue unhindered. Following this approach, you may graduate new applications to the parent over time.

You can create a bare environment child instance by invoking the following command:

```
karaf@root> create instanceName
```

You can create a clone child instance by invoking the following command:

```
karaf@root> clone existingInstance cloneName
```

In both cases the child instance will be created in `KARAF_HOME/instances`.

Karaf provides a set of administrative commands for managing child instances; under Karaf 2.3, these can be accessed in the admin scope.

Determining which type of child instance to use in your deployments will depend upon your applications and their dependencies. Generally, clones serve the purpose of testing a deployment in an existing Karaf environment, while the bare environment provides a clean room to experiment in.

Basic security configuration

Prior to deploying a Karaf instance in production, some considerations must be taken with regards to securing the container. In this section we are going to discuss how to lock down Karaf management; application-level security is beyond the scope of this book.

Out of the box, Apache Karaf provides default login credentials; it is best practice to change these values immediately. You can find their configuration in `KARAF_HOME/etc/users.properties`. Each entry in this file is of the format: `user=password[,role] [,role] [,role]` It is important that we change these values, as these credentials allow access to the remote console, JMX management, and the optional web console.

Karaf uses **Java Authentication and Authorization Service (JAAS)**. By default, you have one realm (named `karaf`) that uses two login modules (`PropertiesLoginModule` uses `etc/users.properties` and a key login module).

You can change the Karaf realm by adding/replacing the login modules. Karaf provides `LDAPLoginModule`, `JDBCLoginModule`, and `OSGiLoginModule` (to use `LoginModule` as an OSGi service).

You can also create a new realm (used by your applications). Realm and login modules can be described using Apache Blueprint and handled using the `jass:*` shell commands (and corresponding JMX Mbeans).

Managing roles

By default the admin role provides a user full access to all Karaf's management facilities; this policy, however, may be too generous. To restrict how much access we give a particular user, we set up role group names for each of the access layers and then apply access in the `users.properties` file.

For example, we may modify the configuration files as shown in the following table:

Files	Commands
<code>etc/org.apache.karaf.shell.cfg</code>	<code>sshRole = remoteShell</code>
<code>etc/org.apache.karaf.management.cfg</code>	<code>jmxRole = jmx</code>
<code>etc/org.apache.karaf.webconsole.cfg:</code>	<code>role = web</code>



This requires installation of Karaf's optional web console.

Then, in `users.properties`, update/add the following users:

```
testSSH=test,remoteShell
testJMX=test,jmx
testWEB=test,web
testALL=test,remoteShell,jmx,web
```

Attempt to access the various management layers of Karaf with these test accounts.



Admin role not working?

Once we assign roles to the three access layers, the default admin role will no longer be useable for authentication.

Password encryption

Changing usernames, passwords, and setting roles is a good start; however, this leaves clear text passwords in the `configuration` folder. To resolve this issue, we may enable password encryption in Karaf.

To enable encrypted passwords, we edit `/etc/org.apache.karaf.jaas.cfg`, changing the entry `encryption.enabled` to `true`. Upon the next login to Karaf, the user's password in `/etc/users.properties` will become encrypted.



Identifying encrypted passwords

When encryption is enabled, stored passwords in `users.properties` will be marked with `{CRYPT}` as follows:

```
karaf = {CRYPT}e7ebf747769e8522b52d1bf47f718788{CRYPT},admin
```

Locking down JMX access

Finally, when deploying Karaf, we should lock down JMX access to the instance. We can accomplish this quickly by editing the values in `etc/org.apache.karaf.management.cfg` and `etc/user.properties`.

In the management configuration file, we can simply uncomment the `jmxRole=admin` entry; then in `user.properties`, append `jmxRole` to our default karaf user. This simple procedure will make remote connections require authentication.

In JConsole, enter your remote URL in the following format:

```
service:jmx:rmi://{hostname}:44444/jndi/rmi://{hostname}:1099/karaf-root
```

If you desire to change the ports Karaf listens to, you may alter the port numbers in the management configuration file.

Summary

We've taken a step back to consider production operating concerns for our Karaf deployment. We have discussed using an offline repository to control what our production Karaf instances may access for provisioning. We've tweaked our logging configuration to be more durable and administrator-friendly. We then discussed at length various deployment options to improve service availability. Finally, we started the process of locking down our deployment with a basic security setup.

In the next chapter we'll depart from core Karaf concerns, and delve into Cellar – the Hazelcast-powered clustering solution for Karaf.

7

Apache Karaf Cellar

Apache Karaf has its own approach to clustering. Apache Cellar is used to provide a clustering solution powered by Hazelcast. The core concept behind Cellar is that each Karaf instance is a node that can be deployed as part of a group or groups, while providing synchronization between them. This can be very powerful for controlling both scaling and high availability. In this chapter we will cover:

- Node discovery
- Cluster groups
- Distributed configuration admin, features, and bundles
- Cloud discovery

Getting started

Cellar is not part of the default installation provided by Karaf. In order to install it, the URL for the feature definition must be added to the container as follows:

```
karaf@root> features:addurl mvn:org.apache.karaf.cellar/  
apache-karaf-cellar/2.3.1/xml/features
```

Once the container knows about the Cellar features, the `install` command can be utilized:

```
karaf@root> features:install cellar
```

The following is the list of the `cluster` commands that are available by installing the Cellar feature:

```
karaf@root> cluster:<TAB>  
cluster:config-list cluster:config-proplist  
cluster:config-propset cluster:consumer-start
```

```
cluster:consumer-status cluster:consumer-stop
cluster:features-install cluster:features-list
cluster:features-uninstall cluster:group-create
cluster:group-delete cluster:group-join
cluster:group-list cluster:group-quit
cluster:group-set cluster:handler-start
cluster:handler-status cluster:handler-stop
cluster:list-nodes cluster:ping
cluster:producer-start cluster:producer-status
cluster:producer-stop
```

We will not cover the uses of every command, but we will touch on all of the commonly used functions in order to get a cluster up and running.

