C o m m u n i t y   E x p e r i e n c e   D i s t i l l e d

HornetQ Messaging Developer's Guide

Avoid being stung by JBoss HornetQ messaging service whether you're an existing user or a newcomer with this agile, fast-paced, example-rich guide

Piero Giacomelli

[PACKT] PUBLISHING open source* community experience distilled

HornetQ Messaging Developer's Guide

Avoid being stung by JBoss HornetQ messaging service whether you're an existing user or a newcomer with this agile, fast-paced, example-rich guide

Piero Giacomelli

BIRMINGHAM - MUMBAI

HornetQ Messaging Developer's Guide

First published: October 2012

Production Reference: 1181012

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

ISBN 978-1-84951-840-6

www.packtpub.com

Cover Image by Duraid Fatouhi (duraidfatouhi@yahoo.com)

Credits

Author
Piero Giacomelli

Reviewers
Gao Yong Hao
Jean-Pol Landrain
Víctor Romero

Acquisition Editor
Joanna Finchen

Lead Technical Editor
Ankita Shashi

Technical Editors
Farhaan Shaikh
Dominic Pereira
Veronica Fernandes

Copy Editors
Laxmi Subramanian
Insiya Morbiwala
Aditya Nair

Project Coordinator
Joel Goveya

Proofreader
Maria Gould

Indexer
Tejal Daruwale

Graphics
Aditi Gajjar

Production Coordinator
Prachali Bhiwandkar

Cover Work
Prachali Bhiwandkar

About the Author

Piero Giacomelli started playing with a computer back in 1986 when he received his first PC (a commodore 64). Despite his love for computers, he graduated in mathematics and entered the professional software industry in 1997 when he started using Java.

He has been involved in lots of software projects using Java, .NET, and PHP. He is a great fan of JBoss and Apache technologies but also uses Microsoft technologies without any moral issues.

He is married with two kids, so in his spare time, he regresses to his infancy to play with toys with his kids.

I would like to thank my family for supporting me during the exciting, but also stressful months during which I wrote this book.

My mother Milena always encouraged me, and my father in-law Roberto and mother in-law Angelia helped Michela and me with the kids anytime we needed it without asking why.

A special acknowledgement to Gao Yong Hao, who is one of the main contributors on the HornetQ project and helped me review this book. He is so mighty in Java programming that I am not even worthy to stoop down and untie the straps of his sandals. This book would not have seen light without his precious help.

Last but not least I would like to thank Theresa Chettiar, Joanna Finchen, Pramila Balan, Joel Goveya, and the entire Editorial team at Packt Publishing for helping me with all the editorial issues that a book needs. Their patience with a non-native English speaker greatly helped me feel followed and comfortable.

About the Reviewers

Gao Yong Hao (Howard) was born on 22nd June, 1969, Tianjin, China. He studied in Northwestern Polytechnic University (Xian, Shanxi Province), majoring in Electronic Engineering, from 1987 to 1991, and graduated with a Bachelor of Engineering degree. He has over 15 years of experience in software development, working with software companies both domestic and overseas. His expertise mainly focuses on enterprise level techniques, such as CORBA, J2EE, and enterprise application integration (EAI). In particular, he has in-depth knowledge of Transactions, Messaging, and Security. He is now a member of the HornetQ project, a Java-based high performance messaging system. It is also an open source project in the JBoss Community.

Mr. Gao currently works for RedHat Inc. as a Senior Developer mainly taking part in the development work of HornetQ and maintenance work of JBoss Messaging. Previously he has worked with various other prestigious companies like Borland, Singapore (where he worked in CORBA area) and IONA, Beijing (where he worked mainly on its EAI product and SOA product).

Jean-Pol holds a BSc degree in Software Engineering since 1998 with an orientation in network, real-time, and distributed computing. He gradually became a Software Architect with more than 14 years experience in object-oriented programming, in particular with C++, Java/JEE, various application servers, and related technologies.

He works for Agile Partner, an IT consulting company based in Luxembourg already dedicated as early as 2006 to the promotion, education, and application of agile development methodologies. In his current assignment, he participates in the selection and validation of tools and technologies targeting the development teams of the European Parliament.

He participated in the book reviews of ActiveMq in Action and of Spring in Action 1st edition for Manning Publications.

I would like to thank my fantastic wife, Marie, and my 6-year old daughter, Phoebe, for their daily patience regarding my passion for technology and the time I dedicate to it. I would also like to thank my friends and colleagues because a life dedicated to technology would be boring without the fun they bring to it. In particular, I want to thank Pierre-Antoine Grégoire and David Dossot for having thought about me when looking for my current job, and also for the inspiring examples that they are. Pierre-Antoine is the President of the Luxembourg Java user group, the YaJUG, and also a speaker at various IT conferences in Europe. David Dossot is one of the two authors of Mule in Action.

Víctor Romero currently works as a Software Architect for Grupo Zed. He started his career in the dot-com era and has been a regular contributor to open source software ever since. He is the co-author of Mule in Action, 2nd edition and he is project "despot" of the SpEL module for Mule. Although hailing from the sunny Malaga, Spain, he has been integrating the clouds from a skyscraper in New York City for ShuttleCloud and creating networks for the Italian government in Rome.

I remember the verse of a poem the author of which unfortunately I do not remember. The author stated that:

"We are under the light of these cold stars, these cold and whores stars"

to describe the senselessness of the everyday human being fight.

This book is dedicated to my wife. Michela, stars light could look cold, but you are the star whose light drives me even in my darkest hours and the positive motivating force within my life.

This book is also dedicated to my kids Davide and Enrico. My fights in the day to day trenches of adult existence against this world, have the only goal of leaving it as a better place for you.

Table of Contents

Preface

Any person involved in the field of software development has used and is comfortable with the expression "big data".

With the rise of the social network paradigm, new types of challenges have raised attention in the developer community. Only to make an example of the numbers involved, let us take a look at the infographics describing a normal day on Twitter (`http://econsultancy.com/it/blog/8049-10-twitter-infographics`). On average we have 1650 tweets per second. But when particular events occur like, the first TV debate between Obama and Romney, we have a peak of 8000 tweets per second. To manage such a huge amount of information, we need to rethink the way we exchange data, store data, and elaborate data. We think that this is only the dawn of this problem. With millions of new Internet users that gain Internet access both in the western and developing countries, any type of Internet software that will succeed, will need to deal with billions of messages that need to be exchanged and treated the right way.

Exchanging information in the form of short messages is becoming more and more important, so frameworks for doing this will be a key factor in software development.

In other words, this book will present the HornetQ framework and how to deal with it.

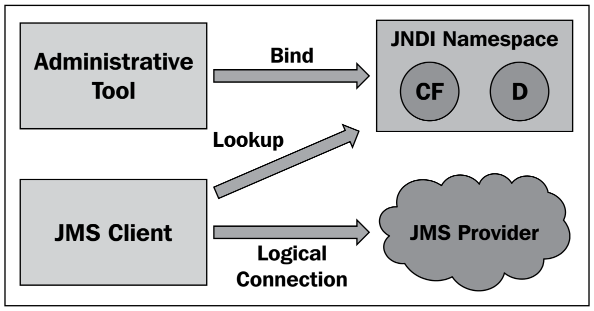
But before going on with the HornetQ environment we think that it is important to recall that HornetQ implements the **Java Message Service** (**JMS**) API. So many key concepts in HornetQ programming derive themselves directly from the JMS specification. So we will give a quick introduction on the JMS key concept architecture to allow the reader to feel more comfortable with the rest of the book.

The first specification for JMS was provided in 1999 and a standard for **Messaging Oriented Middleware** (**MOM**) was defined so a standard for exchanging messages between software in an enterprise environment was established.

Broadly speaking a JMS application should have the following parts:

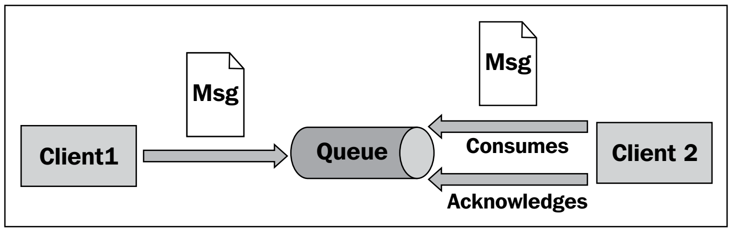
- JMS provider: This is a messaging system that implements the JMS interfaces and provides administrative and control features.
- JMS clients: These are the programs or components, written in the Java programming language, that produce and consume messages.
- Messages: These are the objects that communicate information between JMS clients.
- Administered objects: These are preconfigured JMS objects created by an administrator for the use of clients.
- Native clients: These are programs that use a messaging product's native client API instead of the JMS API. An application first created before the JMS API became available and that is subsequently modified is likely to include both JMS and native clients.

All these components act together in different scenarios, and mimic the purposes that software exchanging messages can have. We need these parts to interact as shown in the following diagram:



JMS implements (and so HornetQ) two different kind of approaches to exchange messages.

The first one is the point-to-point approach as illustrated in the following diagram:



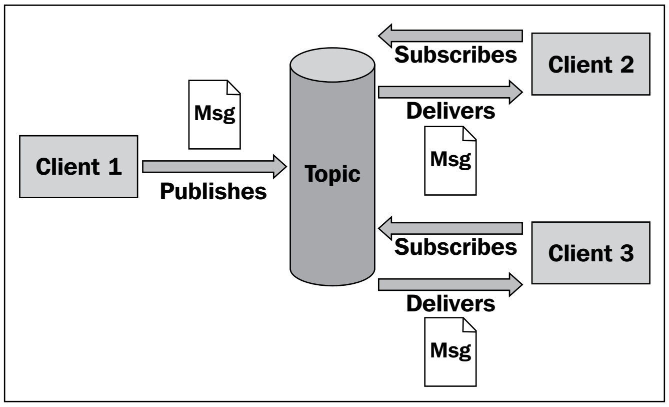
In this case the messages are sent from the clients to a queue where the consumer after the acknowledgement of the message consumes the message. In this case we have the following conditions:

- Each message has only one consumer
- A sender and a receiver of a message work in an asynchronous way

This approach is good when we have the condition where any message on the queue must be processed only by one consumer.

