JBoss AS 7 Configuration, Deployment, and Administration

Build a fully-functional, efficient application server using JBoss AS

**Francesco Marchioni**

BIRMINGHAM - MUMBAI

JBoss AS 7 Configuration, Deployment,

and Administration

Copyright © 2011 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 author, 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: December 2011

Production Reference: 1091211

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-84951-678-5

www.packtpub.com

Cover Image by Charwak A (charwak86@gmail.com)

**Author**

Credits

**Project Coordinator**

Francesco Marchioni

**Reviewers**

Jaikiran Pai

Deepak Vohra

**Acquisition Editor** Sarah Cullington

**Development Editors** Dayan Hyames

Reshma Sundaresan

**Technical Editors** Lubna Shaikh

Mohd. Sahil

**Indexers**

Hemangini Bari

Tejal Daruwale

Monica Ajmera Mehta

Jovita Pinto

**Proofreaders**

Jonathan Todd

Lesley Harrison

**Graphics**

Manu Joseph

**Production Coordinator** Shantanu Zagade

**Cover Work**

Shantanu Zagade

About the Author

**Francesco Marchioni** is a Sun Certified Enterprise Architect employed by an Italian company based in Rome. He started learning Java in 1997, and since then he has followed the path to the newest Application Program Interfaces released by Sun. He joined the JBoss Community in 2000, when the application server was running the release 2.X.

He has spent many years as a software consultant, where he has overseen many successful software migrations from vendor platforms to open source products such as JBoss AS, fulfilling the tight budget requirements of current times.

Over the last five years, he started authoring technical articles for O'Reilly Media and running an IT portal focused on JBoss products (http://www.mastertheboss.com).

In December 2009, he published the title "JBoss AS 5 Development", which describes how to create and deploy Java Enterprise applications on JBoss AS: http://www.packtpub.com/jboss-as-5-development/book.

In December 2010, he published his second title, "JBoss AS 5 Performance Tuning", which describes how to deliver fast and efficient applications on JBoss AS: http://www.packtpub.com/jboss-5-performance-tuning/book

I need to thank a lot of people for this book. First, I owe my hearty thanks to the JBoss community which helped me through this

journey in developing a completely new product. In particular, I'd like to thank Jaikiran for sharing his experience as a reviewer of the book, improving its quality substantially.

I'd like also to express my gratitude to Packt Publishing team that shared with me the challenge to write a book which was already rewritten many times as application server changed its skin.

Last but not least, I'd like to thanks my wife Linda, who has kept doing housework patiently while I was writing the book, just asking when that damned JBoss thing will end.

About the Reviewers

**Jaikiran Pai** works at Red Hat and is part of the JBoss AS and EJB3 development team. In his role as a software developer, Jaikiran has been mainly involved in Java language and Java EE technologies. He completed his graduation in 2004 and started working in a software company in Pune, India. During this period, he developed an interest in JBoss Application Server and has been active in the JBoss community ever since. Subsequently, he joined Red Hat to be part of the JBoss EJB3 team.

Jaikiran is one of the co-authors of JBoss AS Getting Started DZone RefCard (http://refcardz.dzone.com/refcardz/getting-started-jboss). When he's not doing anything JBoss related, you'll find him at his other favorite and friendly place - JavaRanch (http://www.javaranch.com), where he has been a moderator since 2007.

**Deepak Vohra** is a consultant and a principal member of the NuBean.com software company. Deepak is a Sun Certified Java Programmer and Web Component Developer and has worked in the fields of XML and Java programming and J2EE for over five years. Deepak is the co-author of the Apress book Pro XML Development with Java Technology, and he was the technical reviewer for the O'Reilly book WebLogic: The Definitive Guide. Deepak was also the technical reviewer for the Course Technology PTR book Ruby Programming for the Absolute Beginner and the technical editor for the Manning Publications book Prototype and Scriptaculous in Action. Deepak is also the author of the Packt Publishing books JDBC 4.0 and Oracle JDeveloper for J2EE Development, Processing XML Documents with Oracle JDeveloper 11g, and EJB 3.0 Database Persistence with Oracle Fusion Middleware 11g.

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?**

⓿ Fully searchable across every book published by Packt

⓿ Copy & paste, print and bookmark content

⓿ 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.

*This book is dedicated to my family who have always been there for me and in particular to my child Alessandro who's only four but wants to be an actor in theater. Never doubt about your dreams Alessandro, no matter how crazy they might be.*

*—***Francesco Marchioni**

Table of Contents

**Preface 1 Chapter 1: Installing JBoss AS 7 7 Getting started with the application server 9** Installing the Java environment 9 Installing JBoss AS 7 12 Starting up JBoss AS 13 Connecting to the server with the command line interface 14 Stopping JBoss 15 Locating the shutdown script 16 Stopping JBoss on a remote machine 16 Restarting JBoss 16 Installing Eclipse environment 16 Installing JBoss tools 17 **Exploring the application server file system 18** The bin folder 20 The docs folder 20 The domain folder 21 The standalone folder 21 The welcome-content folder: 22 The modules folder 22 Loading application server modules 24 **Summary 26 Chapter 2: Configuring the Application Server 27 Configuring the application server 27** Extensions 29 Paths 29 Management interfaces 30 Profiles 31 Interfaces 31

*Table of Contents*

Socket binding groups 32 System properties 33 Deployments 34

**Configuring core subsystems 34** Configuring the Thread Pool subsystem 35 Configuring the Thread factory 35 Bounded Thread Pool 36 Unbounded Thread Pool 38 Queueless Thread Pool 39 Scheduled Thread Pool 40 **Configuring the application server logging 40** Choosing your logging implementation 41 Configuring the logging subsystem 42 Configuring loggers 48 Configuring log4j in your application 49 **Summary 51 Chapter 3: Configuring Enterprise Services 53 Configuring database connectivity 53** Installing the JDBC driver 54 Adding a local datasource 56 Configuring the connection pool 57 Configuring the statement cache 58 Adding an xa-datasource 58 A shortcut for installing a datasource 59 Choosing the right driver deployment strategy 60 Configuring a datasource programmatically 62 **Configuring Enterprise Java Beans 62** Configuring the EJB components 64 Configuring Stateless Session Beans 65 Configuring Stateful session beans 67 Configuring the message driven beans 69 Configuring the Timer service 70 Configuring the messaging system 71 Configuring the transport 74 Configuring connection factories 77 Configuring JMS destinations 78 Customizing destinations with address settings 79 HornetQ persistence configuration 80 Configuring transactions 82 **Summary 84 Chapter 4: JBoss Web Server Configuration 85 Configuring the Web container 85** Where is the Web server configuration? 86 Configuring the Web server connectors 86

**~~[ ii ]~~**

*Table of Contents*

Configuring the Web server resources 92 Configuring static resources 92 Configuring dynamic resources 93

**Creating and deploying a web application 95** Creating a new Dynamic Web Project 96 Adding JSF components 98 Adding the EJB layer 101 Choosing the web context of the application 103 Deploying the web application 104 Adding a remote EJB client 106 Configuring data persistence 110 Configuring entities 112 Configuring persistence in other application archives 114 Switching to a different provider 115 **Summary 116 Chapter 5: Configuring a JBoss AS Domain 117 Introducing the JBoss AS domain 117** Understanding the default domain configuration 118 Starting up and stopping a domain 120 **Configuring the domain 121** Configuring the domain.xml file 122 Configuring the host.xml file 124 Configuring the management interfaces 125 Configuring the network interfaces 125 Configuring the domain controller 126 Configuring the JVM 127 Adding JVM options to a server definition 127 Order of precedence between elements 128 Configuring server nodes 129 Applying domain configuration 130 **An example domain configuration 131** Changing the domain configuration at runtime 136 **Summary 138 Chapter 6: Deploying Applications on JBoss AS 7 141 Deploying resources on the application server 142** The JAR archive 142 The WAR archive 142 The EAR archive 143 **Deploying applications on JBoss AS standalone 144** Automatic application deployment 144 Deploying applications on a custom folder 145 Changing the behavior of the application scanner 145 Deploying an application using the CLI 146

**~~[ iii ]~~**

*Table of Contents*

Deploying an application using the Web admin console 148 Deploying an application using the Eclipse environment 149 Manual application deployment 151 **Deploying applications on a JBoss AS domain 154** Deploying to an AS domain using the CLI 155 Deploy to all server groups 155 Deploy to a single server group 156 Deploying to an AS domain using the Admin console 158 **JBoss AS 7 classloading explained 160** Getting to know module names 161 Finding the isolation level 162 Implicit dependencies: 162 Explicit dependencies 163 Setting up global modules 166 Advanced deployment strategies 166 Setting up a single module dependency 167 Excluding the server automatic dependencies 168 Isolating sub-deployments 168 Using Class-Path declaration to solve dependencies 171 **Summary 172 Chapter 7: Managing the Application Server 175 The Command Line Interface (CLI) 175** How to use the CLI 177 Navigating through the resources and executing operations 178 Executing commands with the CLI 185 Getting help from the CLI 190 Executing CLI scripts in batch 190 Advanced batch commands 191 Executing scripts in a file 192 Redirecting non-interactive output 193 Taking snapshots of the configuration 193 What the application server saves for you 195 Taking your own snapshots 196 History of CLI 197 **The new Web admin console 197** Accessing the admin console 198 Configuring server profiles 199 Configuring datasources 200 Configuring JMS destinations 203 Configuring Socket Binding groups 204 **CLI or Web console ? 205 Summary 206**

**~~[ iv ]~~**

*Table of Contents*

**Chapter 8: Clustering 207 Setting up JBoss clustering 207** Setting up a cluster of standalone servers 208 Cluster of nodes running on different machines 208 Setting up a cluster of domain servers 215 **Configuring JBoss clustering 219** Configuring the JGroups subsystem 220 Customizing the protocol stack 223 **Configuring the Infinispan subsystem 224** Configuring session cache containers 225 Choosing between replication and distribution 228 Configuring hibernate cache 231 Using replication for hibernate cache 234 Advanced Infinispan configuration 234 Configuring Infinispan threads 235 Configuring Infinispan transport 236 **Clustering the messaging subsystem 236** Configuring messaging credentials 239 **Summary 240 Chapter 9: Load-balancing Web Applications 241 Using Apache web server with JBoss AS 7 242** Configuring mod\_jk 242 Configuring mod proxy 245 **Load-balancing with mod\_cluster 248 Installing mod\_cluster 249 Managing mod\_cluster with the CLI 253 Managing your web contexts with CLI 255 Adding native management capabilities 255 Managing web contexts using the configuration file 257 Troubleshooting mod\_cluster 257 Testing mod\_cluster 258 Load-balancing between nodes 259** Using load metrics 260 Example setting dynamic metrics on a cluster 262 **Configuring clustering in your applications 265 Clustering session beans 265 Clustering entities 267 Caching entities 269**

**~~[ v ]~~**

*Table of Contents*

**Caching queries 270 Clustering web applications 271 Summary 272**

**Chapter 10: Securing JBoss AS 7 273 Approaching Java security API 274** JBoss AS 7 security subsystem 276 Using the UserRoles login module 279 Using the database login module 280 Encrypting passwords 281 Using an LDAP login module 282 Connecting LDAP to JBoss AS 283 Securing web applications 286 Securing EJBs 288 Securing web services 290 **Securing the AS 7 management interfaces 291 Securing the transport layer 294** Enabling the Secure Socket Layer on JBoss AS 296 Certificate management tools 297 Securing the HTTP communication with a self-signed certificate 298 Securing the HTTP communication with certificate signed by a CA 299 **Summary 301 Chapter 11: Taking JBoss AS 7 in the Cloud 303 Introduction to cloud computing 303** Cloud Computing versus Grid Computing 304 Advantages of cloud computing 305 Types of cloud computing 306 Layers of cloud computing 307 **JBoss cloud infrastructure 309** Starting with OpenShift Express 310 Installing OpenShift client tools 310 Create an OpenShift Express domain 311 Create your first OpenShift Express application 312 Watching what's happening on the server 317 Managing AS 7 applications 318 Configuring your applications on the cloud 319 OpenShift storage management 320 Moving to OpenShift Flex 321 Creating a sample application 323 Going beyond the basics of OpenShift Flex 329 **Summary 333**

**~~[ vi ]~~**

*Table of Contents*

**Appendix 335 Startup options 335 General commands 335 Domain mode 336 Deploy commands 337 JMS 337 Datasources 338**

Datasources (using operation on resources) 338 **Mod\_cluster 339 Batch 340 Snapshots 340 Index 341**

**~~[ vii ]~~**

Preface

As the Java EE standard has evolved and matured, the API has become increasingly rich and complex. The next generation of application servers needs to be modular and configurable to run only specific services or containers. JBoss AS 7 promises to meet those requirements but the configuration of a complex application server is composed of a mix of administrative and management tasks which often overlap, generating confusion.