In order to see the nodes in the cluster, you can use the command `cluster:list-nodes`. This will list all nodes currently connected to this instance. The instance being used is indicated by an `*` in the list:

```
karaf@root> cluster:list-nodes
No. Host Name Port ID
* 1 node1.local 5701 node1.local:5701
  2 node2.local 5702 node2.local:5702
```

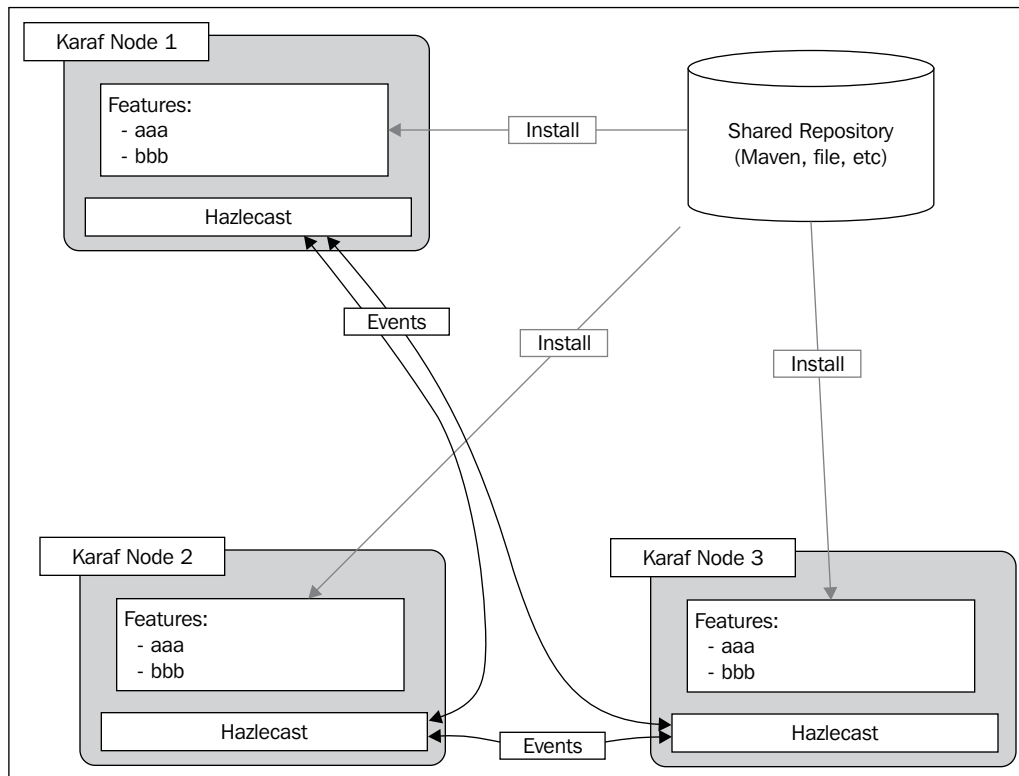
A node can be tested also using the `ping` command:

```
karaf@root> cluster:ping node2.local:5702
Pinging node :node2.local:5702
PING 1 node2.local:5702 82ms
PING 2 node2.local:5702 11ms
PING 3 node2.local:5702 14ms
```

Node discovery

Once you install Cellar on a Karaf instance, that instance automatically becomes a node in a cluster and will try to discover other nodes. It will do this using multicast or unicast provided by Hazelcast. The great thing about using Hazelcast for discovery is that there is no single point of failure in a multinode deployment. Each node has its own instance of Hazelcast and will receive event/changes from all other nodes in the topology. When a node is instructed to update/add/remove a configuration, feature, or bundle, it will check the shared repository for the change and update the local container accordingly.

At that point, the event is broadcast to the other nodes and they are instructed to make the same change using the same shared repository, as shown in the following diagram:

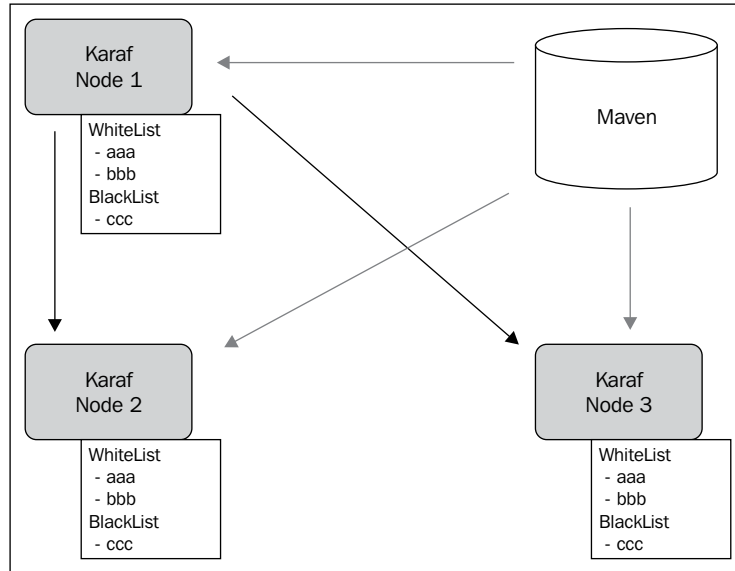


Let's look at a more specific example.

In the previous diagram, there are three nodes in the cluster that are all using a Maven-shared repository and have been grouped together in a cluster using Cellar. There are four main types of events that will be shared between the three nodes:

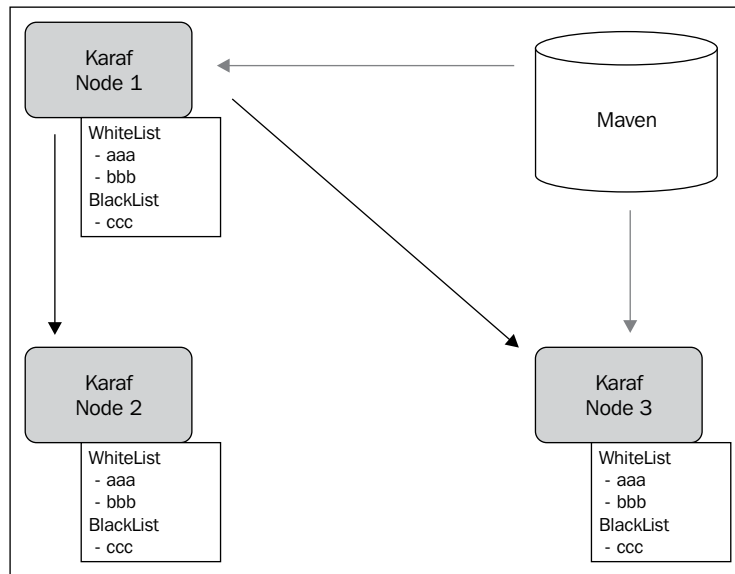
- Configuration changes
- Feature repository add/remove
- Feature install/uninstall
- Bundle install/uninstall

This means that anytime a feature repository is added to a node, all nodes now have the ability to install from that feature set. Once a node is tasked with one of the four types of events, that event is replicated to all other nodes in the cluster, as shown in the following diagram:



In the previous diagram, **Node 1** is configured to install `aaa`. **Node 1** will go to the shared repository and pull down the bundles for feature `aaa` and install them. After successfully installing and starting feature `aaa`, **Node 1** will broadcast an event to the other nodes in the cluster group (indicated by the black lines). Each node will then replicate the same actions of instances. **Node 2** and **Node 3** will also install the feature `aaa` from the shared repository (as indicated by red lines), so all nodes in the cluster are identical. This feature is now available locally, while the other systems are receiving the events and determining whether or not to install them.

Ok, but what if we do not want a node to get updated with a certain feature? Through the miracle of configuration, we can control what features a node is able to install. Using the `etc/org.apache.karaf.cellar.groups.cfg` file, a Karaf node can determine whether or not to install a feature when an event is received. A Karaf node can also decide not to broadcast a change event if a change is made locally.