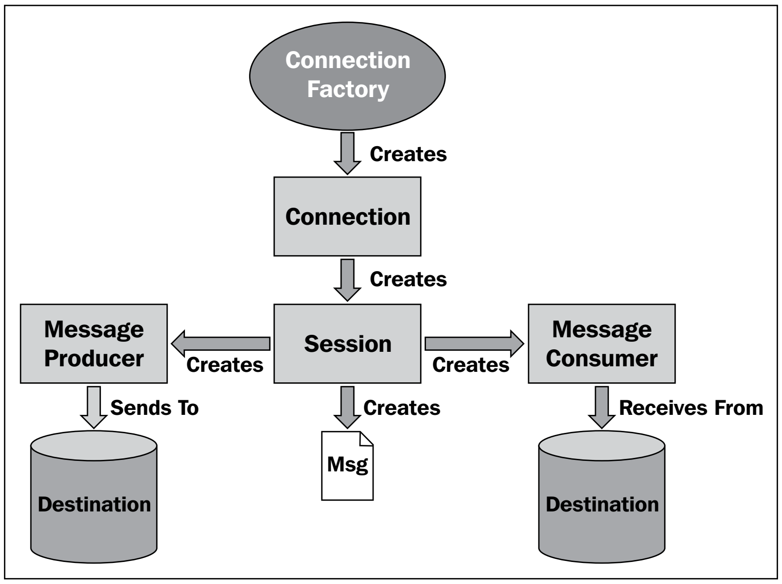
The other approach is the publish/subscribe one that is owned by the following properties:

- Each message may have multiple consumers.
- Publishers and subscribers have a timing dependency. A client that subscribes to a topic can consume messages published only after the client has created a subscription, and the subscriber must continue to be active in order for it to consume messages.



Surely the best way to use this approach is when there is the need that each message can be processed by more than one client.

Moving from this general architecture view to the programming API, we have the condition that the Java interface for a MOM must follow the schema, as shown in the following diagram:



So to produce/consume messages we need to have a **ConnectionFactory** that handles the **Connection**. The connection handles the **Session** both for the consumer and the producer that send/receive message in that session.

What this book covers

Chapter 1, *Getting Started with HornetQ*, introduces the reader with a full example of the HornetQ infrastructure and coding procedure using the JMS specifications.

Chapter 2, *Setting Up HornetQ*, covers the main configuration of a HornetQ standalone server. It also covers basic production and development set up both on Windows and Ubuntu machines.

Chapter 3, *Basic Coding with HornetQ*: *Creating and Consuming Messages*, provides you with a fully working example on how to create and consume messages in HornetQ using its internal core API.

Chapter 4, *Monitoring HornetQ*, introduces the reader on the management API provided to control the queues created within the HornetQ server. JMS management and JMX management will also be covered.

Chapter 5, *Some More Advanced Features of HornetQ*, goes into some more advanced features of JMS HornetQ specifications by considering the delivery of large messages and the management of undelivery messages.

Chapter 6, *Clustering with HornetQ*, provides some theory details and some configuration settings, which will help the reader in setting up a cluster and coding a consumer producer layer on it.

Chapter 7, *Divert and Filter Messages*, describes how to create some kind of automatic forwarding mechanism for messages from one HornetQ queue to another one on a different machine.

Chapter 8, *Controlling Message Flow*, describes some advanced configuration settings for creating and monitoring a high-frequency message production environment, as HornetQ is designed to manage billions of messages per second.

Chapter 9, *Ensuring your HornetQ Security,* introduces you to the HornetQ security mechanism and to the SSL connection configuration between the client and the server. Securing the way a client sends messages to a HornetQ server is a basic setting that a real application should have. This is the purpose of this chapter.

Chapter 10, *HornetQ in JBoss Environment*, introduces the reader on how to set up HornetQ within a JBoss application server and how to code servlets that use HornetQ Bean that can be used by servlets. HornetQ is the official Message-Oriented Framework for the JBoss application server.

Chapter 11, *More on HornetQ Embedding*, covers the theme of HornetQ being a set of POJO files so it can be embedded in any Java application. This chapter will also cover the Spring integration of HornetQ.

What you need for this book

This book is a developer-oriented one. So you need an IDE for developing purposes. We will create examples in this book that can be compiled using both Eclipse or NetBeans. The book's code can be equally used in other IDE such as IntelliJ with minor changes.

In the last chapter we will also use the Spring environment so the example has been coded using the Spring Tool Suite, so this last example will run only using that IDE.

If you are using the same machine for development purposes we also provide all the instructions to set up a running instance of a standalone HornetQ server, both on Windows and Linux systems in *Chapter 1, Getting Started with HornetQ*.

Who this book is for

This book is intended to provide a complete coverage of all the HornetQ features from a coder's point of view. If you are a first time Java coder or a senior developer experienced in Java programming who needs to acquire skills on HornetQ, then this book is for you.

The book will also cover some basic configurations to help the software developer understand how to properly configure a HornetQ standalone/cluster installation.

The examples can be followed by a first time user or an experienced one. We try to underline every passage within the code and the configuration task.

Conventions

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

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

A block of code is set as follows:

```
[default]
exten => s,1,Dial(Zap/1|30)
exten => s,2,Voicemail(u100)
exten => s,102,Voicemail(b100)
exten => i,1,Voicemail(s0)
```

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

```
[default]
exten => s,1,Dial(Zap/1|30)
exten => s,2,Voicemail(u100)
exten => s,102,Voicemail(b100)
exten => i,1,Voicemail(s0)
```