JBoss AS 7 Configuration, Deployment and Administration will give you an expert's understanding of every component that makes up the JBoss application server, and will show you how to use them, helping you to cut down the learning curve for this exciting product dramatically.

This book will guide you through configuration, management, deployment, and advanced administration of JBoss AS 7 in a logical order to avoid the common pitfalls of setting up a new AS. At first, we will The book dives into the new application server structure and shows you how to install it. You will then learn how to configure the core AS services, including thread pools, the messaging system, and the transaction service. Finally, you will learn how to deploy and manage your applications through all possible configuration variants, including standalone or domain servers, through a single node or a cluster of application servers. and then tune the performance of your AS to achieve an efficient, indispensible application server. In the last part of this book, we will show also how to deliver your applications on the cloud as a service.

**What this book covers**

*Chapter 1*, *Installing JBoss AS 7*, will introduce you to the new application server, depicting its most important features and showing how to install it on your machine.

*Chapter 2*, *Configuring the Application Server*, covers the application server main configuration file and some core parts of it, such as like the Thread pool configuration and the Logging subsystem.

*Preface*

*Chapter 3*, *Configuring Enterprise Services,* will teach the reader how to model the core Java Enterprise services using the standalone configuration file.

*Chapter 4*, *JBoss Web Server Configuration*, completes the standalone server configuration by looking at the Web subsystem. This chapter also includes a full Java EE example which teaches the reader how to create and configure a Java EE 6 application on JBoss AS 7.

*Chapter 5*, *Configuring a JBoss AS Domain*, teaches the reader how to shape the domain server configuration and which is the criteria behind the choice of a standalone or domain server configuration.

*Chapter 6*, *Deploying Applications on JBoss AS*, covers all the nuts and bolts related to the application deployment. It also discusses the class loading mechanism which that is used by the application server when applications are loaded.

*Chapter 7*, *Managing the Application Server*, teaches the reader which management tools can be used to control the application server instances.

*Chapter 8*, *Clustering*, covers the AS 7 clustering capabilities that serve as an essential component to providing scalability and high availability to your applications.

*Chapter 9*, *Load Balancing Web Applications*, discusses the second important concern of clustering, which is the ability to make several servers participate in the same service and do the same work. In other words, it discusses how to load balance the number of requests across the available servers.

*Chapter 10*, *Securing JBoss AS 7*, covers the foundation of JBoss Security framework and how to secure Enterprise applications and the server management interfaces.

*Chapter 11*, *Taking JBoss AS 7 into the cloud*, shows how to deliver Java EE applications on a cloud environment using the Red Hat OpenShift platform.

*Appendix*, provides a quick reference for the most common commands and operations that can be used to manage the application server with the CLI.

**What you need for this book**

Some prior knowledge of Java Enterprise is expected, but no prior knowledge of JBoss application server is needed. However, the book contains many hints to upgrading your existing JBoss AS configuration in the new server release, so affectionate JBoss users will surely gain better dividends from reading this book.

**~~[ 2 ]~~**

*Preface*

**Who this book is for**

Java system administrators, developers, and application testers will benefit from this book. The brand new features in AS 7 mean that everyone can get something from this book, whether you have used JBoss AS or not.

**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: " Next, you need to state the path to the JDBC driver resource and finally the module dependencies."

A block of code is set as follows:

<module xmlns="urn:jboss:module:1.0" name="com.mysql"> <resources> <resource-root path="mysql-connector-java-5.1.17-bin.jar"/> </resources>

<dependencies>

<module name="javax.api"/>

<module name="javax.transaction.api"/>

</dependencies>

</module>

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

Connection result = null;

try {

Context initialContext = new InitialContext();

DataSource datasource =

(DataSource)initialContext.lookup("java:/MySqlDS");

**result = datasource.getConnection();**

}

catch ( Exception ex ) {

log("Cannot get connection: " + ex);

}

**~~[ 3 ]~~**

*Preface*

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

**[disconnected /] connect**

**Connected to localhost:9999**

**[localhost:9999 /] :reload**

**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: " Now, you should be able to see the JBoss AS 7 enlisted as a server by choosing **New** | **Server** from the upper menu and expanding the JBoss Community option:".

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 book that you need and would like to see us publish, please send us a note in the **SUGGEST A TITLE** form on www.packtpub.com or e-mail suggest@ packtpub.com.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book on, 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.

**~~[ 4 ]~~**

*Preface*

**Downloading the example code for this book** 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/support, selecting your book, clicking on the **let us know** 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.

**~~[ 5 ]~~**

Installing JBoss AS 7

Java continues to be the most widely-used single-programming language today, edged out only by all scripting languages grouped together. Java remains also the language and/or platform of choice for Enterprise and Web application development, especially large-scale application development.

Nevertheless the Java language has changed a lot since its first appearance and will possibly continue to change in the future. Oracle, who has acquired Sun, the home of the Java language, has stated that its high-level Java strategy is to enhance and extend the reach of Java to new and emerging software development paradigms: **simplify**, **optimize**, and **integrate** the Java platform into new deployment architectures; and invest in the Java developer community allowing for increased participation.

At this point, one question arises: are application servers ready to meet these new paradigms? Originally designed as web containers for supporting web applications and, later, as EJB containers for remotely accessible services, application servers have expanded considerably from their simple origins. Today, most application servers provide a comprehensive service layer, which delivers support for distributed transactions, clustering, security, and so on.

In addition, a large number of open source building blocks have been added to the application server and they are heavily used in today's products. However, integrating all these libraries does not come without a price because each library has, in turn, evolved with complexity, following its own unsynchronized evolution path and requiring more and more additional libraries to work.

As most IT experts agree, the challenge for today's application server is to combine a rich set of features requested by the customers along with a lightweight and flexible container configuration.

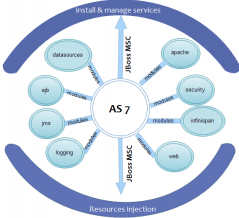
*Installing JBoss AS 7*

The 7.0 release of JBoss AS is designed around a brand new kernel, which is now based on two main projects:

• **JBoss Modules**: This handles class loading of resources in the container. You can think about JBoss Modules as a thin bootstrap wrapper for executing an application in a modular environment.

• **Modular Service Container** (**MSC**): This one provides a way to install, uninstall, and manage services used by a container. MSC further enables resources injection into services and dependency management between services.

The following picture depicts the basic architecture of the new application server kernel:



Understanding the details of the new modular kernel may be a little invasive at the beginning of the book, however, we will smoothly introduce some useful concepts within this chapter, just to make sure you get acquainted with the server configuration basics.

For the time being, let's just start installing the core server components and their required dependencies.

**~~[ 8 ]~~**

*Chapter 1*

**Getting started with the application server** The first step in learning about the application server will be installing all the necessary stuff on your machine in order to run it. The application server itself requires just a Java Virtual Machine environment installed.

As far as the hardware requirements are concerned, you should be aware that the server distribution, at the time of writing, requires about 75MB of hard-disk space and allocates a minimum of 64MB and a maximum of 512MB for a standalone server.

In order to get started, this is our checklist:

1. Install the Java Development Kit where JBoss AS 7 will run.

2. Install JBoss AS 7.0.

3. Install Eclipse development environment.

At the end of this chapter, you will have all the instruments to get started with the application server.

**Installing the Java environment**

JBoss AS is a full Java application server and hence it requires a Virtual Machine for the Java Platform to run on.

At the time of writing, the J2SE 1.7 has been finally released and

it's available for download. Although, there are no known issues

with this Java release, there are still a few users who have tested it

intensively with the new application server.

For this reason, at the moment, we suggest you start with J2SE 1.6

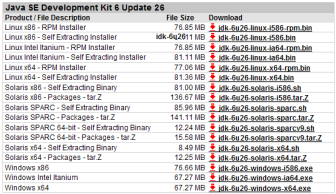
and consider moving to the new Java platform in the near future.

So, let's move on the Oracle download page, which now hosts all JDK downloads: http://www.oracle.com/technetwork/java/javase/downloads/index.html

**~~[ 9 ]~~**

*Installing JBoss AS 7*

Choose to download the latest JDK/JRE which is, at the time of writing, the JDK 1.6 Update 26.



The download will take a few minutes depending how fast your network is. Once the download is complete, run the executable file to start the installation. (The actual name of the installer varies, also if you are on a 32-bit system or 64-bit. However, the steps will be the same, just the name will change).

**jdk-6u26-windows-i586.exe # Windows**

**sh jdk-6u26-linux-i586.bin # Linux**

If you are installing on a Linux/Unix box, you can safely accept all the defaults given to you by the setup wizard. The Windows users should stay away from the installation paths that include empty spaces, such as C:\Program Files, as this leads to some issues when you are referencing the core libraries. An installation path such as C:\Software\Java or simply C:\Java is a better alternative.

When the installation is complete, we need to update a couple of settings on the computer so it can interact with Java. The most important setting is JAVA\_HOME, which is directly referenced by JBoss startup script.

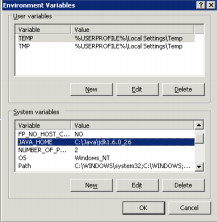
If you are running Windows XP-2000, follow these steps:

1. Right-click on **My Computer** and select **Properties** from the context menu. 2. On the **Advanced** tab, click on the **Environment Variables** button. 3. Then, in the **System Variables** box, click on **New**.

**~~[ 10 ]~~**

*Chapter 1*

4. Name the new variable as JAVA\_HOME, and give a value of the path to your JDK installation; I would recommend something like: C:\Java\ jdk1.6.0\_26.