If we tell **Node 1** to install `aaa`, that event is pushed to all other nodes and those nodes will check the configuration and install `aaa`. If **Node 1** installs `bbb`, all other nodes will receive the event, but only node 3 will install it. This is because **Node 2** has the feature `bbb` as part of its blacklist.

Looking at the default configuration file for Cellar groups in `etc/org.apache.karaf.cellar.groups.cfg`, we can see the inbound and outbound definitions. The inbound method references any events that are received by other nodes. The outbound method is the broadcasting of events to other nodes. So, we have the ability to control both in and out events. This helps minimize traffic between nodes. The following is an example of what a black and white list configuration would look like:

```
groups = default
```

```
default.config.whitelist.inbound = *
default.config.whitelist.outbound = *
default.config.blacklist.inbound = org.apache.felix.fileinstall*, \
    org.apache.karaf.cellar*, \
    org.apache.karaf.management, \
    org.apache.karaf.shell, \
    org.ops4j.pax.logging, \
```

```
    org.ops4j.pax.web
default.config.blacklist.outbound = org.apache.felix.fileinstall*, \
    org.apache.karaf.cellar*, \
    org.apache.karaf.management, \
    org.apache.karaf.shell, \
    org.ops4j.pax.logging, \
    org.ops4j.pax.web
default.config.sync = true

default.features.whitelist.inbound = *
default.features.whitelist.outbound = *
default.features.blacklist.inbound = config,management,hazelcast,cellar*
default.features.blacklist.outbound = config,management,hazelcast,cellar*
default.features.sync = true
default.features.repositories.sync = true

default.bundle.whitelist.inbound = *
default.bundle.whitelist.outbound = *
default.bundle.blacklist.inbound = none
default.bundle.blacklist.outbound = none
default.bundle.sync = true

default.obr.urls.sync = true
default.obr.bundles.sync = true
```

Using this configuration, we can turn on and off syncing for the main event types (feature repositories, features, bundles, and configurations). The default configuration is to sync everything. If you wish to turn off events for a specific type, just change the appropriate line to `false`. The following properties are examples of the synchronization flags set to `true`:

```
default.config.sync = true
default.features.sync = true
default.features.repositories.sync = true
default.bundle.sync = true
```

Notice the blacklists for the `config` events. These are good examples that show how to define blacklisted configurations in a comma-separated list. The same can be used to define whitelists:

```
default.config.blacklist.outbound = org.apache.felix.fileinstall*, \
    org.apache.karaf.cellar*, \
    org.apache.karaf.management, \
    org.apache.karaf.shell, \
    org.ops4j.pax.logging, \
    org.ops4j.pax.web
```

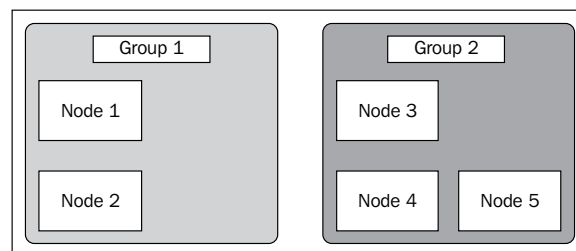
With the group configuration file, the node events can be controlled to minimize administration tasks and network traffic. Also, it is important to know that some configuration events should not be transmitted, like the ones that are blacklisted by default. Events that are fired by these configurations can cause these applications to fail to startup or startup incorrectly.



Avoid unnecessary network traffic and node event processing by setting outbound blacklists to avoid sending events that are not consumed by any other nodes.

Cluster groups

Cellar allows for grouping of nodes. A group is a working set of instances that are synchronized. This allows nodes to exist independently from a group. This can be useful when using Karaf instances in clusters that have different defined purposes. For example, you may have a set of Karaf instances clustered to provide web services, while having another group defined to provide backend business logic. These two groups would need to be administered separately:



The console provides the ability to manage the nodes in different groups via commands. `group-list` can be used to identify groups and the members associated with those groups:

```
karaf@root> cluster:group-list

Group           Members
* [default      ] [localhost:5701* ]
  [default      ] [node-2:5701* ]
```

Another way to view the node list is to use the command `cluster:node-list`:

```
karaf@root> cluster:node-list

ID                Host Name          Port
* [localhost:5701 ] [localhost        ] [ 5701]
```

Use `cluster:node-ping` to check if one node can ping another node in order to connect:

```
karaf@root> cluster:ping localhost:5701
Pinging node : localhost:5701
PING 1 localhost:5701 82ms
PING 2 localhost:5701 11ms
```

Now let's create a new group that will be the backend logic group:

```
karaf@root> cluster:group-create logic-group
```

After creating the group, you can list the groups again and see that you have made a new group with no nodes associated with it:

```
karaf@root> cluster:group-list

Group           Members
[logic-group     ] []
* [default      ] [localhost:5701* ]
```

Now that we have the new group, let's associate a node with it:

```
karaf@root> cluster:group-join logic-group

Group           Members
* [logic-group   ] [localhost:5701* ]
* [default      ] [localhost      :5701* ]
```

By using the command `cluster:group-join`, we have associated the local node with the new group.

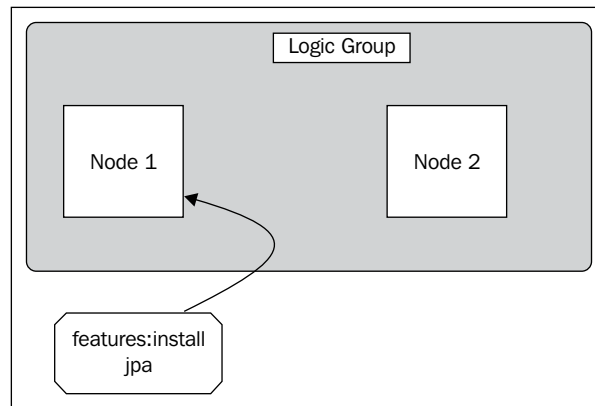


A node can be associated with more than one group; this makes it important to use blacklists in order to avoid exposing bundles to groups that may not want it.

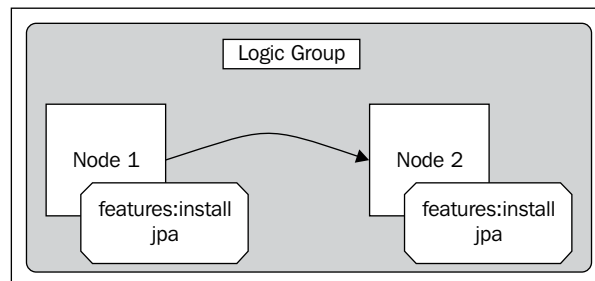
Syncing the nodes is simple at this point. If we install a new feature on the `localhost` node, we can see that it is also installed on the other nodes in the group. The same works for uninstalling features and configuration changes

Anytime a change is made to a cluster node, all other nodes in the cluster are updated accordingly. We install the `jpa` feature on **Node 1** using the following command:

```
karaf@root> cluster:features-install <group-name>jpa
```



Once this feature is installed, it will broadcast an event to a distributed topic that is listened to by the group, in this case **Logic Group**:



This will make it, so that any node that updates in the future will also get the changed events and automatically apply them. Now you can go to any node in the cluster group and see that the features are installed:

```
karaf@node-2> features:list | grep jpa
[installed ] [1.0.1] jpa          karaf-enterprise-2.3.1 OSGi Persistence
Container
```

Cloud discovery

There are a couple of optional components available from Cellar that make clustering more efficient and functional.