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

```
# cp /usr/src/asterisk-addons/configs/cdr_mysql.conf.sample
     /etc/asterisk/cdr_mysql.conf
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "clicking the **Next** button moves you to the next screen".

[Warnings or important notes appear in a box like this.]

[Tips and tricks appear like this.]

Reader feedback

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

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

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

Customer support

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

Downloading the example code

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

Errata

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

Piracy

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

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

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

Questions

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

1
Getting Started with HornetQ

This chapter will give us a brief introduction on how to use HornetQ in standalone mode, and will implement an example of message production and consumption using the JMS API. To cover this topic, we will also create a fully working development environment for the first-time user. We will cover the following topics:

- Downloading the software
- Installing and running HornetQ as a standalone server
- Installing the IDE for coding the example
- Installing the NoSQL database for storing the results

A quick introduction to HornetQ

HornetQ is an open source project for building multi-protocol, embeddable, very high performance, clustered, and asynchronous messaging systems. If we consider HornetQ from a higher level of abstraction, we could say that it is the Java implementation of a **Message Oriented Middleware** (**MOM**). Essentially, we could say that HornetQ is the Java implementation of a software built to send and receive messages in a distributed system. But, as it sometimes happens, this definition does not answer the main questions:

- What is HornetQ suitable for?
- Why should we adopt it in our code or in our infrastructure?

From a developer's perspective, we could state that HornetQ is a framework devoted to exchanging messages, which implement the JMS specifications. HornetQ can handle multiple message queues and provide a set of Java classes that can handle both the insertion and reading of these queues. Moreover, HornetQ provides classes for monitoring purposes, and by using the **Java EE Connector Architecture** (**JCA**), it can be integrated as the JMS messaging system for other application servers.

HornetQ implements the **Java Messaging System** (**JMS**) protocol along with its own protocol to exchange messages. It is highly configurable, even for clustered environments, and can be used natively inside the JBoss Application Server starting from Version 5 and onwards, or in standalone mode, and even in an embedded mode.

Introduction to our fictional example

There are some interesting fields of application for high-performance, messaging-oriented middleware such as finance, medicine, and, in general, whenever there is the need of exchanging a chunk of information in a very fast way.

Throughout this book, we will use the same fictional example that will help us to describe, chapter by chapter, the various functionalities of HornetQ both for software coding and framework configuration. The example may seem far from everyday software coding, with a database as a backend, and a frontend that could be a web interface or a GUI. However, the trickiness of the configuration and of its requirements will really help you see the advantages of using HornetQ to exchange messages.

Our fictional example deals with a software house that is developing a software called **HomeECG**. The main purpose of this software is to allow people to do an **Electrocardiography** (**ECG**) exam at home. An ECG device is able to detect and amplify the tiny electrical changes on the skin that are caused by the heart during contractions. Using such biophysical signals, a doctor is able to detect abnormal heart functionality. Every person who needs this capability at home will be equipped with a device that is able to trace the ECG, and send the data using an HTTP connection to a server inside a health care institution.

Describing how such a system works is outside the scope of this book, so we won't explain in detail all the requirements such a measurement device should have. For the purpose of our demo, the important thing is that we have many messages coming from different patients, which need to be stored in a queue and transformed, to be inserted in a more structured way inside a database system.

Considering this case, we also have to deal with some important performance issues arising from the fact that a normal ECG could take three different measurements ten times per second. So our HomeECG framework should be able to manage—if one hundred patients do the ECG for ten minutes, all at the same time—potentially, nearly two billion messages!

So we need a messaging framework that is able to process a large number of messages (such as ECG measures) in an asynchronous way and store them in the final database. Now that we have described our fictional software, we need to move onto a developer PC to see how to install and configure a full, testing environment.

Installation and configuration of the environment

All the code for setting up a development environment and installing it is available at `http://www.packtpub.com`, and can be downloaded.

As we do for recipes, we need to identify the ingredients:

- **Operating system**: All our examples have been developed using a 32-bit Windows 2008 Server R2, installed on a virtual machine. However, as HornetQ is based on Java, which is nearly system agnostic, all the examples can be carried out with some minor adjustments on a Linux-based OS. For this chapter, we will use Windows for our examples, but in *Chapter 2*, *Setting Up HornetQ*, we will also discuss in detail the installation procedure for a Linux environment.
- **Java development kit**: This is an obvious requirement. However, it's less obvious which version is compatible with HornetQ. While writing this book, I used JDK 1.7. *Chapter 2*, *Setting Up HornetQ*, will detail which version should be used, according to the framework specifications.
- **HornetQ binaries**: We have used the latest available stable release in this book—2.2.14. We only need to download the ZIP file from the HornetQ website (`http://www.jboss.org/hornetq/downloads`).
- **Eclipse Indigo for J2EE edition**: This is the most popular IDE for Java coding. NetBeans is another great IDE for working in Java; however, we will not be considering all the differences between the configuration tasks of the two IDEs. Each of this book's examples can also be implemented using NetBeans, even if we don't cover every single step of the configuration; it can be easily mimicked by the Eclipse IDE.
- **MongoDB**: This is the database system that will be used to show how to store a message from a HornetQ queue into a database. The choice of using a NoSQL database was done because Mongo is easy to set up and run, and is also easy to scale. For the interested reader, there is a Packt book called *PHP and MongoDB Web Development Beginner's Guide by Rubayeet Islam*. Coupled with Mongo, we also need the `jar` connector that allows data storage using Java.

In our example, we will download and use the standalone server, which means that we will run HornetQ without any interaction or dependencies with other frameworks or application servers. But the first HornetQ versions were developed to be fully integrated in JBoss Application Server, and in fact, HornetQ started with Version 2.0 because it was an evolved version of the JBoss JMS messaging framework. HornetQ can be easily integrated as the messaging framework for the JBoss Application Server. It can also be easily embedded in Java code, giving the developer the possible advantages of a messaging framework created and instantiated directly from the Java code.

A word on the OS

HornetQ has been developed using Java so it can run on both Linux and Windows systems. A small portion of native code has been used for allowing HornetQ to use the Linux AIO feature. However, this is an optional feature, which we will not use in this book. We will cover every configuration on a Linux OS in depth. For testing purposes, we will use Ubuntu 12.04 (codename Precise Pangolin) and detail all the differences between running HornetQ on Windows and Linux systems.

Downloading the software

The HornetQ stable version does not follow high-frequency releases, but you can find them all grouped on the HornetQ download page located at `http://www.jboss.org/hornetq/downloads`. Eclipse IDE for J2EE development, the Indigo version can be found at `www.eclipse.org`. Finally, the MongoDB version can be found at `www.mongodb.org/downloads` and the corresponding API for Java development can be found at `http://www.mongodb.org/display/DOCS/Drivers`.

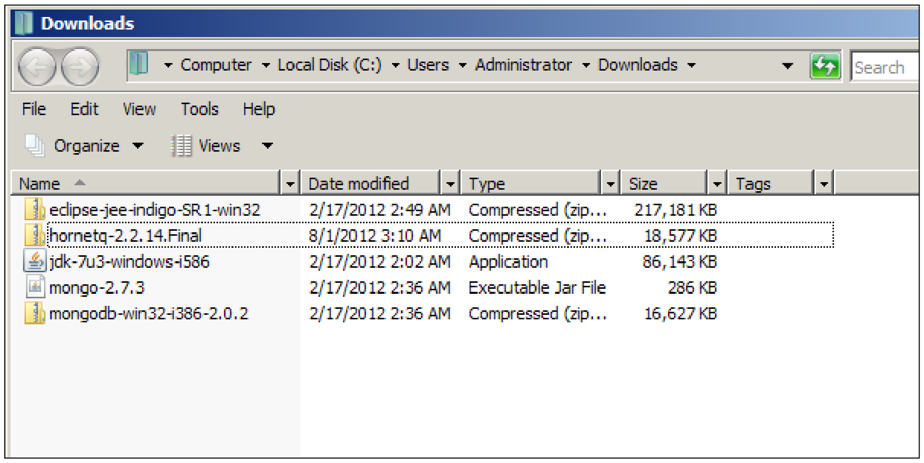
Installing HornetQ

We are going to install a HornetQ, non-clustered standalone server by carrying out the following steps:

1. Installing JDK from `http://www.Java.net/`. In a production environment, only JRE is needed, but we will use the JDK for compiling our example code.
2. Download HornetQ binaries as an archive from `http://www.jboss.org/hornetq`.
3. Unzip the archive.
4. Go to the `bin` folder of the unzipped archive.
5. Launch a script.

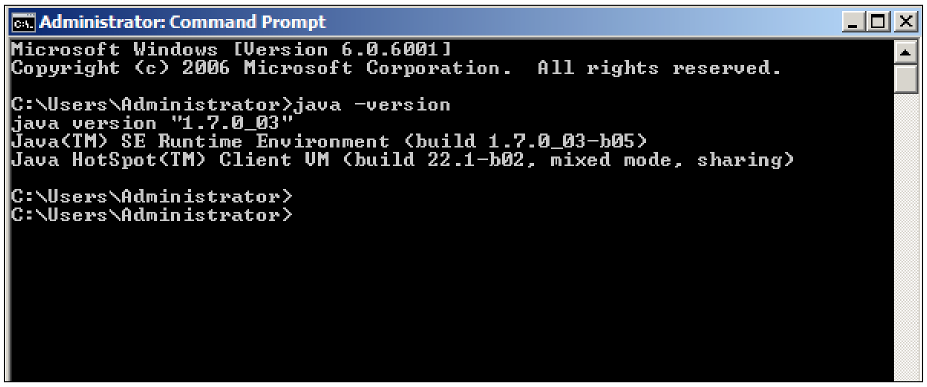
For our example, we will download the binaries directly so we have a HornetQ standalone server that can be launched directly from the command-line prompt. It is also possible to download the source code in an anonymous way and compile it from various sources. We will cover that possibility in *Chapter 2*, *Setting Up HornetQ*, as it is required for Maven to be configured and installed on the system. As HornetQ is based on Java code that is, with some minor changes, portable from Windows OS to Linux systems, you can download the same binary on a Linux platform and run the standalone version by using a simple bash script.

After downloading all the software from the previous list into a folder, you should have all the files displayed as shown in the following screenshot:


Downloads
Computer ▸ Local Disk (C:) ▸ Users ▸ Administrator ▸ Downloads
Search
File Edit View Tools Help
Organize Views

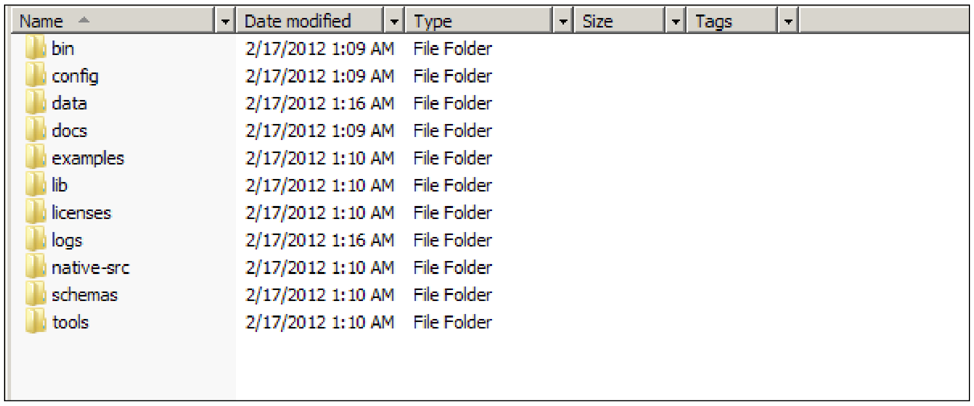
Name	Date modified	Type	Size	Tags
eclipse-jee-indigo-SR1-win32	2/17/2012 2:49 AM	Compressed (zip...	217,181 KB	
hornetq-2.2.14.Final	8/1/2012 3:10 AM	Compressed (zip...	18,577 KB	
jdk-7u3-windows-i586	2/17/2012 2:02 AM	Application	86,143 KB	
mongo-2.7.3	2/17/2012 2:36 AM	Executable Jar File	286 KB	
mongodb-win32-i386-2.0.2	2/17/2012 2:36 AM	Compressed (zip...	16,627 KB	

First of all, we need to install the JDK by double-clicking on the `.exe` application, and once finished, open a command prompt and enter the command `Java –version`. The output should be similar to the one in the following screenshot:



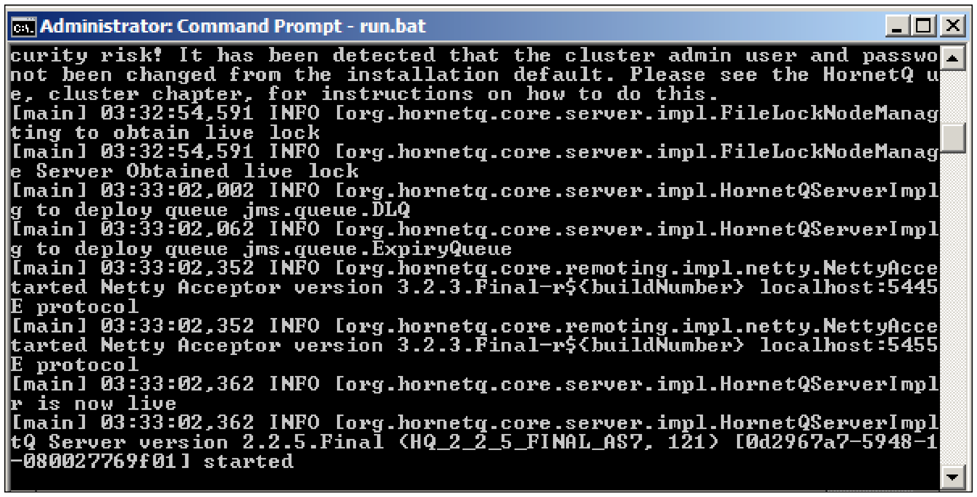
```
Microsoft Windows [Version 6.0.6001]
Copyright (c) 2006 Microsoft Corporation.  All rights reserved.

C:\Users\Administrator>java -version
java version "1.7.0_03"
Java(TM) SE Runtime Environment (build 1.7.0_03-b05)
Java HotSpot(TM) Client VM (build 22.1-b02, mixed mode, sharing)

C:\Users\Administrator>
C:\Users\Administrator>
```

Now, we are ready to install the HornetQ standalone server by simply unzipping the `HornetQ2.2.5.final.zip` to the `C:\` and renaming it to `hornetq`. You can unzip it wherever you want, so to avoid any path-related issue from now on, we will refer to `HORNETQ_ROOT` as the location containing the unzipped folder. In our case, `HORNETQ_ROOT` will refer to `c:\hornetq` as shown in the following screenshot:



Name	Date modified	Type	Size	Tags
bin	2/17/2012 1:09 AM	File Folder		
config	2/17/2012 1:09 AM	File Folder		
data	2/17/2012 1:16 AM	File Folder		
docs	2/17/2012 1:09 AM	File Folder		
examples	2/17/2012 1:10 AM	File Folder		
lib	2/17/2012 1:10 AM	File Folder		
licenses	2/17/2012 1:10 AM	File Folder		
logs	2/17/2012 1:16 AM	File Folder		
native-src	2/17/2012 1:10 AM	File Folder		
schemas	2/17/2012 1:10 AM	File Folder		
tools	2/17/2012 1:10 AM	File Folder		

Now, we are ready to start the HornetQ standalone server by entering the `HORNETQ_ROOT\bin` folder and double-clicking on the `run.bat` file. This should open a command prompt that, after some logging messages, will show a last line saying **HornetQ server started** as shown in the following screenshot:



```
curity risk! It has been detected that the cluster admin user and passwo
not been changed from the installation default. Please see the HornetQ u
e, cluster chapter, for instructions on how to do this.
[main] 03:32:54,591 INFO [org.hornetq.core.server.impl.FileLockNodeManag
ting to obtain live lock
[main] 03:32:54,591 INFO [org.hornetq.core.server.impl.FileLockNodeManag
e Server Obtained live lock
[main] 03:33:02,002 INFO [org.hornetq.core.server.impl.HornetQServerImpl
g to deploy queue jms.queue.DLQ
[main] 03:33:02,062 INFO [org.hornetq.core.server.impl.HornetQServerImpl
g to deploy queue jms.queue.ExpiryQueue
[main] 03:33:02,352 INFO [org.hornetq.core.remoting.impl.netty.NettyAcce
tarted Netty Acceptor version 3.2.3.Final-r${buildNumber} localhost:5445
E protocol
[main] 03:33:02,352 INFO [org.hornetq.core.remoting.impl.netty.NettyAcce
tarted Netty Acceptor version 3.2.3.Final-r${buildNumber} localhost:5455
E protocol
[main] 03:33:02,362 INFO [org.hornetq.core.server.impl.HornetQServerImpl
r is now live
[main] 03:33:02,362 INFO [org.hornetq.core.server.impl.HornetQServerImpl
tQ Server version 2.2.5.Final (HQ_2_2_5_FINAL_AS7, 121) [0d2967a7-5948-1
-080027769f01] started
```

So there you are. You now have a full, HornetQ standalone server running on your machine, waiting to dispatch your messages.

Installing HornetQ on Linux/Unix/Mac the easy way

In *Chapter 2*, *Setting Up HornetQ*, we will see how to set up the same server on Linux and consider which is the right version for your production purposes. However, if you have a Linux test machine (physical or virtual) and would like to try to install the HornetQ standalone server on it without going through all the details, here are the commands to execute it:

```
wget http://downloads.jboss.org/hornetq/hornetq-2.2.5.Final.tar.gz
tar –xvzf hornetq-2.2.5.Final.tar.gz
mv hornetq-2.2.5.Final hornetq
cd hornetq
chmod 777 run.sh
./run.sh
```

All you should see is the HornetQ logging messages summarizing your first successful HornetQ Linux startup.

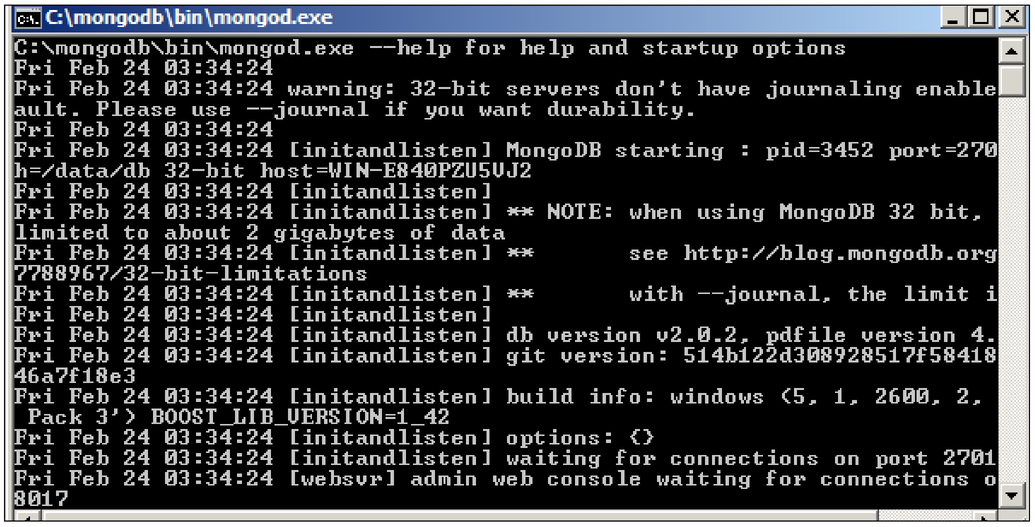
Do not forget that as we will see the HornetQ JMS queue accept connection from port 1099, this port should be opened for the incoming TCP connection. This is the default port for the JNDI lookup, so it is needed for the lookup of the JMS `ConnectionFactory`.

Installing Eclipse

Now we have two more installation steps to complete before coding our first example. Installing Eclipse Indigo for J2EE development is not a hard task; again, we only need to unzip the ZIP file downloaded from the Eclipse website (`http://www.eclipse.org/downloads`) to the main hard disk (`C:\`), and rename the folder as `eclipse`. Once installed, you only need to go to the folder and double-click on the file `eclipse.exe`. The second step is installing and configuring MongoDB.

Installing and configuring MongoDB

Lastly, we need to install MongoDB. MongoDB only needs to create the folder where the database file will be created and saved. The default data folder for a Windows environment is `C:\data\db`. So, prior to unzipping the MongoDB ZIP file to your hard disk and renaming the folder to `mongodb`, you should create the data folder on `C:\` and the `db` folder inside the `C:\data\` folder. To launch MongoDB, go to the `c:\mongodb` folder and double-click on the `mongod.exe` file. As a result, you should see a command prompt window as shown in the following screenshot:



```
C:\mongodb\bin\mongod.exe --help for help and startup options
Fri Feb 24 03:34:24
Fri Feb 24 03:34:24 warning: 32-bit servers don't have journaling enable
ault. Please use --journal if you want durability.
Fri Feb 24 03:34:24
Fri Feb 24 03:34:24 [initandlisten] MongoDB starting : pid=3452 port=270
h=/data/db 32-bit host=WIN-E840PZU5VJ2
Fri Feb 24 03:34:24 [initandlisten]
Fri Feb 24 03:34:24 [initandlisten] ** NOTE: when using MongoDB 32 bit,
limited to about 2 gigabytes of data
Fri Feb 24 03:34:24 [initandlisten] **       see http://blog.mongodb.org
7788967/32-bit-limitations
Fri Feb 24 03:34:24 [initandlisten] **       with --journal, the limit i
Fri Feb 24 03:34:24 [initandlisten]
Fri Feb 24 03:34:24 [initandlisten] db version v2.0.2, pdfile version 4.
Fri Feb 24 03:34:24 [initandlisten] git version: 514b122d308928517f58418
46a7f18e3
Fri Feb 24 03:34:24 [initandlisten] build info: windows (5, 1, 2600, 2,
 Pack 3') BOOST_LIB_VERSION=1_42
Fri Feb 24 03:34:24 [initandlisten] options: {}
Fri Feb 24 03:34:24 [initandlisten] waiting for connections on port 2701
Fri Feb 24 03:34:24 [websvr] admin web console waiting for connections o
8017
```

If you are using a Linux system, we suggest you refer to the tutorial on the MongoDB site at `http://docs.mongodb.org/manual/tutorial/install-mongodb-on-debian-or-ubuntu-linux/`.

You have now reached the end of the setup procedure. You have successfully installed:

- HornetQ2.2.14-final, the standalone server version
- Eclipse Indigo for the J2EE edition, as the IDE for developing our code
- MongoDB for storing our message after consumer parsing

So you are now ready for your first code example with HornetQ.

Configuring the Eclipse IDE for HornetQ

Following our fictional scenario, we now need to code the production and consumption of the message containing the ECG measurements that came from the patient's ECG device. Our first example will be done in a single Java class, but considering that most of the time you have two different softwares that write/read on HornetQ, a more realistic example will have two classes; one that writes messages to the queue and one that reads those messages from the queue. HornetQ implements the JMS specification, so we will use JMS messages for this example; in the next chapter, we will describe other ways to implement the messages. We will now describe the methods of the Java class that reflect the flow of the messages in our example. The next coding steps involve the implementation of these methods. We named our class `ECGMessageProducerConsumerExample.java`; once created, we will call the following methods:

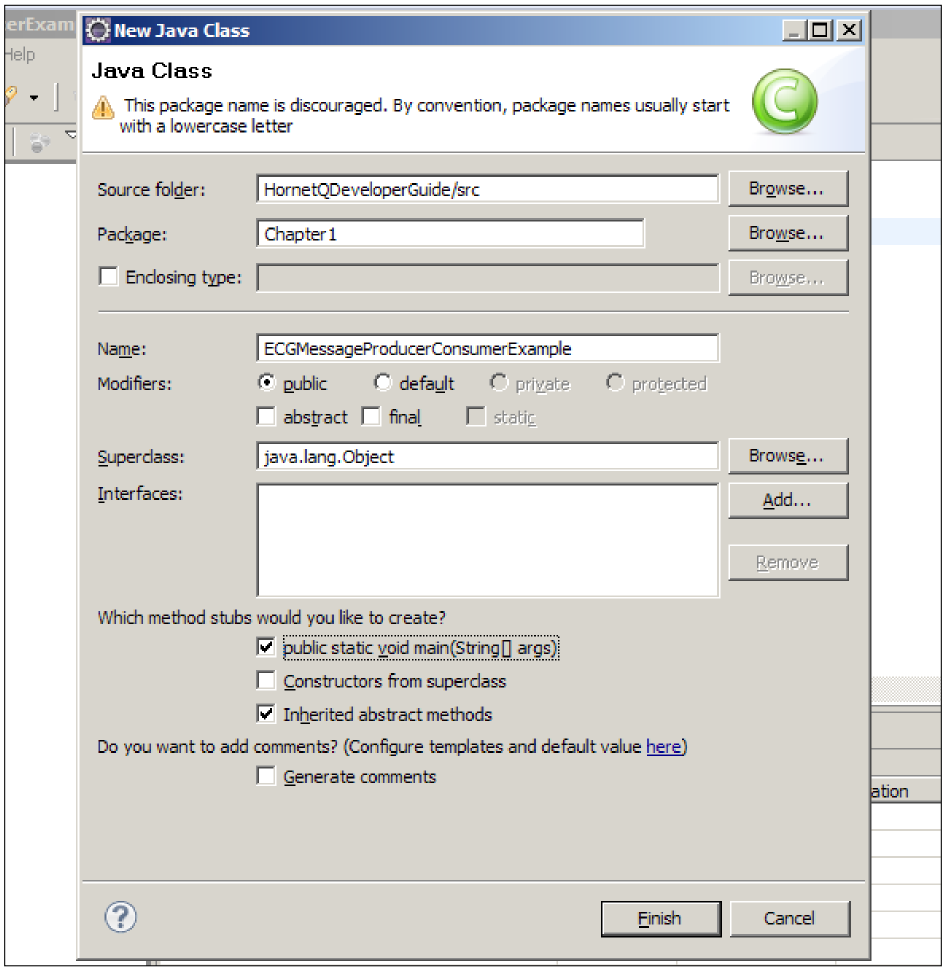
- `getInitialContext`: This method will initialize the properties that will identify the HornetQ server and queue
- `connectAndCreateSession`: This method is used for creating a shared connection and session from both the producer of the message and the consumer of the message
- `produceMessage`: A JMS message will be produced in this method and put on the queue
- `consumeMessage`: This method will read the message just stored, and by connecting to MongoDB, store it in the database
- `closeConnection`: In the end, this method does a clean closing of the resources used

Before entering the coding phase, we need to set up Eclipse to be able to deal with all the HornetQ API that are needed from `ECGMessageProducerConsumerExample.java`, to talk to the server.

So after firing up Eclipse, follow these steps to create the project for our class:

1. Select **File** | **New** | **Java Project** so that a new window appears asking for the project name, which is `chapter01`, and then click on **Finish**.
2. Right-click on the project name and choose the **Properties** item.
3. Once the **Properties** window is shown, choose **Java Build path** | **Libraries**, and click on the **Add external jars** button.
4. Go to the `HORNETQ_ROOT\lib` folder and select the following `.jar` files—`hornetq-jms.jar`, `jboss-jms-api.jar`, `jnp-client.jar`, `netty.jar`, and `hornetq-core-client.jar`. Then click on **OK**.

5. Click on the **add external jars** button again and go to where you have downloaded the `mongodb jar` from the MongoDB website, add it to your build path, and close the **Project properties** windows.
6. Right-click on the project name and choose **New | Class**. In the displayed pop-up window, enter `ECGMessageConsumerProducerExample` as the class name and check the **public static void main (string[] args)** checkbox, as shown in the following screenshot:

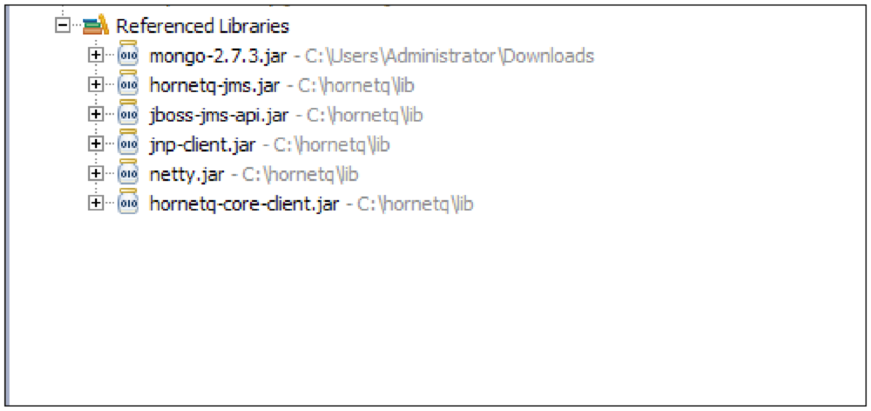


This package name is discouraged. By convention, package names usually start with a lowercase letter

Source folder: HornetQDeveloperGuide/src
Package: Chapter1
Enclosing type: [ ]
Name: ECGMessageProducerConsumerExample
Modifiers: [✓ public] default private protected
[ ] abstract [ ] final [ ] static
Superclass: java.lang.Object
Interfaces:

Which method stubs would you like to create?
[✓] public static void main(String[] args)
[ ] Constructors from superclass
[✓] Inherited abstract methods
Do you want to add comments? (Configure templates and default value here)
[ ] Generate comments

Most of our examples will be implemented using this standard configuration, so remember to re-use it any time you need to code a project for HornetQ. If all the previous steps have been done correctly, in the **Explorer** tab of Eclipse, you should see the `.jar` files included, as shown in the following screenshot:



```
Referenced Libraries
  mongo-2.7.3.jar - C:\Users\Administrator\Downloads
  hornetq-jms.jar - C:\hornetq\lib
  jboss-jms-api.jar - C:\hornetq\lib
  jnp-client.jar - C:\hornetq\lib
  netty.jar - C:\hornetq\lib
  hornetq-core-client.jar - C:\hornetq\lib
```

Coding our first example

Now that we are ready for the coding phase, as mentioned previously, we only need to implement the various methods that we have listed. Moving from this, we need to code the following steps:

1. Adding class fields.
2. Initializing the HornetQ environment.
3. Creating a connection, session, and looking for a JMS queue.
4. Creating a message and sending it to the queue.
5. Receiving the message sent in the previous step.
6. Saving to MongoDB.
7. Closing all connections and freeing up resources.

Every time you need to access a HornetQ queue, you need to create a connection as for a database, and create a session. If you access a queue as a message producer or as a message consumer, you need to create a connection and a session in it. All the configuration options that are needed to identify the queue or manage the connections will be described in *Chapter 2, Setting Up HornetQ*. For the moment, we will only describe how to create the connection and session. In our example class, both the session and connection are shared between the message producer and message consumer, so we will instantiate them only once. But in most of the cases, if you decide to write separate classes that access the queue from the consumer or the producer layer, you have to configure and open the session in both, separately.

Class fields

In our example, we will avoid the `import` directive and use objects identified by their fully qualified namespace, so that the reader can see where the objects are located in the namespace. The `import` statements do not affect runtime performance even if they affect compile-time ones, but using a wildcard, in our opinion, affects core readability.

So let us start with the fields needed through the various methods:

```
javax.naming.Context ic = null;
 javax.jms.ConnectionFactory cf = null;
 javax.jms.Connection connection =  null;
 javax.jms.Queue queue = null;
 javax.jms.Session session = null;

 com.mongodb.Mongo m;
 com.mongodb.DB db;

 String destinationName = "queue/DLQ";
```

Downloading the example code

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

As you can see, we need to have a `Context` object that is responsible for mapping names to the objects, because the discovery of the queue name and other configuration options is done using JNDI in a `.xml` configuration file.

We also have a `ConnectionFactory` object that is able to create multiple connections to HornetQ.

The `connection` is an object that will be used from the message producer layer and message consumer layer to send/receive messages.

The `queue` is an object that maps the queue defined in HornetQ configuration files. The `DLQ` queue is a particular one that we will use in this example, only to simplify the code. We will see how to create our own queues in the following chapters.

Finally, we have the `Mongo` and `db` objects that are responsible for connecting and storing data in the MongoDB database.

The `destinationName` string variable stores the queue name. The name should be exactly the one above otherwise our example will not work.

You are probably asking yourself where all these configuration parameters are stored. We will describe them in depth in *Chapter 2*, *Setting Up HornetQ*; for now, all we can say is that you can mimic JBoss configuration files. Also, HornetQ comes with a set of XML-based configuration files that are stored in the `HORNETQ_ROOT\config` folder.

Initializing the context

Now, we are ready to code the `GetInitialContext` method. This method is implemented in the following way:

```
java.util.Properties p = new java.util.Properties();

p.put(javax.naming.Context.INITIAL_CONTEXT_FACTORY,
"org.jnp.interfaces.NamingContextFactory");

p.put(javax.naming.Context.URL_PKG_PREFIXES,
"org.jboss.naming:org.jnp.interfaces");

p.put(javax.naming.Context.PROVIDER_URL, "jnp://localhost:1099");

ic = new javax.naming.InitialContext(p);
```

We instantiate a `Properties` object to store three key/value properties that are needed by `Context` to identify where to find the `.xml` files that contain the information for the HornetQ instance that we are going to use. The most significant file, for the moment, is the third one that tells us where the HornetQ server is running. By default, when it is started without other configuration issues, the HornetQ server is running on localhost and starts a Java naming provider on port 1099. As you can see these values (hostname and port) are mapped as the value for the provider_url object.

When `InitialContext` is created as an object, it can throw a runtime `javax.naming.NamingException`, which should be managed in case something goes wrong; for example, the name/IP address of the Java naming provider server could be wrong or the port may not be accessible from other machines due to the firewall configuration. In our case, everything should work fine considering that we are running everything from the same machine.

Now that we have our `InitialContext` object correctly assigned and created, we are ready to open the real connection to the HornetQ server.

Creating and opening the connection

This is done by the `connectAndCreateSession` method. The following code shows how to arrange the connection:

```
cf = (javax.jms.ConnectionFactory)ic.lookup("/ConnectionFactory");
queue = (javax.jms.Queue)ic.lookup(destinationName);
connection = cf.createConnection();
session = connection.createSession(false, javax.jms.Session.AUTO_
ACKNOWLEDGE);
connection.start();
```

First, we use a lookup on the context that is mapped to the `hornetq-jms.xml` file inside the `HORNETQ_ROOT\config` folder containing the configuration properties to identify the entry `/ConnectionFactory`. How JNDI queries the naming context to find the value is out of the scope of this book, but from a higher level of abstraction, it uses a tree directory-like structure. The `ConnectionFactory` object uses Netty, an asynchronous, event-driven application framework that supports lots of protocols, such as FTP, SMTP, HTTP, and SSL.

After the `ConnectionFactory` object has been created, we can also create the queue to store the messages. Using a lookup, we search for the queue whose name is `queue/DLQ` as defined in the class fields. It is important to notice that we use such a name because it was one of the default queues created by the standalone HornetQ server. If you take a look at the HornetQ default log stored in the `HORNETQ_ROOT\logs` folder, created when launching the `run.bat` script, you should see some lines like the following ones:

```
[main] 06:51:25,013 INFO trying to deploy queue jms.queue.DLQ
[main] 06:51:25,053 INFO trying to deploy queue jms.queue.ExpiryQueue
```

So the server that you started created two queues—one named `DLQ` and the other one named `ExpiryQueue`.

Now, you are ready to create the connection by calling the `createConnection()` method of `ConnectionFactory`. Before opening the connection, we will create a single session on it. Creating the session is pretty easy, but we need to describe the two parameters needed. The first one is a boolean value that tells if the session is transacted or not. This means that if you choose the session to be transacted, the delivery of the messages within that specific transaction will be made once you call the `Commit()` method of that session; so, this could potentially cause a significant overhead on the HornetQ server. In this case, we choose "not transacted" messages for easiness. The second parameter tells us how the messages are delivered in that session.

The possible choices are:

- **Client_Acknowledge mode**: This is not a feasible option (when you have the freedom to choose from the other two options) since the JMS server cannot send subsequent messages till it receives an acknowledgement from the client.
- **Auto_Acknowledge mode**: This mode follows the policy of delivering the message once and only once, but it incurs an overhead on the server to maintain this policy.
- **Dups_Ok_Acknowledge**: This mode has a different policy of sending the message more than once, thereby reducing the overhead on the server (imposed when using the Auto_Acknowledge mode), but imposing an overhead on the network traffic by sending the message more than once. The Auto_Acknowledge mode cleans up resources early from the persistent storage/memory, due to which the overhead is reduced.
- **Session_Transacted mode**: This mode should be used when the session is transacted.

In summary, the Client_Acknowledge mode or Dups_Ok_Acknowledge mode give a better performance than the Auto_Acknowledge mode.

Apart from the specific parameter, it is always good to remember that the session is specific to the connection; so once opened and used, it is good practice to close it before closing the connection.

Finally, we are able to start the connection with the specific session inside it. So we are now ready to move on to the production of our first message. ECG observation can be stored using ANSI protocols like HL7. This is too specific to our purpose, so we code a simple string that contains the patient's unique identifier, which was the date when it was recorded and the three ECG signals that were recorded.

For example, propose that we use a string that is like a line of a **CSV** (**Comma Separated Value**), so one single measurement will be a string like the following one:

```
1;02/20/2012 14:01:59.010;1020,1021,1022
```

The string is pretty self explanatory, so we move on to the `createMessage` method.

Producing the message

We are now ready to create our first JMS message and store it on `queue/DLQ`.