**Windows Vista tip**

Because of increased security in Windows Vista, standard

users must have **User Account Control** (UAC) turned on

to change the environment variables, and the change must be

completed via user accounts. In the User Accounts window,

under Tasks, select **Change my environment variables**.

Use the **New**, **Edit** or **Delete** buttons to amend environment

variables

5. Now it's time to modify the system's PATH variable. Double-click on the PATH system variable. In the box that pops up, navigate to the end of the **Variable Value** line, add a semicolon to the end, and then add the path to your JDK. This will be something like %JAVA\_HOME%\bin.

Unix/Linux users can add the following commands in the user's profile scripts:

**export JAVA\_HOME=/installDir/jdk1.6.0\_26**

**export PATH=$JAVA\_HOME/bin:$PATH**

**~~[ 11 ]~~**

*Installing JBoss AS 7*

**Installing JBoss AS 7**

JBoss application server can be freely downloaded from the community site: http://www.jboss.org/jbossas/downloads/.



As you can see from the above picture, as I'm writing this book, you can choose between the 7.0.2 final release (Arc) and the newer 7.1.0 Beta. If you plan to use JBoss AS 7 in production, we suggest you using the former stable release which provides all the functionalities contained in this book except for one mentioned in *Chapter 4* ("Adding a Remote EJB client").

The 7.0.2 release is available in the Web profile certified distribution and in the full distribution, which contains all the available server modules. On the other hand, if you are on a learning path to the new server, just proceed to the new 7.1.0 Beta release, which is available in the single EE6 server release."

Choose to download the full server distribution. You will be then warned that this download is part of a community release and, as such, it is not yet fully supported. As said before, RedHat maintains the Enterprise releases of JBoss middleware and, hopefully, the AS 7 will be soon part of it.

JBoss AS 7 does not come with an installer; it is simply a

matter of extracting the compressed archive.

Windows users can simply use any uncompress utility, such as WinZip, or WinRAR taking care to choose a folder, which doesn't contain empty spaces. Unix /Linux should use the unzip shell command to explode the archive:

**unzip jboss-as-7.0.2.Final.zip**

**~~[ 12 ]~~**

*Chapter 1*

**Security warning**

Unix/Linux users should be aware that JBoss AS does not

require root privileges as none of the default ports used

by JBoss are below the privileged port range of 1024. To

reduce the risk of users gaining root privileges through

the JBoss AS, install and run JBoss as a non-root user.

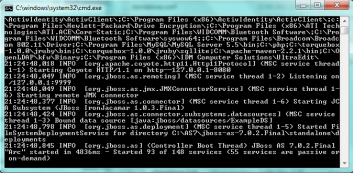
**Starting up JBoss AS**

After you have installed JBoss, it is wise to perform a simple startup test to validate that there are no major problems with your Java VM/operating system combination. To test your installation, move to the bin directory of your JBOSS\_HOME directory and issue the following command:

**standalone.bat # Windows users**

**$ standalone.sh # Linux/Unix users**

Here's a sample JBoss AS 7 startup console:



The above command starts up a JBoss standalone instance that's equivalent of starting the application server with the run.bat/run.sh script used by earlier AS releases. You will notice how amazingly fast is starting the new release of the application server; this is due to the new modular architecture, which only starts up necessary parts of the application server container needed by loaded applications.

**~~[ 13 ]~~**

*Installing JBoss AS 7*

If you need to customize the startup properties of your application server, then you need to open the standalone.conf (or standalone.conf.bat for the Windows users) where the memory requirements of JBoss are declared. Here's the Linux core section of it:

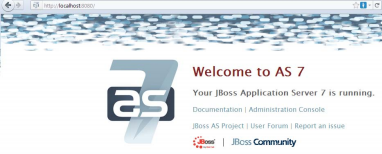
**if [ "x$JAVA\_OPTS" = "x" ]; then**

**JAVA\_OPTS="-Xms64m -Xmx512m -XX:MaxPermSize=256m -Dorg.jboss.resolver. warning=true -Dsun.rmi.dgc.client.gcInterval=3600000 - Dsun.rmi.dgc. server.gcInterval=3600000"**

**fi**

So, by default, the application server starts with a minimum memory requirement of 64MB of heap space and a maximum of 512MB. This will be just enough to get started, however, if you need to run a core Java EE application on it, you will likely require at least 1GB of heap space or 2GB or more depending on your application type. Generally speaking, 32-bit machines cannot execute a process whose space exceeds 2GB, however on 64 bit machines, there's essentially no limit to process size.

You can verify that the server is reachable from the network by simply pointing your browser to the application server's welcome page, which is reachable by default at the well-known address: http://localhost:8080.



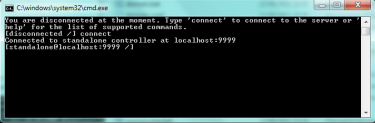
**Connecting to the server with the command line interface**

If you have been using previous releases of the application server you might have heard about the **twiddle** command-line utility that queried the MBeans installed on the application server. This utility has been replaced by a more sophisticated interface named the **Command Line Interface (CLI)**, which can be found in the JBOSS\_HOME/bin folder.

**~~[ 14 ]~~**

*Chapter 1*

Just launch the jboss-admin.bat script (or jboss-admin.sh for Linux users) and you will be able to manage the application server via a shell interface.



We have just started an interactive shell session which is also able to use the command-line completion (by pressing the *Tab* key) to match partly-typed command names. No more searches for finding the exact syntax of commands!

In the previous screenshot, we have just connected to the

server using the connect command, which by default uses the

loopback server address and plugs into the port 9999.

The command-line interface is discussed in depth in Chapter 7, *Managing the Application Server*, which is all about the server-management interfaces; We will, however, have an initial taste of its basic functionalities in the following sections to get you accustomed to this powerful tool.

**Stopping JBoss**

Probably the easiest way to stop JBoss is by sending an interrupt signal with *Ctrl+C*.

However, if your JBoss process was launched in the background or rather is running on another machine (see next), then you can use the CLI interface to issue an immediate shutdown command:

**[disconnected /] connect**

**Connected to localhost:9999**

**[localhost:9999 /] :shutdown**

**~~[ 15 ]~~**

*Installing JBoss AS 7*

**Locating the shutdown script**

There is actually one more option to shutdown the application server, which is pretty useful if you need to shut down the server from within a script. This option consists of passing the --connect option to the admin shell, thereby switching off the interactive mode:

**jboss-admin.bat --connect command=:shutdown # Windows jboss-admin.sh --connect command=:shutdown # Unix / Linux**

**Stopping JBoss on a remote machine**

Shutting down the application server, which is running on a remote machine, is just a matter of connecting and providing the server's remote address to the CLI:

**[disconnected /] connect 192.168.1.10**

**Connected to 192.168.1.10:9999**

**[192.168.1.10:9999 /] :shutdown**

Since JBoss AS 7.1.0 Beta1 accessing remotely the CLI requires

authentication. Check out Chapter 10 for more information about it.

**Restarting JBoss**

The command-line Interface contains a lot of useful commands. One of the most interesting options is the ability to reload all or parts of the AS configuration using the reload command.

When issued on the **root node path** of the AS server, it is able to reload the services configuration:

**[disconnected /] connect**

**Connected to localhost:9999**

**[localhost:9999 /] :reload**

**Installing Eclipse environment**

Although the main focus of this book is the administration of the application server, we are also concerned with the application packaging and deploying. For this reason, we will sometimes add examples that require a development environment to be executed.

**~~[ 16 ]~~**

*Chapter 1*

The development environment used in this book is Eclipse known by Java developers worldwide and contains a huge set of plugins to expand its functionalities. Besides this, Eclipse is the first IDE that is compatible with the new application server.

So, let's move to the downloading page of Eclipse which is located at: http://www.eclipse.org.

From there, download the latest Enterprise Edition (at the time of this writing, it's version 3.7 and is also known as Indigo). The compressed package contains all the Java EE plugins already installed and requires about 210MB of disk space:



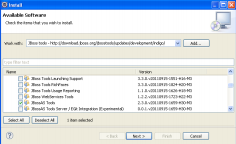
Once you have unzipped the previously downloaded file, you will see a folder named eclipse. In that folder, you will find the eclipse application (a big blue dot). We recommend you create a shortcut on the desktop to simplify the launching of Eclipse. Note that just like JBoss AS, Eclipse does not have an installation process. Once you have unzipped the file, you are done!

**Installing JBoss tools**

The next step will be installing the JBoss AS plugin, which is a part of the suite of plugins named JBoss tools. Installing new plugins in Eclipse is pretty simple; just follow these steps:

1. From the menu, choose **Help** | **Install New Software**.

2. Then, click on the **Add** button where you will enter the JBoss tools' download URL (along with a description): http://download.jboss.org/jbosstools/ updates/development/indigo/.

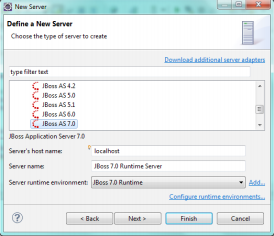


**~~[ 17 ]~~**

*Installing JBoss AS 7*

As you can see from the previous screenshot, you need to check the **JBossAS Tools** plugin and move forward to the next options where you will complete the installation process. Once done, restart when prompted.

Now, you should be able to see the JBoss AS 7 enlisted as a server by choosing **New** | **Server** from the upper menu and expanding the **JBoss Community** option:



Completing the server installation into the Eclipse is quite straightforward as it just requires pointing to the folder where your server distribution, hence we will leave this to the reader as a practical exercise.

**Exploring the application server file system**

Once done with the installation of all the necessary tools, we will concentrate on the new application server structure. The first thing you'll notice when you browse through the application server folders is that its file system is basically divided into two core parts: the dichotomy reflects the distinction between **standalone** servers and **domain** servers.

**~~[ 18 ]~~**

*Chapter 1*

The concept of the domain server is not new in the market of application servers, however, it is introduced for the first time in JBoss AS as a way to *manage and coordinate* a set of instances of the application server. An application server node which is not configured as part of a domain is qualified as standalone server and it resembles in practice to a single instances of the application server you used in earlier releases.

We will discuss the concept of domains in detail in *Chapter 5, Configuring a JBoss AS Domain*, of this book; for the time being, we will explore the different file system structures for both kinds of servers.

From a bird's eye perspective, we can see the one that the main file system is split in two: one section which is pertinent to **domain servers** and another which is relative to **standalone servers**. The following screenshot depicts the new tree of the application server:

bin

JBoss-7.0.0

docs

domain

modules standalone

configuration content

lib

configuration deployments lib

ext ext

In the next section, we will enter into the single folders of the new AS infrastructure, dissecting their content and what they are used for in the application server.

**~~[ 19 ]~~**

*Installing JBoss AS 7*

**The bin folder**

This is where you start your application server instances. You can start a standalone server by launching the standalone.bat (standalone.sh for Linux users). In addition to the startup scripts, you can find the standalone.bat, which can be used to customize JBoss' bootstrap processIf you are going to use a domain of servers, you will use the domain.bat script (domain.sh for Linux users). This shell script starts up a set of application server instances as specified by the domain configuration file.

The bin folder also includes another useful script command, named jboss admin.bat (jboss-admin.sh for Linux users), which starts the new interactive command-line interface.