The cloud discovery service is available from most cloud providers that allow cloud storage. Cellar uses JCloud to provide a location to store the IP addresses of each node in the cluster. Hazelcast uses the file to find nodes based on the IP addresses listed. Basically, this process allows nodes to register themselves with a central file by placing its IP address there for other nodes to find.

In order to use it, two features need to be installed – Cellar-cloud and the JCloud feature—for the provider that is used (if using `aws` the feature would be `jcloud-aws-s3`):

```
features:install jclouds-<provider>
features:install cellar-cloud
```

Once the features are installed, a configuration file needs to be created in the `etc` directory for the provider. The file will need to follow the given naming convention: `org.apache.karaf.cellar.cloud-<provider>.cfg`.

This file will need to contain the following information:

```
provider=<blobstore provider>
identity=<the identity of the blobstore account>
credential=<the credential/password of the blobstore account>
container=<the name of the bucket>
validity=<the amount of time an entry is considered valid, after
that time the entry is removed>
```

Once this file is created, the service will check for new nodes, update Hazelcast, and restart the instance to incorporate all the changes.

Summary

Clustering Karaf is an important part to enterprise deployments. Cellar brings this capability to Karaf along with the ease of administration. As you have learned in this chapter, there are many different ways to architect a cluster. It is up to you how you want the clustering to affect the different levels of functionality in your deployment architecture. With Cellar, there are no limitations to how you can deploy it. *With great power, comes great responsibility.*

8

Our Final Programming Project

With the help of all the things we've learned so far about Apache Karaf, we will develop a complete application. With this application we will touch upon:

- A Maven build
- Java and OSGi code
- Apache Aries Blueprint
- Extending Apache Karaf's command system
- Deployment descriptors and features

Our application

For this chapter's application code, we will build a set of commands that will allow you to monitor events utilizing the OSGi Compendium specifications.

The JavaDocs for these specifications can be found at <http://www.osgi.org/javadoc/r4v43/cmpn/>.

We will utilize the `EventHandler` of `EventAdmin` to listen on topics for messages sent through the OSGi runtime. The following is from Javadocs:

EventHandler objects are registered with the Framework service registry and are notified with an Event object when an event is sent or posted.

EventHandler objects can inspect the received Event object to determine its topic and properties.

EventHandler objects must be registered with a service property `EventConstants.EVENT_TOPIC` whose value is the list of topics in which the event handler is interested.

The application we will build is going to allow us to add new command features to the Karaf command-line interface:

- `eventhandler:add`
- `eventhandler:remove`
- `eventhandler:list`

These three commands will allow us to register for events with the framework service registry; we will display the events on `System.out` as they arrive. In some ways this is a very rudimentary debugging tool for Apache Karaf.

A Maven build

Like so many other things in Apache and Java land, we'll start this application with a Maven build. The full developer documentation for building Karaf commands can be found at <http://karaf.apache.org/manual/latest/developers-guide/extending-console.html>.

We are utilizing a readymade `pom.xml` file in favor of utilizing an archetype; it is left to the reader to pick one method over the other. We start our `pom` file by defining the project metadata:

```
<groupId>org.packt.learning.osgi.code</groupId>
<artifactId>learning-karaf-eventhandler</artifactId>
<version>1.0-SNAPSHOT</version>
<name>Learning Karaf ::: learning-karaf-eventhandler</name>
<description>Learning Karaf ::: learning-karaf-eventhandler</
description>
<packaging>bundle</packaging>

<properties>
```

```
<karaf.version>2.3.2</karaf.version>
<osgi.version>4.3.1</osgi.version>
<gogo.version>0.10.0</gogo.version>
</properties>
```

We are initially moving out the versions of dependencies into properties, setting up descriptive names, and declaring this project to be a 1.0-SNAPSHOT version, as it is our first development effort. Utilizing properties and moving all the version information into one place allows us to effectively declare our dependencies and upgrade versions at one place in a file. For a project this small, using all Maven's features may not be necessary, but it is a good habit to get into.

Since we are building this application to run in an OSGi environment, the next important step in our build is to incorporate the correct bundle information as well. We do this utilizing the Apache Felix Maven bundle plugin. It is a Maven plugin that will help us to add a correctly formatted `MANIFEST.MF` file to our JAR file, declaring it a bundle. We will configure the plugin in the following way:

```
<plugin>
  <groupId>org.apache.felix</groupId>
  <artifactId>maven-bundle-plugin</artifactId>
  <extensions>true</extensions>
  <configuration>
    <instructions>
      <Export-Package>
        com.packt.learning.osgi.command
      </Export-Package>
      <Import-Package>
        org.apache.felix.service.command,
        org.apache.felix.gogo.commands,
        org.apache.karaf.shell.console,
        com.packt.learning.osgi.command,
        *
      </Import-Package>
    </instructions>
  </configuration>
</plugin>
```

Now, with this configuration, we have told Maven that when we are building this JAR file, we want its packaging form to be of type bundle. The bundle plugin is called during build and writes out a `MANIFEST.MF` file for our JAR.

The complete plugin documentation can be found at <http://felix.apache.org/site/apache-felix-maven-bundle-plugin-bnd.html>.

We use another interesting plugin to make sure our `features` file gets deployed into the repository – making it accessible as a normal Maven artifact that can be referenced by Karaf when we get to the deployment stage:

```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>build-helper-maven-plugin</artifactId>
  <executions>
    <execution>
      <id>attach-artifacts</id>
      <phase>package</phase>
      <goals>
        <goal>attach-artifact</goal>
      </goals>
      <configuration>
        <artifacts>
          <artifact>
            <file>target/classes/features.xml</file>
            <type>xml</type>
            <classifier>features</classifier>
          </artifact>
        </artifacts>
      </configuration>
    </execution>
  </executions>
</plugin>
```

Java and OSGi code

With our dependencies and the basic project infrastructure declared, we can move on to actually building the application. Karaf provides a wonderful set of tools and annotations for building commands that perform actions in the shell. A command is a simple Java class that extends `OsgiCommandSupport` and has the right annotations for presentation and injection of fields into the methods you need to implement.