```
String theECG = "1;02/20/2012 14:01:59.010;1020,1021,1022";
javax.jms.MessageProducer publisher = session.createProducer(queue);
javax.jms.TextMessage message =
session.createTextMessage(theECG);
publisher.send(message);
System.out.println("Message sent!");
publisher.close();
```

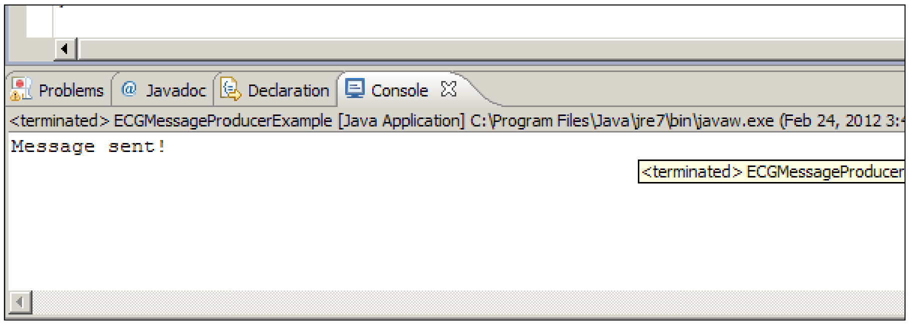
Once we have instantiated the queue and the session, we create a `MessageProducer` object, and using a `TextMessage` object, we send it to the chosen queue. The pushing of the object on the "queue.DLQ" is done using the `send` method on the `MessageProducer` object. We recall for the moment that, following the JMS specifications, the message producer object is equipped with the following four main properties:

- **Destination**: This is a JMS-administrated object
- **deliveryMode**: This specifies how the message should be delivered
- **Priority**: This specifies what the priority of the message is
- **timeToLive**: This specifies how many milliseconds the message should be alive for, in the queue

At the moment, we won't specify any of the parameters; we will manage that when dealing with the expiration of messages in *Chapter 5*, *Controlling Message Flow*, but such parameters greatly impact the performance, so they need to be carefully configured in a high-performance environment.

As good programming practice, close the `messageProducer` object once it has done its job.

We are now ready to have a first run of our code. If everything is ok, you should see on the **Console** tab of the Eclipse IDE what is shown in the following screenshot:



```
<terminated> ECGMessageProducerExample [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (Feb 24, 2012 3:
Message sent!
<terminated> ECGMessageProducer
```

So your first program, publishing a JMS Message on HornetQ, has been successfully coded! But there are two more steps to be implemented to complete the example. First, we need to read the message just stored on the queue, retrieve it, and save it to the MongoDB database. In the end, we will close all the resources involved in the correct way.

Consuming the message

Let us now start by introducing the code for the `consumeMessage` method. Inside this, we will call a private method called `storeIntoMongo` to perform the commit operation to MongoDB. How MongoDB manages and stores the document using the NoSQL paradigm is an extensive task, which does not affect this book. But we would like to demonstrate how to store information to a final destination. The reader should think that the consumer of the messages of a HornetQ server should do some other tasks apart from reading the messages. The obvious one would be to store the message for a second-stage analysis, such as storing log messages for parsing and using historical data.

For the interested reader, I would like to suggest to you, only as an exercise, to think of how to code the message storing using your preferred RDBMS, or even modify the method we implemented to store the messages on a text file.

Before doing this exercise, you should take a look at the following `consumeMessage` code:

```
javax.jms.MessageConsumer messageConsumer = session.
createConsumer(queue);
javax.jms.TextMessage messageReceived = (TextMessage)messageConsumer.
receive(5000);
insertMongo(messageReceived);
System.out.println("Received message: " + messageReceived.getText());
messageConsumer.close();
```

As you can see, the code is somewhat similar to the produceMessage method. On the same session object and the same queue that we used to produce the message, we consume it. This time, we create a MessageConsumer object. The messageConsumer object can receive messages in a synchronous or asynchronous way, the main difference being that the synchronous way can be obtained by simply calling the receive method; while in the asynchronous way, when a new message is received, an event is triggered. So with asynchronous message consumption, we need to register with the messageConsumer object a new MessageListener event whose onMessage() method will be called. In this case, we have implemented a synchronous receive() method using a long number that gives the time in milliseconds. In this case, messageConsumer receives the next message in a time frame of 5 seconds. Once we have the TextMessage message, we pass it to the method that will insert it into the MongoDB database, print the message to the console, and close the messageConsumer object.

As we said before, the messageConsumer object can receive the messages in a synchronous/asynchronous way. In both the cases, however, when you call the close method, the action is blocked until a receive method or message listener is in progress.

Connecting to MongoDB

As the last part of our example program, we will show you how to get the messageConsumer text message and store it in MongoDB. Again, we will not focus on the details too much, as managing push storage into a NoSQL database is a little bit different from a common insert statement in a classical RDBMS and requires different concepts. We won't be looking at this in much depth, but I would like to show you how things work.

The code of the insertMongo method is as follows:

```
try {
          m = new com.mongodb.Mongo();
          db = m.getDB( "hornetqdb" );
    } catch (UnknownHostException | MongoException e) {
          e.printStackTrace();
    }

    com.mongodb.DBCollection coll = db.getCollection("testCollection")
;
    com.mongodb.BasicDBObject doc = new com.mongodb.BasicDBObject();

   doc.put("name", "MongoDB");
```

```
      doc.put("type", "database");

      com.mongodb.BasicDBObject info = new com.mongodb.BasicDBObject();

      info.put("textmessage", messageReceived.getText());
```

The first `try/catch` block creates a new instance of MongoDB and connects to the `hornetqdb` database. If the database does not exist the first time you launch the program, MongoDB will create it for you. Remember that if, at this time of your test, you have closed the command prompt that launched the MongoDB database, simply go into the main MongoDB `root` folder inside the `bin` folder, and relaunch it.

Storing to MongoDB

The next step is to create a document collection that somehow is like a schema on a RDBMS database, declare it as a database, and put it inside a document named `textmessage`, whose value is the string we received from the message consumer, using the `getText` method of the `messageReceived` object. If you have your MongoDB window open, you should see a text message similar to the following:

Mon Feb 20 14:51:24 [initandlisten] connection accepted from 127.0.0.1:49867 #2

Mon Feb 20 14:51:24 [conn2] end connection 127.0.0.1:49867

This tells you that you were able to successfully store your message in MongoDB.

Closing everything

This first example is nearly over. Consider that the message producer and message consumer layer insist on the same connection, and on the same session. So, it is important to close them to free up the resources. The garbage collector is not in charge of closing unreferenced objects. Here is the closing connection method that follows the best practices in closing the connection and session associated to a JMS queue:

```
  if (session != null ) {
            try {
                  session.close();
            } catch (JMSException e) {
                  e.printStackTrace();
            }
      }

      if (connection != null) {
            try {
                  connection.close();
            } catch (JMSException e) {
   e.printStackTrace();
            }
      }
```

The first step tries to close the session object, and if it succeeds, closes the associated connection. In this case, the connection has been closed in the simplest way, but we have to observe that if we deal with a transactional message producer, then a better way to deal with the closing would be to commit all the transactions that are still pending, or your code will be exposed to the problem of losing messages from the production layer. Similarly, if you use an asynchronous message consumer, you should make sure to close all the connections and sessions while you don't have messages in transaction.

We will cover more advanced features such as this in *Chapter 5*, *Some More Advance Features of HornetQ*. Let's recap what we have covered:

- First of all, we set up a fully working environment to code a simple JMS example using HornetQ
- For the moment, we chose a standalone, non-clustered server for our developing purpose, and we have seen how to launch it
- As an optional task, we will also install and configure a single instance of the MongoDB NoSQL database
- Using Eclipse, we coded a class for connecting to the default queue launched by HornetQ, and for creating a session on it
- Within the session, we coded a simple JMS `messageProducer` object to send a composite string to the default queue
- From the same session object, we instantiated a synchronous `messageConsumer` object that reads the message stored and passes it for storing to the MongoDB instance we created

Summary

Before introducing the next chapter, I would invite you to code two classes starting from our example, one for the producer layer and one for the consumer layer, to see the issues arising from not having the connection shared in the same class. In such a more realistic example, you could also see, by launching the two classes separately, how message consumer and message producer layers work asynchronously with one queue. For the `messageConsumer` object, we also invite you to code a CSV writer, to allow the various messages to be stored in a file-based format.

One of the interesting features of HornetQ is its high interoperability with JBoss Application Server, and its ability to be run in several different OSs or embedded in code. It is also very configurable according to the user's needs. We will now move from our first example to a more configuration-focused chapter, where we will learn how to configure a standalone HornetQ server, and how to configure HornetQ so that it can be used by JBoss as its default JMS server.

We will also learn how to scale the machine to match the system requirements of HornetQ, and even how to run it as a service on both Windows and Linux.

2
Setting Up HornetQ

In the previous chapter, we set up HornetQ as a standalone server running on a Windows machine, in a very easy way, and we gave some advice on how to set up the same server on a Linux machine. This chapter will discuss in detail how to set up a general HornetQ server and the way HornetQ can be installed, either coupled with JBoss or as a standalone server. But before going through the setup procedure, we need to understand which version is right for a given production environment. At the end of this chapter, you should be able to perform the following tasks both on Windows and Linux:

- Installing and configuring a standalone HornetQ server
- Installing and configuring JBoss AS 7 with HornetQ
- Compiling the HornetQ standalone server from various sources
- Running HornetQ as a service

Which HornetQ version do I need?

HornetQ is highly configurable according to the destination purpose, but essentially, it can be used in the following ways:

- **Standalone**: As we have a single instance of the server running on a machine, it is independent from other software/frameworks
- **Integrated**: Starting from JBoss Application Server Version 6, HornetQ is the default Java messaging system for JBoss, so you can use and administrate HornetQ from JBoss
- **Embedded**: As HornetQ is distributed without any dependency, it can be launched and managed from a Java class

It is also possible to use multiple HornetQ instances grouped together in a cluster configuration, but we will cover that in *Chapter 6*, *Advanced Programming Features of HornetQ*.

If you are planning to use HornetQ coupled with JBoss, remember that the single machine where everything should run could need some heavy hardware resources. Our suggestion is to keep JBoss separate from HornetQ in a multi-instance configuration so that it will be easier to expand it in the future, without any need to touch the JBoss configuration, or worse, having to restart JBoss. In general, a good suggestion is to use the standalone version that can be scaled to a cluster environment in an easier way.

What about the cloud?

Cloud computing is becoming another option for the deployment of any framework. At the moment, HornetQ is not cloud-ready, and all the configuration steps for putting a HornetQ standalone server on the cloud is outside the scope of this book. No **PaaS** (**Platform as a Service**) provider gives direct support to HornetQ, but many cloud service providers, such as Amazon or Azure, can deploy a single virtual machine on their cloud infrastructures. If you succeed in configuring a virtual machine with HornetQ, you could move it into a cloud infrastructure.

A word on the operating system

HornetQ is nearly operating system agnostic, but as usual, there are some OS-dependent setup procedures that should be considered. HornetQ comes with its own high-performance journaling, and ready-for-persistence purpose libraries. In a Linux environment, starting from kernel 2.6, HornetQ can also benefit from using Linux's **Asynchronous IO library** (**AIO**). For our test, we will use two virtual machines created using VirtualBox; one is a 32-bit Windows 2008 Server and the other is an Ubuntu 11.10. So I will explain the differences between the two OSs, but in some cases I will only give the procedure for one OS, referring the reader to where they can check the procedure for the other OS.

System requirements

There are no particular requirements for a HornetQ instance to be installed on a machine, but two things should be noted:

- Even if the HornetQ documentation claims that "HornetQ runs on any platform with a Java Version 5 or later runtime environment", if you are planning to use JBoss AS 5 coupled with HornetQ, you need to link different `.jar` files to your client code.
- HornetQ utilizes, by default, 1GB of RAM when started. Even if you can change this parameter, a lower memory level can affect performance.

Before going through the procedure, do not forget to install the latest **Java Runtime Environment** (**JRE**) on your machine if you are planning to create a production-ready machine environment for the HornetQ standalone server. If you are also planning to run JBoss on the same machine, do not forget to download and install not only the JRE but also the **Java Development Kit** (**JDK**).

Installing the Java Virtual Machine

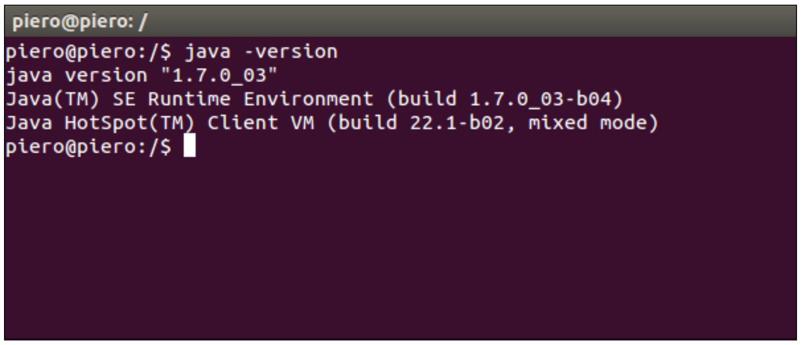
At the time of writing this book, the latest JRE/JDK for Windows was the Version 1.7 Update 3 (`http://www.oracle.com/technetwork/java/javase/downloads/index.html`).

In a Windows environment, you only need to launch the `.exe` file and wait till the installation procedure finishes. For an Ubuntu environment like ours, considering that since Version 1.7 the JRE/JDK is no longer available on Ubuntu repositories, you need to add an extra repository containing a script that will allow you to install/update your JRE/JDK. To do this from a no-root account, you need to follow this procedure (the same command without `sudo` can be done from a root account):