In addition, this folder contains the web services utility scripts (wsconsume.sh and wsprovide.sh) used to generate the web services definition language and the corresponding Java interfaces.

**The docs folder**

In spite of its name, this folder does not contain the server documentation, but two subfolders: the first one named licenses barely contains the licenses information for the application server and its dependencies. You might use the licenses.xml file as quick reference for finding out which are the version shipped with the default application server modules or dependencies. For example, JBoss AS 7 ships with the release 4.0.0.CR2 of Hibernate core libraries.

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-core</artifactId>

<version>4.0.0.CR2</version>

<licenses>

<license>

<name>GNU Lesser General Public License</name> <url>http://www.gnu.org/licenses/lgpl-2.1.html</url> <distribution>repo</distribution>

<comments>See discussion at http://hibernate.org/license for more details.</comments>

</license>

</licenses>

</dependency>

The other subfolder named schema contains the .xsd files which are used by the configuration as schema.

**~~[ 20 ]~~**

*Chapter 1*

**The domain folder**

The next folder is the domain folder, which contains the domain structure, split across a set of folders:

The configuration folder contains as you might imagine the configuration files. The main configuration file is domain.xml which contains all services which are used by the nodes of the domain. It also configures the socket-binding interfaces for all services.

Along with it, the other key file is host.xml which is used to define the management aspects of the domain.

The last file contained in the configuration folder is logging.properties which are used to define the logging format of the bootstrap process.

The content folder is used as a repository to store deployed modules.

The lib folder hosts the sub-folder ext, which is there to support Java SE/EE style "extensions". Some of the application server deployers are able to scan this folder for additional libraries which are picked up by the local class loader; nevertheless this approach is not recommended and maintained only for compliance with the language specifications. The modules folder should be the one and only path for your libraries.

The log folder contains as you might imagine the logging output of the domain. The file, by default, is truncated every time the server is rebooted.

The **servers** folder holds a set of sub-folders for each server defined in the configuration file. The most useful directory contained beneath each server is the log folder which is the location where single instances emit their log. The data folder is used by the application server to store its runtime data, such as the transaction logging The tmp folder is a temporary location for some resource artifacts which is not a big issue for any of you.

**The standalone folder**

If you are running the application server in standalone mode, this is the part of the AS file system you will be interested in. Its structure is quite similar to the **domain** folder with the notable exception of a **deployment** folder. Let's proceed with order. Just below the domain folder, you will find the following set of subdirectories.

**~~[ 21 ]~~**

*Installing JBoss AS 7*

The **configuration** folder contains also the application server configuration files. As a matter of fact the application server ships with a set of different configuration files, each one using a different set of functionalities. Launching the standalone start-up script, by default, the standalone.xml configuration file will be used.

Besides standalone.xml, this folder contains the logging.properties file, which is also about the logging of bootstrap process. The other file you will find within it, mgmt-users.properties, can be used to secure the management interfaces. The security is discussed in detail in *Chapter 10*, *Securing JBoss AS,* of this book.

The data folder is used by the application server to store its runtime data, such as the transaction logging.

The deployments folder is the location in which users can place their deployment content (for example, WAR, EAR, JAR, SAR files) to have it automatically deployed into the server runtime. Users, particularly those running production systems, are encouraged to use the JBoss AS management APIs to upload and deploy deployment content instead of relying on the deployment scanner subsystem that periodically scans this directory. See *Chapter 6*, *Deploying Applications on JBoss AS 7,* for more details.

The lib folder hosts the sub folder ext, which is used to define extensions of the application server. The same considerations for the domain's lib path apply here.

The log folder contains the logs emitted by the standalone instance of the application server. The default log file, named server.log, is truncated every time the server is rebooted.

The tmp folder holds is used by JBoss **Virtual File System** as a temporary location for resource artifacts.

**The welcome-content folder:**

This directory contains the default page which is loaded when you surf on the home of your application server (http://localhost:8080). In terms of web server configuration, this is the **Web root context**.

**The modules folder**

Beneath the modules folder, you will find the application server's set of libraries, which are a part of the server distribution.

**~~[ 22 ]~~**

*Chapter 1*

Historically, JBoss AS releases used to manage their set of libraries in different ways. Lets recap to make some order. Earlier, Release 4.x was used to define the core server libraries into the JBOSS\_HOME/server libraries. Thereafter, each server definition had its specific library into the server/<servername>/lib folder.

This approach was pretty simple this way; however, it led to a useless proliferation of libraries which were replicated in the default/all server distribution.

The release 5.x and 6.x had the concept of the common/lib folder which was the main repository for all modules that were common to all server definitions. Each server distribution still contained a server/<servername>/lib path for the libraries that were specific to that server definition. Unchanged from the earlier release was the repository for core server modules comprised by JBOSS\_HOME/server.

JBoss AS 7 follows a real modular approach deprecating all earlier approaches. The server bootstrap libraries are now located at the root of the application server. There you can find the jboss-modules.jar archive, which is all you need to bootstrap the new application server kernel, based on the JBoss modules.

The application server modules are now defined beneath the modules folder, grouped in a set of subfolders each one dedicated to a set of resources. At first sight, this approach might seem less intuitive than previous releases; however, once accustomed to it, you will find it much easier to handle module installation/updates.

The following table resumes the diverse approaches used across different server releases:

**AS**

**Release**

**Bootstrap libs Server libs**

*4.x* JBOSS\_HOME/server JBOSS\_HOME/server/<server>/ lib

*5.x – 6.x* JBOSS\_HOME/server JBOSS\_HOME/common/lib and

JBOSS\_HOME/server/<server>/

lib

*7.x* JBOSS\_HOME/jboss-modules.jar JBOSS\_HOME/modules subfolders

Listing all the modules will take up too much space; however, the module repository layout is pretty much the same as the module name. For example, the org.jboss. as.ejb3 module will be found in org/jboss/as/ejb3 subfolder of the modules folder. In the last section of this chapter, we will see how modules are actually loaded by the application server.

**~~[ 23 ]~~**

*Installing JBoss AS 7*

**Loading application server modules** Learning more about JBoss module is essential if you want to understand the next chapters that discuss about the server configuration. Basically, each module is a pluggable unit. As depicted by the following picture. JBoss modules can load libraries using two approaches:

• Using a static file system

• Using direct JAR execution

Static module loading from the file system

jboss

modules.jar modules

org

JBoss

logging

main jboss logging.jar

Module direct JAR execution

jboss

modules.jar yourApp.jar

Using a **static file system** approach is the simplest way to load a module, and it's used as a default when starting up the application server. It is based on the assumption that the filesystem reflects the name and version of modules used. All you need to provide to the environment is the location where modules are stored. So, for example, you could start a standalone instance of JBoss AS 7 using the following command:

**java -jar jboss-modules.jar -mp "%JBOSS\_HOME%\modules" org.jboss. as.standalone**

**~~[ 24 ]~~**

*Chapter 1*

The argument module path (-mp) actually points to the root directory (or directories), which will be searched by the default module loader for module definitions. A module is defined using a simple XML descriptor, like this:

<module xmlns="urn:jboss:module:1.0" name="org.jboss.msc">

<main-class name="org.jboss.msc.Version"/>

<resources>

<resource-root path="jboss-msc-1.0.0.Beta8.jar"/> </resources>

<dependencies>

<module name="javax.api"/>

<module name="org.jboss.logging"/>

<module name="org.jboss.modules"/>

</dependencies>

</module>

Basically, a module definition contains two main elements: the **resources** defined in the module (and their path) and the module **dependencies**. The previous example is the module definition for the JBoss MSC module which is contained in the jboss msc-1.0.0.Beta8.jar file and bears a dependency on javax.api, org.jboss. logging and org.jboss.modules module.

A module which is defined with a main-class element is said to be

**executable**. In other words, the module name can be listed on the

command line, and the standard static main(String[]) method in

the named module's main-class will be loaded and executed.

The other way to approach the module repository is by using **direct JAR execution**. This means that the module information, such as its dependencies, is contained in the MANIFEST file.

When the module is executed as a JAR file you just need to provide the name of your application module, which is packed in JAR file, and it will be picked up by JBoss modules:

**java -jar jboss-modules.jar -jar your-app.jar**

**~~[ 25 ]~~**

*Installing JBoss AS 7*

Then, in your application.jar, you might specify a set of module definitions, much the same way you did for flat filesystem modules. These definitions are contained in the META-INF/MANIFEST.MF file:

Main-Class: mypackage/MyClass

Dependencies: org.jboss.logging

**Summary**

In this chapter, we introduced the new application server, its new features, and the reasons which are behind these changes.

The release 7.0 of the JBoss application server provides the foundation for a new generation of application server, which will deliver the truly modular services.

The outcome of this is a lighter platform based on a tiny modular kernel that is able to load and extend its components from the filesystem and activate them just when they are required by the user.

The physical structure of the application server has been modified to reflect the dichotomy between standalone servers and domain servers, the former being basically a single node instance and the latter a set of managed resources controlled by a Domain Controller and a Host Controller.

In the next chapter we will enter into the details of the application server configuration, focusing our attention on the standalone server configuration file, which holds both core application server configuration and the stack of Enterprise services running on top of it.

**~~[ 26 ]~~**

Configuring

the Application Server

The first chapter has given us the initial groundwork to get started with JBoss AS 7. It is time for us to dive right into the new configuration and see how to shape a standalone instance of the application server. As you will see, the application configuration has also been renewed, moving from a large set of XML files to a single monolithic file.

The new configuration file is made up of a list of subsystems, which include the application server core services and standard Java EE services. By the end of this chapter, we will have covered the following topics:

• Introducing the server configuration file

• Configuring the application server's Thread Pool

• Configuring the application server's logging subsystem

**Configuring the application server** The structure of the application server is maintained into a single file, which acts as a main reference for all server configurations. This file is not a static file, as it reflects the changes that are made when the server is running, for example, adding a new component, such as a JMS destination, or deploying an application.

The default configuration files are named standalone.xml for standalone servers and domain.xml for an application server domain. An application server domain can be seen as a specialized server configuration, which also includes the **Domain** and **Host controller** set-up. We will discuss the application server domain in *Chapter 5*, *Configuring a JBoss AS Domain*. However, as far as the core services configuration is concerned, what we learn here will be suitable for the domain configuration as well.

*Configuring the Application Server*

You can define as many configuration files as you need. For example, the AS 7.0.2 release provides a few variants of the standalone.xml, such as standalone-preview.xml (which includes JMS and web services subsystems) and the standalone-ha.xml (which can be used to start a cluster-aware application server). If you want to switch to another

configuration file, just issue:

standalone.bat --server-config customConfiguration.xml

The standalone.xml file is located under the JBOSS\_HOME/standalone/ configuration folder. This configuration file consists of a large XML file, which is validated by a set of .xsd files mentioned in the <server> element.

If you want to check the single .xsd files, you can find them in the

JBOSS\_HOME/docs/schema folder of your server distribution. You

can get to know all the available server parameters with a simple

inspection of these files or by importing them into your Eclipse

environment. Once they are located in your project, right-click your

file and choose **Generate | XML File**.