The `eventhandler:add` command can be implemented as follows:

```
@Command(scope = "eventhandler", name = "add", description =
    "Adds an event listener.")
public class AddEventHandler extends OsgiCommandSupport {

    @Argument(index = 0, name = "filter",
        description = "The event topic to listen to (*,
            org/apache/karaf, org/apache/karaf/*, org/apache/karaf/log,"
```

```

        + "org.apache/karaf/log2) only one handler per topic will
        be created. The filter is space separated",
        required = true, multiValued = false)
    String filter;
    EventHandlerRepository repository;

    @Override
    protected Object doExecute() throws Exception {
        repository.addEvent(filter);
        return null;
    }

```

The `@Command` and `@Argument` annotations here are what is of interest. They'll establish that this is a Karaf command and provides details on how it is called. They will also parse out the command-line text added after the command into a field we can use in our class.

The method we need to implement in our command class is `doExecute()`, a method Karaf will call for us with our argument set. Our simple command just registers a listener for events via what we call an `EventHandler` registry, so we simply return a null value in this implementation.

The main class that does the majority of the legwork is called `EventHandlerRepository`. It is where we store all instances of `ServiceReference`, add new listeners, remove them, and keep track of state when invoked by services:

```

public class EventHandlerRepository implements BundleContextAware {
    private Map<String, Registry> eventHandlers = new HashMap<String,
Registry>();
    protected BundleContext bundleContext;
    protected List<ServiceReference> usedReferences;

    public synchronized void addEvent(String filter) {
        if (!eventHandlers.containsKey(filter)) {
            EventHandler handler = new EventDisplayer();
            Dictionary<String, String> properties = new Hashtable<String,
String>();
            properties.put(EventConstants.EVENT_TOPIC, filter);
            ServiceRegistration registration = getBundleContext().
registerService(EventHandler.class.getName(), handler, properties);
            Registry registry = new Registry(handler, registration);
            eventHandlers.put(filter, registry);
        }
    }

    public synchronized void removeEvent(String filter) {

```

```
if (eventHandlers.containsKey(filter)) {
    Registry registry = eventHandlers.get(filter);
    registry.getRegistration().unregister();
    EventHandler handler = registry.getHandler();
    handler = null;
    registry = null;
    eventHandlers.remove(filter);
}
```

The `addEvent` method will take a filter; a filter is a topic we wish to listen to for event notifications from the `EventAdmin` service. The wildcard (*) allows us to listen to every single event passed through the system. Other filters would look like this: `com/packt/learning/karaf/*`—a filter that would register for updates on anything passed with `com/packt/learning/karaf` and whatever follows this prefix.

The `EventHandlerRegistry` class relies on access to the `OSGi BundleContext` class to register and remove services, making this simple implementation class a part of the OSGi runtime and its lifecycle as well as updates and events. We utilize the fact that we are running this in a Blueprint container to get a hold of the bundle context; other possible ways of doing this would be to implement a `BundleActivator` class or using `FrameworkUtils` to force a lookup inside the container.

Apache Aries Blueprint

With our Java code snippets complete, we need to run this somehow. The currently supported way of accomplishing this in Apache Karaf is via Aries Blueprint. Blueprint provides a dependency injection framework for OSGi and was standardized by the OSGi Alliance in OSGi Compendium R4.2. It is designed to deal with the dynamic nature of OSGi, where services can become available and unavailable at any time. The specification is also designed to work with **plain old Java objects (POJOs)**, enabling simple components to be written and unit-tested in a JSE environment without needing to know of how they are assembled. The Blueprint XML files that define and describe the assembly of various components are key to the Blueprint programming model. The specification describes how the components get instantiated and wired together to form a running module. In simple terms, Blueprint is a standardized form of a DI framework, quite similar to Spring. It should be quite easy for people familiar with the Spring framework to convert existing code to Blueprint as well.

Our deployment Blueprint file is as follows:

```
<blueprint
  xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
```

```

    default-activation="lazy">

    <command-bundle
      xmlns="http://karaf.apache.org/xmlns/shell/v1.1.0">
        <command name="eventhandler/list">
          <action class="com.packt.learning.osgi.command.
ListEventHandler">
            <property name="repository" ref="repository"/>

          </action>
        </command>

        <command name="eventhandler/add">
          <action class="com.packt.learning.osgi.command.
AddEventHandler">
            <property name="repository" ref="repository"/>

          </action>
        </command>

        <command name="eventhandler/remove">
          <action class="com.packt.learning.osgi.command.
RemoveEventHandler">
            <property name="repository" ref="repository"/>

          </action>
        </command>

      </command-bundle>

      <bean id="repository" class="com.packt.learning.osgi.command.
EventHandlerRepository" destroy-method="cleanup">
        <property name="bundleContext" ref="blueprintBundleContext"/>
      </bean>

    </blueprint>

```

In our Blueprint file, we declare all three of our annotated commands and name them in a descriptive manner. The XML tags for the command bundle contain a namespace declaration. This extra namespace tells the Blueprint container that the building of a "recipe" (bean to instantiate) will be handled by an external provider. These external providers are called namespace handlers. There are several namespace handlers written for Aries Blueprint to facilitate things such as building Camel Routes and configuring CXF endpoints. All they do is standardize and add an extra XML schema definition that can be reused for generic tasks, minimizing redundant code.

The last bean we declare is our `EventHandler` repository, that is, our `Singletonish` class, which will live with the bundle and hold all the references to instantiated services and their listener classes. We declare a `destroy` method on this bean to make sure we have nothing hanging around if this bundle is stopped or restarted. The `destroy` and `init` methods are comparable to Spring's `InitializingBean` and `DisposableBean` interfaces. They are the methods called right after all properties are set and right before disposing of the bean and shutting down the Blueprint container.

Extending Apache Karaf's command system

With the annotations in Java, as well as by letting our command classes extend `OsgiCommandSupport`, we have utilized the Apache Karaf command system and introduced new commands. We have added an `EventHandler` service with just a few simple lines of code, and now we can utilize this extra functionality in any version-compatible Karaf installation. A full walk-through of how Karaf commands can be extended is located at <http://karaf.apache.org/manual/latest-2.2.x/developers-guide/extending-console.html>.

The document describing commands is located at <http://felix.apache.org/site/rfc-147-overview.html>.

These resources will explain the history and rationale of how OSGi console commands are implemented.

The last bit of polishing we'll do to our project is by adding some metadata for the help display, which is located in the file `bundle.info` in the `src/main/resources` folder:

```
\u001B[1mSYNOPSIS\u001B[0m
    ${project.name}

    ${project.description}

    Maven URL:
    \u001B[33mmvn:${project.groupId}/${project.
    artifactId}/${project.version}\u001B[0m
```

This allows us to put in a neat "man page"-like documentation snippet. This resource file will be incorporated into the bundle and the command help system will display this information when a help is requested. The help can be displayed by issuing:

```
eventhandler:add -help
```

As you can see, adding commands that control the functionality in Karaf is a very simple and effective way of extending and modifying the console for specific customization. It may be for the purpose of inventory, debugging, or simply controlling features of newly developed code.