The application server configuration follows a tree-like structure that contains, at the root element, the server definition.

extensions

paths

management-interfaces

server

profiles

interfaces

socket-binding-group system-properties

deployments

**~~[ 28 ]~~**

1...\* profile subsystem

*Chapter 2*

In the following sections, we will show in detail each part of the server configuration. This will be helpful to understand the role of each single component in the application server, although we warn you from manually changing the configuration file, since this can lead to unchecked data modifications that can easily corrupt the file.

The best practice for changing the server configuration is to use the

**Command Line Interface** (**CLI**) or the web admin console, which are

described in *Chapter 7*, *Managing the Application Server*.

**Extensions**

The application server contains a list of basic modules, called extensions, which are shared by all of its services. Extensions can be seen as a special type of module, which are used to extend the functionalities of the application server. Much like standard modules, they are stored in the JBOSS\_HOME/modules folder. Each extension is in turn picked up by the AS classloader at boot time, before any deployment. Here's an extract from the server configuration:

<extensions>

<extension module="org.jboss.as.clustering.infinispan"/> <extension module="org.jboss.as.messaging"/>

</extensions>

The application server detects that a module is an extension by scanning into the META\_INF/services folder of the library. Modules that are

qualified as Extensions contain in the META\_INF/services folder a

placeholder file named org.jboss.as.controller.Extension.

This file simply declares the name of the Extension. For example, for the EJB3 subsystem, it will contain org.jboss.as.ejb3.subsystem.

EJB3Extension.

**Paths**

Beneath module extensions, you can find the definition of paths that can be defined as logical names for file system paths. For example, the following bit of configuration defines a path relative to the AS log directory named log.dir, which translates for standalone servers into JBOSS\_HOME/standalone/log/mylogdir:

<paths>

<path name="log.dir" path="mylogdir" relative-to=

"jboss.server.log.dir"/>

</paths>

**~~[ 29 ]~~**

*Configuring the Application Server*

Now, this path can be referenced in other sections of the configuration file, for example, we are using it as a folder for storing the logging rotating file handler:

<periodic-rotating-file-handler name="FILE" autoflush="true"> <file relative-to="log.dir" path="myserver.log"/>

</periodic-rotating-file-handler>

Please note that the property relative-to is not mandatory. If you

don't include it in your path configuration, the path is meant as an

absolute path.

By default, the application server provides a set of system paths that are available for use as relative paths and cannot be overridden by the user:

**Path Meaning**

jboss.home The root directory of the JBoss AS distribution user.home The user's home directory

user.dir The user's current working directory java.home The Java installation directory

jboss.server.base.dir The root directory for an individual server instance jboss.server.data.dir The directory the server will use for persistent data file storage

jboss.server.log.dir The directory the server will use for logfile storage jboss.server.tmp.dir The directory the server will use for temporary file storage

jboss.domain.servers.dir The directory under which a host controller will create the working area for individual server instances

**Management interfaces**

One of the innovating features of the application server is the powerful administration and management channels that include a CLI and a web-based administration console. The native CLI interface, by default, runs on port 9999, while the web console is bound on port 9990.

<socket-binding-group name="standard-sockets" default

interface="public">

. . . . . . .

<socket-binding name="management-native" interface="management" port="9999"/>

<socket-binding name="management-http" interface="management" port="9990"/>

. . . . . . .

</socket-binding-group>

**~~[ 30 ]~~**

*Chapter 2*

Management interfaces are discussed in detail in *Chapter 7*, *Managing the Application Server*, which provides a detailed coverage of the application server management tools.

**Profiles**

Letting the configuration file flow, you can find the definition of the server's profiles, which is one of the core concepts introduced in AS. A profile can be

seen as a collection of subsystems: each subsystem in turn contains a subset of functionalities used by the application server. For example, the web subsystem contains the definition of a set of connectors used by the container, the messaging subsystem defines the JMS configuration and modules used by the AS's messaging provider, and so on.

One important difference between a standalone and a domain

configuration file is the number of profiles contained in it. When using a standalone configuration, there's a single profile that contains the set of subsystem configurations. Domain configuration can, on the other hand, provide multiple profiles.

**Interfaces**

The interfaces section contains the network interfaces/IP addresses or host names where the application server can be bound.

By default, the standalone application server defines two available network interfaces: the management and the public interface:

<interfaces>

<interface name="management">

<inet-address value="${jboss.bind.address.

management:127.0.0.1}"/>

</interface>

<interface name="public">

<inet-address value="${jboss.bind.address:127.0.0.1}"/> </interface>

</interfaces>

The public network interface is intended to be used for the application server core services:

**<socket-binding-group name="standard-sockets" default interface="public">**

. . . . . .

</socket-binding-group>

**~~[ 31 ]~~**

*Configuring the Application Server*

The management network interface is referenced by the AS management interfaces, as shown in the management interfaces section.

Both network interfaces resolve, by default, to the loop back address 127.0.0.1. This means that, by default, the application server public services and the management services are accessible only from the local machine. By changing the inet-address value, you can bind the network interface to another IP address, which is available on the machine:

<interface name="public">

**<inet-address value="192.168.1.1"/>**

</interface>

If, on the other hand, you want to bind the network interface to all available sets of IP address, you can use the <any-address /> element:

<interface name="public">

**<any-address />**

</interface>

Another useful variation of network interface is the **Network Interface Card** (**NIC**) element, which gathers the address information from the network card name:

<interface name="public">

**<nic name="eth0" />**

</interface>

**Using command-line options to change network interface bindings**

In earlier releases of the application server, you used to launch the

startup script with the additional -b parameter, followed by a valid

host/IP address. This would cause the server to bind on the host/

IP address provided. This option was not available in the initial AS 7

release however it has been restored in the AS 7.1.0 release.

**Socket binding groups**

A socket binding makes up a named configuration of a socket. Within this section, you are able to configure the network ports, which will be open and listening for incoming connections. As we have just seen, every socket binding group references a network interface through the default-interface attribute:

<socket-binding-group name="standard-sockets"

default-interface="public">

<socket-binding name="jndi" port="1099"/>

<socket-binding name="jmx-connector-registry" port="1090"/>**~~[ 32 ]~~**

*Chapter 2*

<socket-binding name="jmx-connector-server" port="1091"/> <socket-binding name="http" port="8080"/>

<socket-binding name="https" port="8447"/>

<socket-binding name="osgi-http" port="8090"/>

<socket-binding name="remoting" port="4447"/>

<socket-binding name="txn-recovery-environment" port="4712"/> <socket-binding name="txn-status-manager" port="4713"/> <socket-binding name="txn-socket-process-id" port="4714"/> <socket-binding name="messaging" port="5445"/>

<socket-binding name="messaging-throughput" port="5455"/> </socket-binding-group>

In order to change the ports where services are bound, you can change the port attribute of its service. A definitely better approach is, however, to use management interfaces that provide an immediate outcome of the affected change. In the following example, we are changing the default port for the http connector using the CLI:

**[standalone@localhost:9999 /] /socket-binding-group=**

**standard-sockets/socket-binding=http:write-attribute(name="port", value="8090")**

**{**

**"outcome" => "success",**

**"response-headers" => {**

**"operation-requires-reload" => true,**

**"process-state" => "reload-required"**

**}**

**}**

**System properties**

This section contains a set of system-wide properties, which can be added to the application server as part of the booting process. The following configuration snippet sets the property named example to dummyvalue:

<system-properties>

<property name="myproperty" value="dummyvalue"/>

</system-properties>

The property can be later retrieved on the application server using: String s = System.getProperty("myproperty");

**~~[ 33 ]~~**

*Configuring the Application Server*

**Deployments**

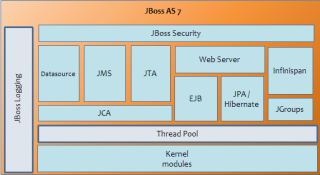
The last section of the configuration file contains the deployed application, which has been registered on the application server. Each time a new application is deployed or un-deployed, this section is updated to reflect the new application stack.

**Configuring core subsystems**

Now that you have grasped the basic concepts of the new configuration file, we will have a look at the peculiarities of single services. Discussing all single subsystems in a single chapter is a daunting task for both the author and for those who will read it later. That's why we had to find a criteria for approaching all subsystems gradually

to make reading interesting and easy-to-understand.

In the following image, you can find a rough representation of core JBoss AS 7 subsystems (for the sake of simplicity we are including just the subsystems that are covered throughout this book):



So, as a first taste of the application server, we will explore the areas that are highlighted in bold in this screenshot. These include the following core application server subsystems:

• The **Thread Pool** subsystem

• The **JBoss Logging** subsystem

Let's see each subsystem in a separate section.

**~~[ 34 ]~~**

*Chapter 2*

**Configuring the Thread Pool subsystem** Thread Pools address two different problems: they usually deliver improved

performance when executing large numbers of asynchronous tasks, due to reduced per-task invocation overhead, and they provide a means of bounding and managing the resources, including Threads, consumed when executing a collection of tasks.

In the earlier releases of the application server, the Thread Pool configuration was centralized in a single file or deployment descriptor. This approach was maintained up to the first snapshots of the new application server. Since the 7.0.0 CR1 release, the individual subsystems that use Thread Pools manage their own Thread configuration.

By appropriately configuring the Thread Pool section, you can effectively tune the specific areas that use that kind of Pool to deliver new tasks. The application server Thread Pool configuration can include the following elements:

• Thread factory configuration

• Bounded Threads configuration

• Unbounded Threads configuration

• Queueless Thread Pool configuration

• Scheduled Thread configuration

Let's see in detail each single element:

**Configuring the Thread factory**

A **Thread factory** (implementing java.util.concurrent.ThreadFactory) is an object that creates new Threads on demand. Using Thread factories removes hardwiring of calls to new Thread, enabling applications to use special Thread subclasses, priorities, and so on.

The Thread factory is not included by default in the server configuration as it relies on defaults, which you will hardly need to modify. Nevertheless, we will provide a sample configuration of it for the experienced user who requires complete control of the Thread configuration.

So, here's an example of a custom Thread factory configuration:

<thread-factory name="MyThreadFactory"

thread-name-pattern="My Thread %t"

group-name="dummy" />

And here are the possible attributes that you can use when defining a Thread factory.**~~[ 35 ]~~**

*Configuring the Application Server*

The name attribute is the name of the created Thread factory.

• The optional priority attribute may be used to specify the Thread priority of created Threads.

• The optional group-name attribute specifies the name of the Thread group to create for this Thread factory.

The thread-name-pattern is the template used to create names for Threads. The following patterns may be used:

**Pattern Output**

%% Emits a percent sign

%g Emits the per-factory Thread sequence number

%f Emits the global Thread sequence number

%i Emits the Thread ID

%G Emits the Thread group name

**Bounded Thread Pool**

A bounded Thread Pool is the most common kind of Pool used by the application server, as it helps prevent resource exhaustion by defining a constraint on the Thread Pool's size; the other side of the medal is that this kind of Pool is also the most complex to use. Its inherent complexity derives from the fact that it maintains both a fixed-length queue and two Pool sizes: a **core size** and a **maximum size**.

Each time a new task is submitted, if the number of running Threads is less than the core size, a new Thread is created. Otherwise, if there is room in the queue, the task is queued.

If none of these options are viable, the executor needs to evaluate if it can still create a new Thread. If the number of running Threads is less than the maximum size, a new Thread is created. Otherwise, the blocking attribute comes into play. If blocking is enabled, the caller blocks until room becomes available in the queue.

If blocking is not enabled, the task is assigned to the designated hand-off executor, if one is specified. In the absence of a designated hand-off, the task will be rejected.

**~~[ 36 ]~~**

*Chapter 2*

The following image summarizes the whole process, showing how all the pieces fit together:

Bounded Queue Thread Executor

new Thread

handoff executor

yes yes yes

threads

is there room

threads

~~no no no no~~

~~no~~

Task

Task Rejected

<core size

in queue?

~~<maximum size~~ ~~blocking enabled?~~ handoff executor?

yes yes

queue block

And here's a sample of bounded Thread Pools, which is included in the configuration:

<bounded-queue-thread-pool

name="jca-short-running" blocking="true">

<core-threads count="10" per-cpu="20"/>

<queue-length count="10" per-cpu="20"/>

<max-threads count="10" per-cpu="20"/>

<keepalive-time time="10" unit="seconds"/>

</bounded-queue-thread-pool>

This is a short description of each attribute:

**Attribute Description**

name Specifies the bean name of the created executor

allow-core timeout

Specifies whether core Threads may time out or not; if false, only Threads above the core size will time out

blocking Specifies whether the submitter Thread will block if no space is available in this executor

core-threads Specifies the core Thread Pool size, which is smaller than the maximum Pool size

max-threads Specifies the maximum Thread Pool size

queue-length Specifies the executor queue length

keepalive-time Specifies the amount of time that Threads beyond the core Pool size should be kept running, when idle

thread-factory Specifies the bean name of a specific Thread factory to use to create worker Threads

handoff executor

Specifies an executor to delegate tasks to in the event that a task cannot be accepted

**~~[ 37 ]~~**

*Configuring the Application Server*

**Performance focus**

**Queue size** and **Pool size** are two samples of performance trade-off for each other. When using a small Pool with a large queue, you cause to minimize the CPU usage, OS resources, and context-switching overhead. It can, however, produce an artificially low throughput. If tasks are

strongly I/O bound (and thus frequently blocked), a system may be able to schedule time for more Threads than you otherwise allow. Use of

small queues generally requires larger Pool sizes, which keeps the CPUs busier but may encounter unacceptable scheduling overhead, which also decreases throughput.

**Unbounded Thread Pool**

This other kind of Thread Pool executor follows a simpler (but more risky!) approach; that is, it always accepts new tasks.

In practice, the unbounded Thread Pool has a core size and a queue with no upper bound. When a task is submitted, if the number of running Threads is less than the core size, a new Thread is created. Otherwise, the task is placed in a queue. If too many tasks are allowed to be submitted to this type of executor, an out of memory condition may occur.

Unbounded Queue Thread Executor

new Thread yes

threads

no

too many task

Task yes Out of Memory

<core size

~~no~~

~~no~~

in queue? queue

Due to its inherent risk, unbounded Thread Pools are not included by default in the server configuration. We will provide a sample here, with only one recommendation: don't try this at home, kids!

<unbounded-queue-thread-pool name="unbounded-threads" > <max-threads count="10" per-cpu="20"/>

<keepalive-time time="10" unit="seconds"/>

</unbounded-queue-thread-pool>

If you want to know more about the meaning of each Thread Pool element, you can refer to the bounded Thread Pool table.

**~~[ 38 ]~~**

*Chapter 2*

**Queueless Thread Pool**

As its name implies, this is a Thread Pool executor with no queue. Basically, this executor short-circuits the same logic of the bounded Thread executor, as it does not attempt to store the task in a queue.

So, when a task is submitted, if the number of running Threads is less than the maximum size, a new Thread is created. Otherwise, if blocking is enabled, the caller blocks until another Thread completes its task and accepts the new one. If blocking is not enabled, the task is assigned to the designated hand-off executor, if one is specified. Without any designated hand-off, the task will be rejected.

Queueless Thread Executor

new Thread

handoff executor

yes yes

threads

~~no no~~

Task ~~no~~ Task Rejected <maximum size ~~blocking enabled?~~ handoff executor?

yes

block

Queueless executors are also not included by default in the configuration file; we will, however, provide a sample configuration here:

<queueless-thread-pool

name="queueless-thread-pool" blocking="true">

<max-threads count="10" per-cpu="20"/>

<keepalive-time time="10" unit="seconds"/>

</queueless-thread-pool>

**~~[ 39 ]~~**

*Configuring the Application Server*

**Scheduled Thread Pool**

The server-scheduled Thread Pool is used for activities on the server-side that require running periodically or with delays. It maps internally to a java.util. concurrent.ScheduledThreadPoolExecutor instance.

Scheduled Thread Executor

Thread

scheduled

yes

threads

Task ~~no~~ Task Rejected

<maximum size

This type of executor is configured with the scheduled-thread-pool executor element:

<scheduled-thread-pool name="remoting">

<max-threads count="10" per-cpu="20"/>

<keepalive-time time="10" unit="seconds"/>

</scheduled-thread-pool>

The scheduled Thread Pool is used by the remoting framework and by HornetQ subsystem, which uses both a bounded JCA Thread executor

and a scheduled Pool for delayed delivery.

**Configuring the application server logging** Every application needs to trace logging statements. At the moment, there are several implementations of logging libraries for Java applications, the most popular ones are:

• **Log4j**: It is a flexible open source logging library from Apache. Log4j is widely used in the open source community, and it was the default logging implementation on earlier releases of JBoss AS.

• **J2SE logging libraries** (**JUL**): It provides the logging classes and interfaces as part of the J2SE platform's standard libraries.

**~~[ 40 ]~~**

*Chapter 2*

Log4j and JUL are indeed very similar APIs. They differ conceptually only in small details, and in the end do more or less the same thing, except Log4j has more features that you may or may not need.

JBoss logging framework is substantially based on JUL, which is built around three main concepts: **loggers**, **handlers**, and **formatters**. These concepts allow developers to log messages according to their type and priority, to control where messages end up and how they look when they get there.

The following image shows the logging cycle using the JUL framework. **Applications** make logging calls on **Logger** objects. These **Logger** objects allocate LogRecord objects, which are passed to **Handler** objects for publication. Both the **Logger** and the **Handler** may use the **Formatter** to arrange the layout of logs and **Filter** to decide if they are interested in a particular LogRecord.

out

application

Logger Handler

Filter Filter Formatter

**Choosing your logging implementation** JBoss AS, through its releases, has used different frameworks to handle application server logs. In JBoss AS 5 and earlier, log4j was the default logging API used by the application server, and it was defined in the server/<server>/conf/jboss-log4j. xml file.

Since JBoss AS 6, the logging provider switched to JBoss's own implementation, which is based on the JDK 1.4 logging system. However, it provides several fixes or works around many serious problems in the default JDK implementation.

For example, the default implementation of java.util.logging provided in the JDK is too limited to be useful. A limitation of JDK logging is the inability to have per-web application logging, as the configuration is per-VM.

**~~[ 41 ]~~**

*Configuring the Application Server*

As a result, JBoss AS replaces the default JUL LogManager implementation with its own implementation, which addresses these shortcomings. The following image illustrates the modules that make up the JBoss AS 7 logging subsystem:

javax.api

org.jboss,logmanager

org.jboss.logging

org.jboss.as.logging

org.apache.log4j

jboss-logmanager-log4j

At the top of the hierarchy there's the org.jboss.logmanager module, which is the top-level library that manages logs for the jboss logging subsystem. Beneath the jboss logmanager, you can find the concrete implementations, such as the org.jboss.logging or the jboss-logmanager-log4j module. By default, the application server uses the former module (org.jboss.logging), which is implemented in its turn by org.jboss.as.logging to manage your logs inside the application server. However, if you want to switch to log4j implementation, the jboss-logmanager-log4j module is what you need (in the last section of this chapter, we will include an example of how to use log4j in your application).

**Configuring the logging subsystem**

The logging subsystem contains a set of log handlers out of the box. A handler object takes log messages from a logger and exports them. It might, for example, write them to a console or write them to a file, or send them to a network logging service, or forward them to an OS log, or whatever. By default, the following handlers are defined:

**~~[ 42 ]~~**

*Chapter 2*

**The console-handler**

The console-handler defines a handler, which simply writes log messages to the console.

<console-handler name="CONSOLE" autoflush="true">

<level name="INFO"/>

<formatter>

<pattern-formatter pattern="%d{HH:mm:ss,SSS} %-5p [%c] (%t) %s%E%n"/>

</formatter>

</console-handler>

The attributes of the console-handler are common to all other handlers. We will shortly describe their meaning here:

The optional autoflush attribute determines if buffered logs are flushed automatically. The default value for this option is true.

The following element, level, defines the log level associated with the handler, ranging from FINEST (lower level) to FATAL (highest value).

Then, the formatter element provides support for formatting LogRecords. The log formatting inherits the same pattern strings for layout pattern of log4j, which was in turn inspired by dear old C's printf function.

For an exhaustive list of logging formats, you can check the log4j documentation at http://logging.apache.org/log4j/1.2/apidocs/org/apache/log4j/ PatternLayout.html.

Here, we will just mention that %d{HH:mm:ss,SSS} outputs the date of the logging event using the conversion included in brackets.

• The string %-5p will output the priority of the logging event

• The string [%c] is used to output the category of the logging event • The string (%t) outputs the Thread that generated the logging event • The string %s outputs the log message

• Finally, the %n string outputs the platform-dependent line separator character or characters

**The periodic-rotating-file-handler**

The periodic-rotating-file-handler defines a handler that writes to a file, rotating the log after a time period derived from the given suffix string, which should be in a format understood by java.text.SimpleDateFormat.

**~~[ 43 ]~~**

*Configuring the Application Server*

Here's the definition of it:

<periodic-rotating-file-handler name="FILE" autoflush="true"> <level name="INFO"/>

<formatter>

<pattern-formatter pattern="%d{HH:mm:ss,SSS} %-5p [%c] (%t) %s%E%n"/>

</formatter>

<file relative-to="jboss.server.log.dir" path="server.log"/> <suffix value=".yyyy-MM-dd"/>

</periodic-rotating-file-handler>

This handler introduces the file element containing the path, which is the actual filename and its relative-to position. In our case, the relative position corresponds to the jboss.server.log.dir application server parameter.

With the default suffix configuration, logs are rolled at 12 PM. By

changing the SimpleDateFormat, you can also change the period when logs are rotated, for example, the suffix yyyy-MM-dd-HH will rotate the logs every hour.

**The size-rotating-file-handler**

The size-rotating-file-handler defines a handler that writes to a file, rotating the log after the size of the file grows beyond a certain point. It also keeps a fixed number of backups.