Deployment descriptors and features

With Maven set up, code written, and Blueprint deployment descriptors ready to go, we can deploy this bundle by simply putting in the `deploy` directory in Apache Karaf. Karaf will use Apache Felix FileInstall and monitor this directory for changes. Any new artifacts located here will go through a resolution process to find out if it is a file, spring XML file, Blueprint file, JAR, bundle and so on. Quite a few formats are supported and more can be added. By default, we deploy bundles in an OSGi environment. We also know that this bundle will not work directly in a freshly downloaded Apache Karaf, as we are using extended features. If you drop this bundle in `deploy`, it will not run until you have issued the following command on the console:

```
features:install eventadmin
```

This will install all the `EventAdmin` facilities for us and will allow our bundle to resolve. While doing this manual setup might be okay for local testing and development purposes, it certainly isn't the way you want to handle automated deployments and container startups. To automate this, we will use the same mechanism that Karaf uses for its own "features": the features service. As previously stated, the XSD for features is available at <http://karaf.apache.org/xmlns/features/v1.0.0>. Registering this schema with your development tools is highly advisable.

Our `features.xml` file is located under `src/main/resources` in our project.

It looks like this:

```
<features name="com.packt.learning.osgi-${project.version}">
  <feature name="eventhandler-command"
    version="${project.version}">
    <feature version="${karaf.version}">eventadmin</feature>
    <bundle>mvn:org.packt.learning.osgi.code/learning-karaf-
      eventhandler/${project.version}</bundle>
    </feature>
  </features>
```

It is a short file we don't have much to deploy, but it already illustrates what was pointed out earlier. We can combine features, bundles, and so on into a larger deployment descriptor, allowing us to iteratively build large and complex deployments.

Our feature here is named and versioned. Once registered in Karaf, we will be able to issue the following command, and Karaf will utilize Pax URL and its Maven support to get the artifacts for us to install and activate. The first thing that will happen is that the `features` service will check to see whether Karaf's `eventadmin` is installed; if not, that'll be done for us prior to the installation of our command bundle:

```
features:install eventadmin
```

This is all we need to invoke to start our bundle now, making it a part of the existing command suite in the Karaf command-line interface.

Since this is now a deployable feature, we can make it part of Karaf's startup procedure as well by adding it to the configuration file `org.apache.karaf.features.cfg` in the `etc` directory. This file allows us to declare `features` repositories as well as a `featuresBoot` setting, where our new command could be declared.

We will go ahead and add our `features` file to the features in Karaf, utilizing the `addurl` command:

```
features:addurl mvn:org.packt.learning.osgi.code/learning-karaf-  
eventhandler/1.0-SNAPSHOT/xml/features
```

If all goes well, you simply get a new line on the console. We can then reissue the `features:listurl` command to see that we actually added in the right descriptor:

```
karaf@root> features:listurl  
  
Loaded    URI  
true      mvn:org.packt.learning.osgi.code/learning-karaf-  
eventhandler/1.0-SNAPSHOT/xml/features  
true      mvn:org.apache.karaf.assemblies.features/enterprise/2.3.2/xml/  
features  
true      mvn:org.apache.karaf.assemblies.features/standard/2.3.2/xml/  
features
```

If we issue a `features:list` command, we can see that our newly added feature is uninstalled and available:

State	Version	Name	
[uninstalled]	[1.0-SNAPSHOT]	eventhandler-command	com.packt. learning.osgi-1.0-SNAPSHOT

We now have a fully functional deployment model!

Summary

In this chapter we learned how to set up a bootstrap Maven OSGi project, utilizing the Apache Karaf documentation to extend the console. We added code and Apache Karaf-specific command code in Java that does the work for us. In total, we added three different commands that instantiate, remove, and list service consumers.

We declared a Blueprint context for the bundle, allowing us to treat it as a command bundle in Apache Karaf, thanks to the already implemented namespace handler support.

We added a bean into the mix to keep track of the services we are utilizing and the routines we need to clean up after us.

We explained what is necessary to extend the Apache Karaf command system and where it originated. We also added some nice help page displays to our project.

Lastly, we dug into how features are used, installed, and how to write and deploy our own features.

The complete source code for this chapter is available at <https://github.com/seijoed/learning-karaf>.

Apache Karaf Commands

In this reference chapter we will outline all the core Karaf commands for a reference set. The commands listed here are up-to-date until the time of writing. Karaf being continuously evolving software, I presume there would be few more commands that might get released after the book is published. Karaf commands are grouped by scope; there are several scopes of interest for the new Karaf developer as well as for Karaf administrators.

For a developer, the `osgi:*`, `dev:*`, and `packages:*` commands will be of particular interest as they contain several valuable tools for monitoring and debugging bundle lifecycles.

For the administrator, the `features:*`, `log:*`, `osgi:*`, `admin:*`, and `config:*` commands will be where the majority of work is performed. The following table outlines all the basic commands in Apache Karaf; there are several extensions to these developed in other open source projects such as Apache Camel and Apache ActiveMQ. These commands can be deployed into Karaf and will be just like our example in *Chapter 8, Our Final Programming Project*. They extend Karaf with new functionalities and are listed as follows:

- As the scope indicates, the `admin` commands allow you to control Karaf from a system administration perspective, create new containers, and manage instances:

Command	Description
<code>admin:change-opts</code>	Changes options of container
<code>admin:change-rmi-registry-port</code>	Changes the RMI registry port
<code>admin:change-rmi-server-port</code>	Changes the RMI server port
<code>admin:change-ssh-port</code>	Changes the secure shell port
<code>admin:clone</code>	Clones a container instance

Command	Description
admin:connect	Connects to a container
admin:create	Creates a new container
admin:destroy	Destroys a container instance
admin:list	Lists container instances
admin:rename	Renames a container instance
admin:start	Starts a container instance
admin:stop	Stops a container instance

- The configuration commands allow you to modify and control configurations; the same configurations can also be controlled via configuration files for persistence:

Command	Description
config:cancel	Cancels the changes in progress
config:delete	Deletes a configuration
config:edit	Creates or edits a configuration
config:list	Lists existing configurations
config:propappend	Appends a value to a property
config:propdel	Deletes a property
config:proplist	Lists properties
config:propset	Sets a property
config:update	Saves and propagates

- dev commands are not necessarily of much use in a production environment but are quite valuable during development and testing phases:

Command	Description
dev:create-dump	Creates an archive with info
dev:dynamic-import	Enables/disables dynamic-import
dev:framework	OSGi Framework options
dev:print-stack-t	Prints the full stack traces
dev:restart	Restarts Karaf

Command	Description
<code>dev:show-tree</code>	Shows the tree of bundles based on the wiring information
<code>dev:system-property</code>	Gets or sets a system property
<code>dev:wait-for-service</code>	Waits for a given OSGi service
<code>dev:watch</code>	Watches and updates bundles

- features commands control deployment and concealment and prove to be an indispensable tool for system administrators.