There's no size handler defined in the standard configuration. However, we can find out its basic configuration from the JBOSS\_HOME/docs/schema/jboss-as-logging. xsd file:

<size-rotating-file-handler name="FILESIZE" autoflush="true" > <rotate-size value="500k" />

<level name="INFO"/>

<formatter>

<pattern-formatter pattern="%d{HH:mm:ss,SSS} %-5p [%c] (%t) %s%E%n"/>

</formatter>

<file relative-to="jboss.server.log.dir" path="server.log"/> </size-rotating-file-handler>

**~~[ 44 ]~~**

*Chapter 2*

**The asynchronous handler**

The asynchronous handler is a composite handler, which attaches to other handlers to produce asynchronous logging events. Behind the scenes, this handler uses a bounded queue to store events. Every time a log is emitted, the asynchronous handler appends the log into the queue and returns immediately. Here's an example of asynchronous logging for the FILE appender:

<async-handler name="ASYNC" >

<level name="INFO" />

<queue-length>1024</queue-length>

<overflow-action>block</overflow-action>

<sub-handlers>

<handler-ref name="FILE" />

</sub-handlers>

</async-handler>

In this handler, we are also specifying the size of the queue where events are sent and the action to take when the async queue overflows. You can opt between **blocked** where the calling Thread is blocked, and **discard** where the message will be discarded.

**When should I use the asynchronous handler?**

The asynchronous handler produces a substantial benefit to applications, which are heavily I/O bound, since an asynchronous logging event might be fired at times where Threads are blocked on intensive I/O operations. At the opposite end, CPU-bound applications might not benefit at all

from asynchronous logging as it will put additional stress on the CPU.

**Custom handlers**

So far, we have seen just a few basic log handlers, which are usually included in your server configuration. If you need a more advanced approach to your logs, you can define a custom logging handlers. In order to add a custom handler, you need to define class that extends the java.util.logging.Handler interface and overrides its abstract methods. For example, this class named JdbcLogger is used to write the logs on a database storage (full code is available at: http://community.jboss.org/ wiki/CustomLogHandlersOn701).

public class JdbcLogger extends Handler{

@Override

public void publish(LogRecord record){

try{

insertRecord(record);

**~~[ 45 ]~~**

*Configuring the Application Server*

}

catch (SQLException e) {

e.printStackTrace();

}

}

@Override

public void flush() { . . . . }

@Override

public void close() { . . . . }

}

Once compiled, this class needs to be packaged in an archive (for example, logger. jar) and installed as a module into the application server. We will name the module com.JDBCLogger, which requires the following structure under the modules folder:

modules

com

JDBCLogger

Path to be created

main

logger.jar

module.xml

The label **Path to be created** shows off the directory structure under which we will place the **logger.jar** archive and its configuration file (**module.xml**), which follows here:

<module xmlns="urn:jboss:module:1.0" name="com.JDBCLogger"> <resources>

<resource-root path="logger.jar"/>

</resources>

<dependencies>

<module name="javax.api"/>

<module name="org.jboss.logging"/>

<module name="com.mysql"/>

</dependencies>

</module>

**~~[ 46 ]~~**

*Chapter 2*

Please notice that this module has a dependency on another module named

com.mysql. In the next chapter, we will show how to connect to a database by installing the appropriate module.

We are almost done. Now, insert the handler in the logging subsystem, which contains within its properties the database connection strings and the statement that will be used to insert logs into the database:

<custom-handler name="DB" class="com.sample.JdbcLogger" module="com.JDBCLogger">

<level name="INFO"/>

<formatter>

<pattern-formatter pattern="%d{HH:mm:ss,SSS} %-5p [%c] (%t) %s%E%n"/>

</formatter>

<properties>

<property name="driverClassName" value="com.mysql.jdbc.Driver"/> <property name="jdbcUrl"

value="jdbc:mysql://localhost:3306/mydb"/>

<property name="username" value="root"/>

<property name="password" value="admin"/>

<property name="insertStatement" value="insert into log\_table values (?, $TIMESTAMP, $LEVEL, $MDC[ip], $MDC[user], $MESSAGE, hardcoded)"/>

</properties>

</custom-handler>

<root-logger>

<level name="INFO"/>

<handlers>

<handler name="CONSOLE"/>

<handler name="FILE"/>

<handler name="DB"/>

</handlers>

</root-logger>

The new handler, named DB, is enlisted in the root-logger to collect all logging statements that have a priority of INFO or above. Before testing the logger, don't forget to create the required tables on your MySql database:

CREATE TABLE log\_table(

id INT(11) NOT NULL AUTO\_INCREMENT,

`timestamp` VARCHAR(255) DEFAULT NULL,

level VARCHAR(255) DEFAULT NULL,

mdc\_ip VARCHAR(255) DEFAULT NULL,

mdc\_user VARCHAR(255) DEFAULT NULL,

message VARCHAR(1500) DEFAULT NULL,

**~~[ 47 ]~~**

*Configuring the Application Server*

hardcoded VARCHAR(255) DEFAULT NULL,

PRIMARY KEY (id)

)

ENGINE = INNODB

AUTO\_INCREMENT = 1

If you have carefully followed all the required steps, you will notice that the log\_table contains the logging events that have been triggered since server startup:



**Configuring loggers**

A **Logger** object is used to log messages for a specific system or application components. Loggers are normally named, using a hierarchical dot-separated namespace. Logger names can be arbitrary strings, but they should normally be based on the package name or class name of the logged component. For example, the logger instructs the logging system to emit logging statements for the package com.sample, if they have the log level "WARN" or higher:

<logger category="com.sample">

<level name="WARN"/>

</logger>

At the top of the hierarchy, there's the root-logger. It is exceptional in two ways:

• It always exists

• It cannot be retrieved by name

In the default server configuration, the root-logger defines two handlers, which are connected to the CONSOLE and to the FILE handler:

<root-logger>

<level name="INFO"/>

<handlers>

<handler name="CONSOLE"/>

<handler name="FILE"/>

</handlers>

</root-logger>

**~~[ 48 ]~~**

*Chapter 2*

**Configuring log4j in your application**

So far, we have seen how to configure the application server logs operating on the main configuration file (standalone.xml). Chances are, however, that users will want to provide a log configuration on an application basis, using the widely adopted log4j framework. This section shows the simple steps needed to adopt log4j in your application.

Let's create a basic web application named LogExample. For this purpose, you can start a **New Dynamic Web** project from the Eclipse IDE. In order to configure log4j, we will need to provide a log4j configuration file, which, by default, is named log4j.properties or log4j.xml, and place it at the root of the Java sources (named src in Eclipse).

The following sample, log4j.properties, defines two appenders: the first one (stdout) prints messages on the console, while the second one(R) is connected with a RollingFileAppender:

log4j.rootLogger=warn, stdout, R

# stdout is set to be a ConsoleAppender.

log4j.appender.stdout=org.apache.log4j.ConsoleAppender

# stdout uses PatternLayout.

log4j.appender.stdout.layout=org.apache.log4j.PatternLayout

# Pattern to output the caller's file name and line number. log4j.appender.stdout.layout.ConversionPattern=%5p [%t] (%F:%L) - %m%n

# R is set to be a RollingFileAppender.

log4j.appender.R=org.apache.log4j.RollingFileAppender

log4j.appender.R.File=example.log

# Max file size is set to 100KB

log4j.appender.R.MaxFileSize=100KB

# Keep one backup file

log4j.appender.R.MaxBackupIndex=1

# R uses PatternLayout.

log4j.appender.R.layout=org.apache.log4j.PatternLayout

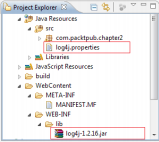
log4j.appender.R.layout.ConversionPattern=%p %t %c - %m%n log4j.logger.com.packtpub=DEBUG, stdout, R

**~~[ 49 ]~~**

*Configuring the Application Server*

As we said, this file will be placed at the root of your web project, so that when the project is built, it will be moved in the WEB-INF/classes folder of your web application that is visible to the application's classpath.

Additionally, place the log4j libraries in the WEB-INF/lib folder of your application. Here's how the Web application should look:



Now, you can add logging statements to your classes, and they will be intercepted by the console appender and by the file appender. For example, the following Servlet prints out the value of the System variable named myproperty, which has been added earlier in the server configuration file:

@WebServlet("/LoggerServlet")

public class LoggerServlet extends HttpServlet {

**private static org.apache.log4j.Logger logger =**

**org.apache.log4j.Logger.getLogger(LoggerServlet.class);** protected void doGet(HttpServletRequest request,

HttpServletResponse response) throws ServletException, IOException {

logger.info("System variable

myproperty="+System.getProperty("myproperty"));

PrintWriter out = response.getWriter();

out.println("The Servlet just logged.");

}

}

**~~[ 50 ]~~**

*Chapter 2*

**Why have we added log4j library to the application?**

If you try to deploy your application without the log4j library, you will find that the deployer raises a ClassNotFoundException on the classes using the log4j package. This can be a bit surprising, since log4j

libraries are indeed included in the application server modules.

The explanation to it is that JBoss AS 7 is not based any more on a

hierarchical class loader, but it is entirely based on module classloading. This means that in practice, each deployment unit is itself a module,

which is isolated from other modules, such as .jars that are included in the application server.

The first good news is that you will not face any more conflicts between the application classes and the server classes, also known with the

infamous epithet *classpath hell*.

The second good news is that you can easily override the default

(isolating) behavior by simply adding a dependency on other

modules installed. For example, in this case, you would need to add a

dependency on org.apache.log4j module so that log4j libraries

are automatically linked by your application. *"Chapter 6*, *Deploying*

*Applications on JBoss AS 7*, which is about deploying applications, covers all the steps required to solve classloader issues.

**Summary**

In this chapter, we've gone through the basics of the application server configuration, which is now composed by a single monolithic file that contains information about all the services installed.

It is important to stress again that the server configuration will be your main reference to get a full understanding on the new AS infrastructure, although it is recommended to use a management interface to modify parts of it.

So, after a detailed examination of each section, we have covered the Thread Pool configuration, which relies on the J2SE Thread Executor API to define a set of Pools that are used by the application server core services.

Next, we have discussed the JBoss logging framework, which is built around the **Java Util Logging** framework, addressing some known shortcomings of it. We have described how to customize the logging configuration and how to use, as an alternative, the well-known **log4j** framework in your applications.

In the next chapter, we will take a look at some core enterprise services configurations, such as datasource and messaging subsystems, which are the backbone of many enterprise applications.

**~~[ 51 ]~~**

Configuring Enterprise Services

This chapter completes the configuration of the application server, by adding a comprehensive description of the Java Enterprise services that can be run on top of the application server. Each service itself is a core subsystem, which can be included or removed, depending on the kind of applications you are delivering. Here, we will describe the most interesting ones, which have been increasingly adopted by the application server end-users, going in the following order of topics:

• Configure the database connectivity

• Configure the enterprise Java Bean container

• Configure the messaging service

• Configure the transaction service

**Configuring database connectivity** In any application server, you can configure database connectivity by adding datasources to your server configuration. Each datasource contains a pool of database connections that are reserved as soon as the server is started up. Applications acquire a database connection from the datasource by looking it up on the JNDI tree and then calling getConnection().

Connection result = null;

try {

Context initialContext = new InitialContext();

DataSource datasource =

(DataSource)initialContext.lookup("java:/MySqlDS");

*Configuring Enterprise Services*

**result = datasource.getConnection();**

}

catch ( Exception ex ) {

log("Cannot get connection: " + ex);

}

Once the connection is established, the application should call connection.close() as early as possible, which returns the database connection to the pool for other applications to use.