Command	Description
<code>features:addurl</code>	Adds a list of repository URLs
<code>features:chooseurl</code>	Adds a repository url for well known features
<code>features:install</code>	Installs a feature
<code>features:info</code>	Shows feature information
<code>features:listrepositories</code>	Lists all repositories
<code>features:list</code>	Lists all existing features
<code>features:listversions</code>	Lists all versions of a feature
<code>features:listurl</code>	Lists all defined URLs
<code>features:removerepository</code>	Removes a repository
<code>features:refreshurl</code>	Reloads the list of available features from the repositories
<code>features:uninstall</code>	Uninstalls a feature with the specified name and version
<code>features:removeurl</code>	Removes the given list of repository URLs from the features service

- We can invoke the the help system at any point of time; all commands have a `-help` option for descriptions:

Command	Description
<code>help</code>	Displays generic help.

- **Java Autentication and Authorization Services (JAAS)** allows you to control realms, logins, users, and roles settings:

Command	Description
jaas:cancel	Cancels modification on a Realm
jaas:pending	Lists the modifications on the selected JAAS Realm/Login Module
jaas:manage	Manages users and roles
jaas:roleadd	Adds a role to a user
jaas:realms	Lists JAAS Realms
jaas:update	Updates the selected JAAS Realm
jaas:roledel	Deletes a role from a user
jaas:userdel	Deletes a user
jaas:useradd	Adds a user
jaas:users	Lists the users of the selected JAAS Realm/Login Module

- The `log` commands allow you to control logging, follow and display logs, and change logging level in running systems:

Command	Description
log:display-exception	Displays the last exception
log:display	Displays log entries
log:set	Sets the log level
log:get	Shows the log level
log:tail	Continuously displays log entries

- The `osgi` commands are usually the most frequently used ones; they give you full control over a bundle lifecycle and allow you to deploy, start, stop, and suspend bundles:

Command	Description
osgi:bundle-level	Gets or sets the start level of a given bundle
osgi:classes	Displays classes in bundles
osgi:bundle-services	Lists OSGi services per bundle
osgi:headers	Lists headers of a given bundle
osgi:find-class	Locates a specified class

Command	Description
<code>osgi:install</code>	Installs one or more bundles
<code>osgi:info</code>	Displays detailed information
<code>osgi:ls</code>	Lists OSGi services
<code>osgi:list</code>	Lists all installed bundles
<code>osgi:refresh</code>	Refreshes a bundle
<code>osgi:name</code>	Shows or change instance name
<code>osgi:restart</code>	Stops and restarts bundle(s)
<code>osgi:resolve</code>	Resolves bundle(s)
<code>osgi:start</code>	Starts bundle(s)
<code>osgi:shutdown</code>	Shuts down the framework down
<code>osgi:stop</code>	Stops bundle(s)
<code>osgi:start-level</code>	Gets or sets the system start level
<code>osgi:update</code>	Updates bundle
<code>osgi:uninstall</code>	Uninstalls bundle(s)
<code>osgi:version</code>	Displays the instance version

- The `packages` commands provide a convenient way of showing packages and how they are exported and imported. They are very helpful during deployment testing:

Command	Description
<code>packages:exports</code>	Displays exported packages
<code>packages:imports</code>	Displays imported packages

- Shell commands provide you with convenient, Unix-like commands; you can alias, execute code, list files, and source scripts:

Command	Description
<code>shell:alias</code>	Creates an alias to a command.
<code>shell:clear</code>	Clears the console buffer.
<code>shell:cat</code>	Displays a file or URL.
<code>shell:each</code>	Displays the current time in the given FORMAT
<code>shell:exec</code>	Executes system processes.
<code>shell:echo</code>	Echoes or prints arguments to STDOUT.
<code>shell:head</code>	The first lines of a file.

Command	Description
shell:grep	Prints lines matching a pattern.
shell:if	If/Then/Else block.
shell:history	Prints command history.
shell:java	Executes a Java standard application.
shell:info	Prints system information.
shell:more	File pager.
shell:logout	Disconnects shell from current session.
shell:printf	Formats and prints arguments.
shell:new	Creates a new java object.
shell:sort	Writes sorted concatenation of all files to standard output.
shell:sleep	Sleeps for a bit then wakes up.
shell:tac	Captures the STDIN and returns it as a string. Optionally, it writes the content to a file.
shell:source	Runs a script.
shell:watch	Watches and refreshes the output of a command.
shell:tail	Displays the last lines of a file.
shell:wc	Prints newline, word, and byte counts for each file.

- And finally the `ssh` commands allow you to get connected externally as well as create local access facilities:

Command	Description
ssh:ssh	Connects to a remote SSH server
ssh:sshd	Creates an SSH server

Index

Symbols

-n parameter 28

-p parameter 28

A

addEvent method 94

addurl command 98

admin commands 101

Apache Aries Blueprint

Apache Karaf's command system, extending 96

deployment descriptors 97, 98

features 97, 98

Apache Cellar 79, 80

Apache Felix Bundle Plugin 48

Apache Karaf

booting 11, 12

commands 13, 14

features 42-45

installing 10, 11

system requisites 7-9

Apache Karaf 2.3.x

versus Apache Karaf 3.0.x 10

Apache Karaf 3.0.x

versus Apache Karaf 2.3.x 10

Apache Karaf client 17

Apache Karaf Commands

admin commands 101

configuration commands 102

dev commands 102

features commands 103

JAAS 104

log commands 104

osgi commands 104

packages commands 105

Shell commands 105

ssh commands 106

URL 90

Apache Karaf distribution

obtaining 9, 10

URL 9

Apache Maven

repositories 37-40

system repository 41, 42

URL 37

Apache Tomcat

URL 60

application logging

improving 70-72

Applications 42

B

bin folder 11

Blueprint

deploying 61

build element 69

bundle

about 47

building 48, 49

deploying 47, 48

deploying, file handler used 50

deploying, hot deployment used 50, 51

deploying, HTTP used 50

deploying, Maven used 49

BundleActivator class 94

Bundle-Level command 15

bundle life-cycle

states 14

C

- camel-spring feature 44
- cat command 16
- child instances 75, 76
- chkconfig command 73
- Cloud-based feature 88
- cluster command 79
- cluster groups 85, 87
- command
 - creating 17
 - Karaf-command-archetype 18
- command class 93
- CommandCompleter class 20
- command element 19
- configuration commands 102
- console
 - logging 28, 29
- console configuration commands 31, 32
- Create-dump command 16

D

- data folder 11
- demos folder 11
- deploy folder 11
- deployment descriptors 97, 98
- descriptors configuration element 69
- destroy method 96
- dev commands 102
- doExecute() method 19, 93
- Dynamic-import command 16