Earlier JBoss AS releases needed a well-known datasource configuration file (ending in –ds.xml), which had to be deployed in order to be used by applications. Since the release 7 of JBoss AS, you need to use a different approach, because of the modular nature of the application server.

Out of the box, the application server ships with the H2 open source database engine (http://www.h2database.com), which can be used for testing purposes because of its small footprint and its useful browser-based console.

However, a real world application requires an industry standard database, such as Oracle database or MySQL. In the following section, we will show how to configure a datasource for the MySQL database.

Basically, any database configuration requires a two-step procedure:

1. Installing the JDBC driver.

2. Adding the datasource to your configuration.

Let's see each section in detail:

**Installing the JDBC driver**

In JBoss AS 5 and 6, you used to install the JDBC driver into the common/lib folder of your server distribution. In the new modular server architecture, you have more than one option to install your JDBC driver. The first and recommended approach consists of installing the driver as a module.

In the section named *Installing the driver as a deployment unit*, we will

account for another approach, which is usually a bit faster. However, it has a few limitations.

As we have seen in the previous chapter, the procedure for installing a new module requires copying the .jar libraries in the appropriate modules path and adding a module.xml file, which declares the module and its dependencies.

**~~[ 54 ]~~**

*Chapter 3*

The actual path for the module will be: JBOSS\_HOME/modules/<module>/main

The main folder is where all key module components are installed. So, in our example, we will add the following units:

• JBOSS\_HOME/modules/com/mysql/main/mysql-connector-java-5.1.17- bin.jar

• JBOSS\_HOME/modules/com/mysql/main/module.xml

Here's an image of the module tree:

modules

com

mysql

Path to be created

main

mysql

connector.jar

module.xml

The JDBC driver used in this example, also known as Connector/J can freely be downloaded from the MySQL site: (http://dev.mysql.com/downloads/ connector/j/).

The module.xml contains the actual module definition. The most interesting part of it is the module name (com.mysql), which corresponds to the module attribute defined in the your datasource.

Next, you need to state the path to the JDBC driver resource and finally the module dependencies.

<module xmlns="urn:jboss:module:1.0" name="com.mysql">

<resources>

<resource-root path="mysql-connector-java-5.1.17-bin.jar"/> </resources>

<dependencies>

<module name="javax.api"/>

<module name="javax.transaction.api"/>

</dependencies>

</module>

**~~[ 55 ]~~**

*Configuring Enterprise Services*

**Adding a local datasource**

Once the JDBC driver is installed, you need to configure the datasource in the application server. In JBoss AS 7, you can configure two kind of datasources, **local datasources** and **xa-datasources**.

A local datasource does not support two phase commit using a java.

sql.Driver. On the other hand, an xa-datasource supports two phase commit using a javax.sql.XADataSource.

Adding a datasource definition can be completed in several ways: you can just add the datasource definition within the server configuration file, or you can use the management interfaces that will accurately do the work for you.

Showing all possible approaches in this section would maybe add too much information at once, so we will let you digest at first the most intuitive approach, that is cutting and pasting the datasource definition into your server configuration file. In the *Chapter 7*, *Managing the Application Server*, which is about server management, we will show the other available options in more detail.

So, here's a sample MySQL datasource configuration:

<datasources>

<datasource jndi-name="java:/MySqlDS" pool-name="MySqlDS\_Pool" enabled="true" jta="true" use-java-context="true" use-ccm="true"> <connection-url>

jdbc:mysql://localhost:3306/MyDB

</connection-url>

<driver>mysql</driver>

<pool />

<security>

<user-name>jboss</user-name>

<password>jboss</password>

</security>

<statement/>

<timeout>

<idle-timeout-minutes>0</idle-timeout-minutes>

<query-timeout>600</query-timeout>

</timeout>

</datasource>

<drivers>

<driver name="mysql" module="com.mysql"/>

</drivers>

</datasources>

**~~[ 56 ]~~**

*Chapter 3*

As you can see, the new configuration file borrows the same XML schema definition from the earlier -\*.ds.xml file, so it should not be difficult to migrate to the new configuration. Basically, you would define the connection path to the database using the connection-url and the JDBC driver class with the driver section.

Since JBoss AS 7.1.0, it's mandatory that the datasource is bound into the java:/ or java:jboss/ JNDI namespace.

The pool section can be used to define the JDBC Connection pool properties, leaving in this case to the default values. Then the security section lets you configure the connection credentials.

The statement section as well is added just as place holder for statement caching options.

The optional timeout section contains a set elements, such as the query-timeout, which is a static configuration of the maximum of seconds before a query times out. Also the included idle-timeout-minutes element indicates the maximum time a connection may be idle before being closed. Setting to 0 disables it. Default is 15 minutes.

**Configuring the connection pool**

One key aspect of the datasource configuration is the pool section. Strictly speaking, in order to use connection pooling, no configuration is required, because without any configuration JBoss AS will choose some default settings. However, if you want to customize how pooling is done, such as to control the size of the pools and which types of connections are pooled, you would be better learning about its available attributes.

Here's an example of pool configuration, which can be added to your datasource configuration:

<pool>

<min-pool-size>5</min-pool-size>

<max-pool-size>10</max-pool-size>

<prefill>true</prefill>

<use-strict-min>true</use-strict-min>

</pool>

**~~[ 57 ]~~**

*Configuring Enterprise Services*

The attributes included in the pool configuration are actually borrowed from earlier releases, so we include them here for your reference:

**Attribute Meaning**

min-pool-size The minimum number of connections in the pool (default 0 - zero) max-pool-size The maximum number of connections in the pool (default 20) prefill Attempt to pre-fill the connection pool to the minimum number of connections

use-strict-min Whether idle connections below the min-pool-size should be closed

**Configuring the statement cache**

For each connection in a connection pool in your system, JBoss AS Server is able to create a statement cache. When a prepared statement or callable statement is used on a connection, JBoss AS caches the statement so that it can be reused. In order to activate the statement cache, you have to specify a value of prepared-statement cache-size greater than 0:

<statement>

<track-statements>true</track-statements>

<prepared-statement-cache-size>10</prepared-statement-cache-size> <share-prepared-statements/>

</statement>

Notice, we have also included the track-statements to true in the statement section, which enable automatic closing of statements and ResultSets. This is important if you want to use prepared statement caching and/or don't want to leak cursors in your database.

The last element, share-prepared-statements, can be used only with prepared statement cache enabled and determine whether the two requests in the same transaction should return the same statement (default false).

**Adding an xa-datasource**

Adding an xa-datasource requires some tweaks in the datasource configuration. As a matter of fact, the connection information is now acquired as xa-datasource properties. Also the xa-datasource class needs to be specified in the driver section.

**~~[ 58 ]~~**

*Chapter 3*

In the following code, we are adding the equivalent configuration for our MySQL JDBC driver, which now is used to set up an xa-datasource:

<datasources>

<xa-datasource jndi-name="java:/XAMySqlDS" pool-name="MySqlDS\_Pool" enabled="true" use-java-context="true" use-ccm="true"> <xa-datasource-property name="URL">

jdbc:mysql://localhost:3306/MyDB

</xa-datasource-property>

<xa-datasource-property name="User">jboss

</xa-datasource-property>

<xa-datasource-property name="Password">jboss

</xa-datasource-property>

<driver>mysql-xa</driver>

</xa-datasource>

<drivers>

<driver name="mysql-xa" module="com.mysql">

<xa-datasource-class>

com.mysql.jdbc.jdbc2.optional.MysqlXADataSource

</xa-datasource-class>

</driver>

</drivers>

</datasources>

**A shortcut for installing a datasource** As we said at the beginning of the book, with the new release of the application server, every library is a module. Thus simply deploying the JDBC driver to the application server will trigger its installation.

When using this option, we will just copy the mysql-connector-java-5.1.17- bin.jar driver into the JBOSS\_HOME/standalone/deployments folder of your installation as shown in the following image:

JBOSS\_HOME

standalone

deployments

mysql

connector.jar

**~~[ 59 ]~~**

*Configuring Enterprise Services*

Once deployed, you need to add the datasource configuration to your server. You can do it in various ways. The most intuitive approach is pasting the datasource definition into the configuration file:

<datasource jndi-name="java:/MySqlDS" pool-name="MySqlDS\_Pool" enabled="true" jta="true" use-java-context="true" use-ccm="true"> <connection-url>

jdbc:mysql://localhost:3306/MyDB

</connection-url>

<driver>mysql-connector-java-5.1.17-bin.jar</driver>

<pool />

<security>

<user-name>jboss</user-name>

<password>jboss</password>

</security>

</datasource>

Alternatively, you can use the new Command Line Interface or the Web administration console to achieve the same result. *Chapter 7*, *Managing the Application Server*, details some practical examples of adding datasources using the management interfaces.

**What about domain deployment?**

In this chapter, we are discussing about the configuration of standalone servers and, as we said, the services configuration can be applied also to domain servers. Domain servers, however, don't have a specified folder which is scanned for deployment, rather the management interfaces are used to inject resources in the domain. *Chapter 5*, *Configuring a JBoss*

*AS Domain*, will detail all the steps to deploy a module when using a domain server.

**Choosing the right driver deployment strategy** At this point, you might wonder which is the best practice for deploying the JDBC driver. Actually, installing the driver as a deployment unit is an handy shortcut, however, it has a couple of limitations that might limit is usage within development bounds. At first, it requires a JDBC 4-compliant driver.

**~~[ 60 ]~~**

*Chapter 3*

Deploying a non JDBC 4-compliant driver is, however, possible and it just requires a simple patching procedure. Create a META-INF/services

structure containing the file java.sql.Driver. The content of the

file will be the driver name, for example, supposing you had to patch a MySQL driver, the content will be: com.mysql.jdbc.Driver

Once you have created your structure, you can update your JDBC driver with any zipping utility or the .jar command: jar -uf your-jdbc driver.jar META-INF/services/java.sql.Driver

Most current JDBC drivers are JDBC 4-compliant, although curiously, not everyone is recognized as such by the application server. The following table describes some of the most used drivers and their JDBC compliance:

**DB Driver JDBC 4 compliant Contains java. sql.Driver**

MySQL mysql-connector java-5.1.17-bin.jar

PostgreSQL postgresql-9.1-901. jdbc4.jar

Yes, though not recognized compliant by AS 7

Yes, though not recognized compliant by AS 7

Yes Yes

Oracle ojdbc6.jar / ojdbc5. jar

Yes Yes

Oracle ojdbc4.jar No No

As you can see, the most notable exception to the list of drivers is the older Oracle ojdbc4.jar, which is not JDBC 4-compliant and does not contain the driver info in META-INF/services/java.sql.Driver.

The second issue with driver deployment is related to the specific case of xa-datasources. As a matter of fact, by installing the driver as deployment means that the application server by itself cannot deduct the information about the xa-datasource class used in the driver. Since this information is not contained inside the META-INF/services, you are forced to specify information about the xa-datasource class for each xa-datasource you are going to create.

If you recall the earlier example, where we installed a driver as a module, the xa-datasource class information can be shared for all installed datasources.

<driver name="mysql-xa" module="com.mysql">

<xa-datasource-class>

com.mysql.jdbc.jdbc2.optional.MysqlXADataSource

</xa-datasource-class>

</driver>

**~~[ 61 ]~~**