E

- environment variables
 - setting 31
- etc folder 11
- eventhandler:add command 92
- EventHandlerRegistry class 94
- EventHandlerRepository 93
- Event object 90
- Exports command 15

F

- failover
 - configuring 34, 35

- feature descriptors
 - deploying 52-56
- features 42
- features:list command 53, 98
- features:listurl command 98
- featuresBoot 36
- features commands 103
- features configuration element 69
- featuresRepositories 36
- felix.fileinstall.poll property 31
- file
 - logging 27
- file element 56
- file handler
 - used, for deploying bundle 50
- Framework command 15

G

- grep command 16

H

- headers command 49
- Headers command 15
- help command 19
- hostKey 26
- hot deployment
 - configuring 31
 - used, for deploying bundle 50, 51
- HTTP
 - used, for deploying bundle 50

I

- Imports command 15
- Info command 15
- install command 14, 50, 79
- instances folder 11

J

- JAAS 76, 104
- Java Autentication and Authorization Services. *See* JAAS
- Java code 92-94
- JavaDocs
 - URL 89

JAVA_HOME Environment Variable
setting up 8
Java Management Extension. *See* **JMX console**
JMX access
locking down 78
JMX console 21

K

KAR
creating 62-64
deploying 62-64
Karaf
command project 18-21
configuring 29, 30
installing, as service 72, 73
Karaf Archive. *See* **KAR**
Karaf-command-archetype 18

L

lib folder 11
list command 15, 32, 50, 58
log:get command 28
log commands 104
logging properties
about 26
console, logging 28, 29
file, logging 27

M

master-slave failover 73, 74
Maven
building 90, 91
bundle plugin, URL 91
used, for deploying bundle 49

N

node discovery 80-85
non-OSGi JARs
deploying 56-59

O

offline repository
about 67, 68
building 68-70
osgi:list command 58
OSGi Bundle Repository (OBR) 68
OSGi code 92-94
osgi commands 104
outbound method 83

P

packages commands 105
password encryption 77
ping command 80
plain old Java objects (POJOs) 94
Print-stack-traces command 16
project
contents 18-21

R

reboot -c command 30
Refresh command
about 15
versus Update command 15
remote access
about 25, 26
remote console access
Apache Karaf client 16
Resolve command 15
Restart command 16
roles
managing 76, 77
password encryption 77

S

security configuration
about 76
JMX access, locking down 78
roles, managing 76, 77
shade plugin 58
Shell commands 105
Show-tree command 16

shutdown command 16, 17

SingleTonIsh class 96

Spring

deploying 61

spring-dm feature 43

ssh commands 106

sshHost 26

sshIdleTimeout 26

sshPort 26

sshRealm 26

start command 14, 73

Start-Level command 15

startup properties

about 35, 36

remote access 25, 26

startup.properties 41

stop command 14

system folder 11

system properties

about 29

console configuration commands 31, 32

environment variables, setting 31

failover configuration 34, 35

hot deployment, configuring 31

Karaf, configuring 29, 30

web console 33, 34

system repository 41, 42

T

tail command 16

U

Uninstall command 14

Update command

about 15

versus Refresh command 15

url handler command 49

W

WAR

deploying 60, 61

Watch command 16

web console

about 2, 33, 34

installing 22

starting 22

URL 23

Windows System Environment Variables

managing 8

X

XSD

URL 97



Thank you for buying Learning Apache Karaf

About Packt Publishing

Packt, pronounced 'packed', published its first book "Mastering phpMyAdmin for Effective MySQL Management" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: www.packtpub.com.

About Packt Enterprise

In 2010, Packt launched two new brands, Packt Enterprise and Packt Open Source, in order to continue its focus on specialization. This book is part of the Packt Enterprise brand, home to books published on enterprise software – software created by major vendors, including (but not limited to) IBM, Microsoft and Oracle, often for use in other corporations. Its titles will offer information relevant to a range of users of this software, including administrators, developers, architects, and end users.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.



OSGi Starter

ISBN: 978-1-84951-992-2

Paperback: 58 pages

The essential guide to modular development with OSGi

1. Learn something new in an Instant! A short, fast, focused guide delivering immediate results.
2. Learn what can be done with OSGi and what it can bring to your development structure
3. Build your first application and deploy to an OSGi runtime that simplifies your experience
4. Discover an uncomplicated, conversational approach to learning OSGi for building and deploying modular applications



Apache ServiceMix How-to

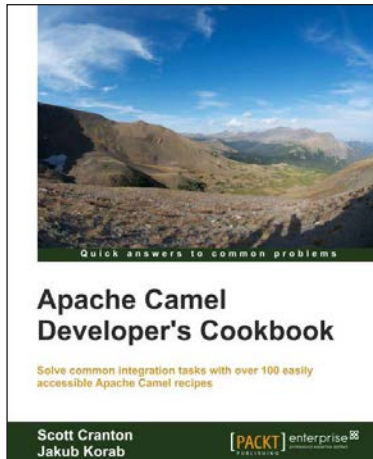
ISBN: 978-1-84951-966-3

Paperback: 66 pages

Learn to create simple ServiceMix-based integration solutions using short, practical, hands-on recipes

1. Learn something new in an Instant! A short, fast, focused guide delivering immediate results.
2. Leverage OSGi to speed up the ESB deployment
3. Define message flow with Camel DSL
4. Expose your system via web services

Please check www.PacktPub.com for information on our titles



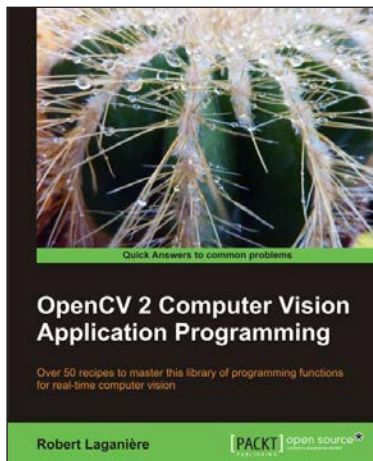
Apache Camel Developer's Cookbook

ISBN: 978-1-78217-030-3

Paperback: 369 pages

Solve common intergration tasks with over 100 easily accessible Apache Camel recipes

1. A practical guide to using Apache Camel delivered in dozens of small, useful recipes
2. Written in a Cookbook format that allows you to quickly look up the features you need, delivering the most important steps to perform with a brief follow-on explanation of what's happening under the covers
3. The recipes cover the full range of Apache Camel usage from creating initial integrations, transformations and routing, debugging, monitoring, security, and more



OpenCV 2 Computer Vision Application Programming Cookbook

ISBN: 978-1-84951-324-1

Paperback: 304 pages

Over 50 recipes to master this library of programming functions for real-time computer vision

1. Teaches you how to program computer vision applications in C++ using the different features of the OpenCV library
2. Demonstrates the important structures and functions of OpenCV in detail with complete working examples
3. Describes fundamental concepts in computer vision and image processing

Please check www.PacktPub.com for information on our titles