```
sudo add-apt-repository ppa:webupd8team/java
sudo apt-get update
sudo apt-get install oracle-jd7-installer
```

In this case, we use a third-party repository; this is not officially supported, so if you prefer a more standard way of installation, we point you to the official Oracle documentation.

After finishing, you should be able to run the `java -version` command and get the output shown in the following screenshot:

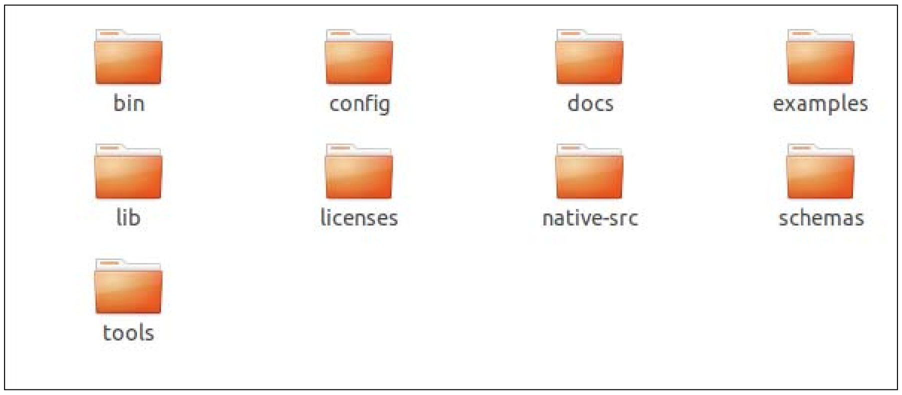


```
piero@piero:/$ java -version
java version "1.7.0_03"
Java(TM) SE Runtime Environment (build 1.7.0_03-b04)
Java HotSpot(TM) Client VM (build 22.1-b02, mixed mode)
piero@piero:/$
```

Installing the HornetQ standalone server

As we have seen in *Chapter 1, Getting Started with HornetQ*, if you want to install HornetQ as a standalone server, you need to download the latest binary version from `http://www.jboss.org/hornetq/downloads`. At the time of writing, the latest stable release was the HornetQ Version 2.2.5.Final. On the same page, you can also download the source files to compile your binaries from scratch. For Windows you can download the file named `hornetq-2.2.5.Final.zip`, and for Linux you can download the file `hornetq-2.2.5.Final.tar.gz`. In both cases, you only need to decompress the ZIP archive and put it where you want on your machine. We should also rename the unzipped folder to `hornetq`, so that for our Windows environment the `HORNETQ_ROOT` folder is `C:\hornetq`, and for our Ubuntu machine the `HORNETQ_ROOT` folder is `/hornetq`.

For your `HORNETQ_ROOT` folder, the folder tree structure is similar to the following screenshot:



Let's consider every folder that we have:

- **bin**: This folder contains the script to start/stop managing the server
- **config**: This folder contains the XML script to configure the instance of the server
- **docs**: This folder contains the documentation and Java doc
- **examples**: This folder contains various examples of using HornetQ API
- **lib**: This folder contains the `.jar` files containing the HornetQ compiled binaries
- **licenses**: This folder contains the `.txt` files containing the HornetQ licenses

- **native-src**: This folder contains the `src` for allowing HornetQ to use the HornetQLibAIO library, so that the kernel AIO calls can be used
- **schemas**: This folder contains the XSDs defining the schemas for the XML config files
- **tools**: This folder contains an `ant` folder that is used for building and running HornetQ from various sources

In a production environment, it is important to remove the unnecessary folders. You can remove, from the `HORNETQ_ROOT` folder, the subfolders—`docs`, `examples`, `licenses`, `tools`—and in a Windows environment you can also remove `native-src`.

Starting/stopping HornetQ

As we saw in *Chapter 1*, *Getting Started with HornetQ*, to run the HornetQ standalone server in Windows, you only have to double-click on the `run.bat` file available in the `HORNETQ_ROOT\bin` folder. For an Ubuntu environment, you have the counterpart, which is the `run.sh` file. Even in this case, starting the HornetQ standalone server is pretty easy. Open a terminal, go to your `HORNETQ_ROOT/bin` folder, and type the command `./run.sh`. In this case, you should see that the server has started correctly.

As I have mentioned before, it is good security practice on Linux (and also on Windows), not to run HornetQ from a root account. In this case, an interesting choice is to create a dedicated system user account named `hornetq`, to start/stop the server. If you choose to do this, you also need to check that the entire `HornetQ_ROOT` folder is accessible with read/write/execute permissions to the HornetQ user; otherwise even if you are able to launch the server, you could get some execution errors.

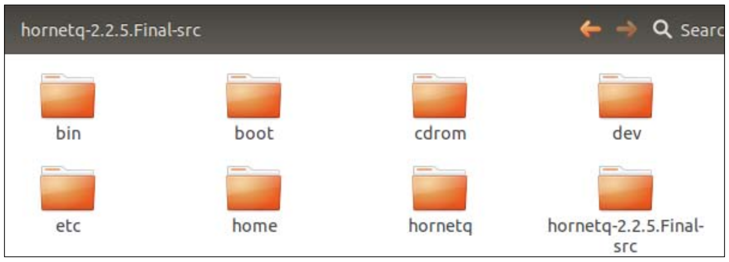
To stop your HornetQ standalone server in a controlled way, you only need to run the `stop.bat` script in Windows, or the `stop.sh` script if you are working with Linux.

Even if, from the terminal/prompt, you decide to force the process to a close by giving an interrupt such as *Ctrl* + *C*, I recommend you to use the script, because otherwise you are interrupting the process; this can cause a lock file to be written. If this happens in the next startup, HornetQ will check that condition, and you will not be in a clean condition.

Compiling from sources

HornetQ sources are distributed with the Apache License Version 2.0, so you can compile them by yourself. If you are keen to contribute to the project as a coder, you could use the `svn` repository, but can also find all the sources directly packed in a ZIP file on the HornetQ main **Downloads** page. If you do not need to change anything in the code, use the packed sources in a production environment. Otherwise, you need to have subversion installed, and you do not have any assurance that the standard procedure will work without compilation errors.

To compile the sources from both Windows and Linux, download the latest sources from `www.jboss.org/hornetq/downloads`. Once you unzip the file in any place, you will have a folder named `hornetq-<version number>.Final-src`, where `<version number>` is the version of the sources. As we did before, from now on we will call this folder the `HORNETQSRC_ROOT` folder. In our Ubuntu, for example, as shown in the following screenshot, the `HORNETQSRC_ROOT` is **hornetq-2.2.5.Final-src**:

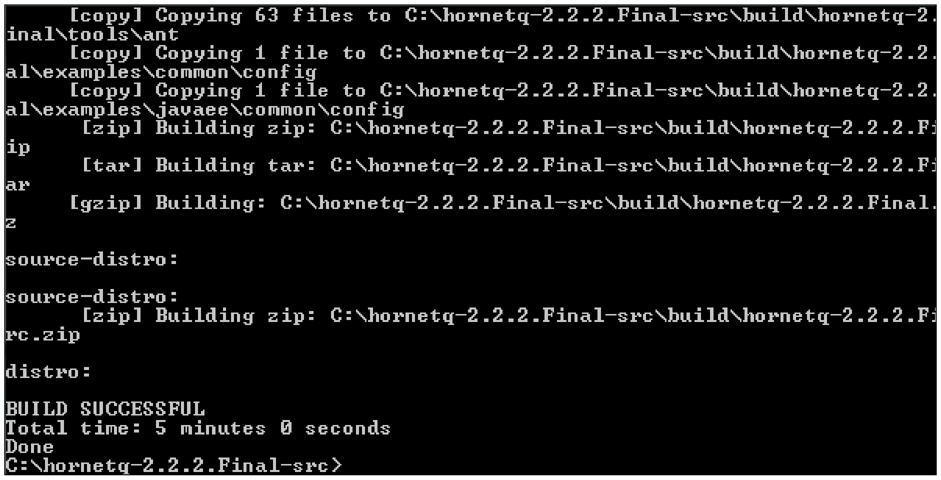


Now enter the **bin** directory in the `HORNETQSRC_ROOT`; you will find a file named `build.bat` on Windows and `build.sh` for Linux. To compile the sources, you need to launch this file with a parameter that tells the ant/maven tools (used to compile) what to create.

You have different possibilities when referring to Linux:

- `./build.sh distro`: This will create a `.zip` and `tar.gz` file containing the full distribution
- `./build.sh jar`: This will create the `.jar` files needed for the purpose of development
- `./build.sh dev-test`: This will run the entire test suite

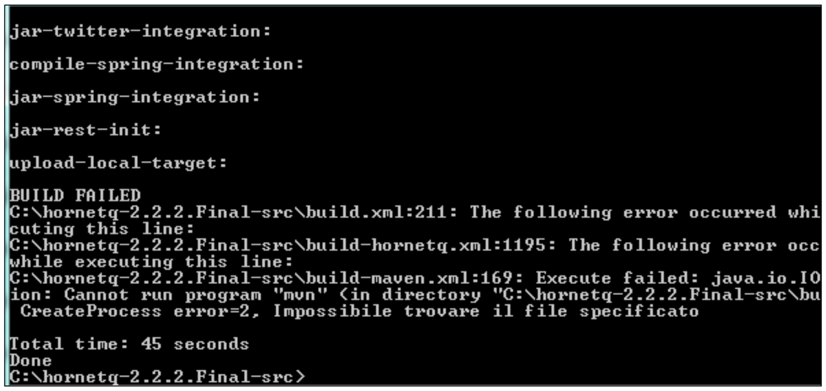
There are also other possible parameters that can be used, and you could refer to the HornetQ documentation or Wikipedia. In our case, you have to run the `./build.sh distro` command in a command prompt on a Linux environment, or the `build.bat distro` command in a Windows command prompt. If everything works fine, you will have the following output:



```
     [copy] Copying 63 files to C:\hornetq-2.2.2.Final-src\build\hornetq-2.
inal\tools\ant
     [copy] Copying 1 file to C:\hornetq-2.2.2.Final-src\build\hornetq-2.2.
al\examples\common\config
     [copy] Copying 1 file to C:\hornetq-2.2.2.Final-src\build\hornetq-2.2.
al\examples\javaee\common\config
      [zip] Building zip: C:\hornetq-2.2.2.Final-src\build\hornetq-2.2.2.Fi
ip
      [tar] Building tar: C:\hornetq-2.2.2.Final-src\build\hornetq-2.2.2.Fi
ar
     [gzip] Building: C:\hornetq-2.2.2.Final-src\build\hornetq-2.2.2.Final.
z

source-distro:

source-distro:
      [zip] Building zip: C:\hornetq-2.2.2.Final-src\build\hornetq-2.2.2.Fi
rc.zip

distro:

BUILD SUCCESSFUL
Total time: 5 minutes 0 seconds
Done
C:\hornetq-2.2.2.Final-src>
```

Once finished, you will find in the `HORNETQSRC_ROOT\build\` folder, a file named `hornetq-<version>.Final.tar.gz` for Linux and `hornetq-<version>.Final.zip` for Windows, which is the same archive we downloaded from the HornetQ website.

If you compile the same way in Windows, you will have an error due to a bug on the `build-maven.xml` file. The error is shown in the following screenshot:



```
jar-twitter-integration:

compile-spring-integration:

jar-spring-integration:

jar-rest-init:

upload-local-target:

BUILD FAILED
C:\hornetq-2.2.2.Final-src\build.xml:211: The following error occurred whi
cuting this line:
C:\hornetq-2.2.2.Final-src\build-hornetq.xml:1195: The following error occ
while executing this line:
C:\hornetq-2.2.2.Final-src\build-maven.xml:169: Execute failed: java.io.IO
ion: Cannot run program "mvn" (in directory "C:\hornetq-2.2.2.Final-src\bu
 CreateProcess error=2, Impossibile trovare il file specificato

Total time: 45 seconds
Done
C:\hornetq-2.2.2.Final-src>
```

To fix it, you need to enter the configuration file `build-maven.xml` in the `HORNETQSRC_ROOT` folder and substitute the lines containing the string `<exec executable="mvn">` with `<exec executable="mvn.bat">`.

Basic HornetQ configuration

Before discussing how to make HornetQ run as a service or the JBoss AS integration, we first need to understand how to configure the HornetQ standalone server. As we mentioned in *Chapter 1*, *Getting Started with HornetQ*, the HornetQ configuration files are stored in the `HornetQ_ROOT\config` folder. But if you take a look at the folder, you should see that it contains the following subfolders:

- `Jboss-as-5`
- `Jboss-as-4`
- `Stand-alone`

And inside every one of these folders are two more subfolders named:

- `Clustered`
- `Non-clustered`

The folder structure mimics the possible configurations of HornetQ according to the type of installation and according to the fact that HornetQ runs in a clustered mode, which means that the workload can be shared between different nodes. The whole of *Chapter 6*, *Advanced Programming Features of HornetQ* will be devoted to understanding such concepts, so for the moment we will focus on the folder under `HORNETQ_ROOT\config\standalone\non-clustered`, where we can find the following XML files:

- `hornetq-beans.xml`: This is the JBoss Microcontainer beans file, which defines what beans the Microcontainer should create and what dependencies to enforce between them. Remember that HornetQ is just a set of **Plain Old Java Objects** (**POJOs**). In the standalone server, it's the JBoss Microcontainer that instantiates these POJOs and enforces dependencies between them and other beans.
- `hornetq-configuration.xml`: This is the main HornetQ configuration file where all the basic configurations take place.
- `hornetq-jms.xml`: This is the configuration file used by the server-side JMS service to load JMS queues, topics, and connection factories.
- `hornetqusers.xml`: HornetQ ships with an implementation of a security manager, which obtains user credentials from the `hornetq-users.xml` file. This file contains username, password, and role information. We will cover such security topics in *Chapter 9*, *Ensuring Your Messages' Security*.

And two extra files:

- `jndi.properties`: This file is responsible for mapping the required JNDI packages.
- `logging.properties`: The Java logging framework is the default logging system for HornetQ. This is the main file that is needed from this framework, to do this job. We will cover the configuration issues in *Chapter 11*, *More on HornetQ Programming*.

If you take a look at the `run.bat` script in Windows, you should see the following lines:

```
set HORNETQ_HOME=..
IF "a%1"== "a" (
   set CONFIG_DIR=%HORNETQ_HOME%\config\stand-alone\non-clustered
) ELSE (
   SET CONFIG_DIR=%1
)
```

So when you launch the `run.bat` or `run.sh` script without any parameter, the default configuration file directory is the `HORNETQ_ROOT\config\stand-alone\non-clustered`. We are now going to take a look at the basic configurations that are needed to run a non-clustered, standalone HornetQ server.

First, we look at the `hornetq-beans.xml` file where you will find a section like the following:

```
<bean name="JNDIServer" class="org.jnp.server.Main">
   <property name="namingInfo">
      <inject bean="Naming"/>
   </property>
   <property name="port">${jnp.port:1099}</property>
   <property name="bindAddress">${jnp.host:localhost}</property>
   <property name="rmiPort">${jnp.rmiPort:1098}</property>
   <property name="rmiBindAddress">${jnp.host:localhost}</property>
</bean>
```

From a coder's point of view, this is the section where we define the JNDI server that will listen on port 1099, to allow the discovery of the queues. There are some other sections that describe the security mechanism and how the configuration is done from file; we will cover them in *Chapter 11*, *More on HornetQ Programming*.

The main configuration file for HornetQ is the `hornetq-configuration.xml` file, where general configuration issues like the Netty configuration are written. For the moment, the interesting part of the XML file is the following one:

```
    <paging-directory>${data.dir:../data}/paging</paging-directory>

    <bindings-directory>${data.dir:../data}/bindings</bindings-
directory>

    <journal-directory>${data.dir:../data}/journal</journal-directory>

    <journal-min-files>10</journal-min-files>

    <large-messages-directory>${data.dir:../data}/large-messages</
large-messages-directory>
```

This section contains the mapping where the HornetQ paging files are stored and how they are stored. As you can see, the bold values are the directories where the datafiles are stored. By default they refer to the `HORNETQ_ROOT\data` folder, but remember that you can put an absolute path to store the data file in a folder of your choice.

If you plan to use JMS, the queue name is mapped in the following XML tag that is present in the `hornet-jms.xml` file, for example the following one:

```
<queue name="ECGQueue">
   <entry name="/queue/ECGQueue"/>
</queue>
```

As we saw in *Chapter 1*, *Getting Started with HornetQ*, this is the name that you need to use for the purpose of discovering inside the consumer/producer layer. If you need to define your extra queue, you could insert a new tag. It is also possible to define the same tag in another file called `hornetq-jms.xml` that you can put into the `HORNETQ_ROOT\config` folder, to separate the queue names from other configurations.

Starting HornetQ as a service in Windows/Linux

As we have seen, starting HornetQ is nothing more than launching a script file, but in the production machine, we need to avoid manually restarting the HornetQ server. In this section, we will detail how it is possible to start from the script and run HornetQ as a boot service. Let us start with Ubuntu as that is the simpler case.

HornetQ as a service in Linux

Configuring the HornetQ standalone server to run in a debian-based Linux distribution at boot time involves simply creating a script that needs to be put in the `/etc/init.d` folder. For an easier configuration, follow these steps:

1. Using your preferred editor, create a file named `HornetQ` in the `/etc/init.d` folder.
2. Add the following lines to the text:

```
#!/bin/sh -e
#
# script for starting/stopping the HornetQ standalone server
#

case "$1" in
    start)
        cd /hornetq/bin  >&2
        ./run.sh -d >&2
        ;;
    stop)
        cd /hornetq/bin  >&2
        ./stop.sh -d >&2
        ;;
    *)
        echo "Usage: $0 {start|stop}" >&2
        exit 1
        ;;
esac
```

3. Save and type the `chmod +x hornetq` command in the command prompt.
4. In the command line, type `update-rc.d hornetq defaults`.

The commands we wrote are the standard ones that are needed for running a console script as a service in a debian-based install.

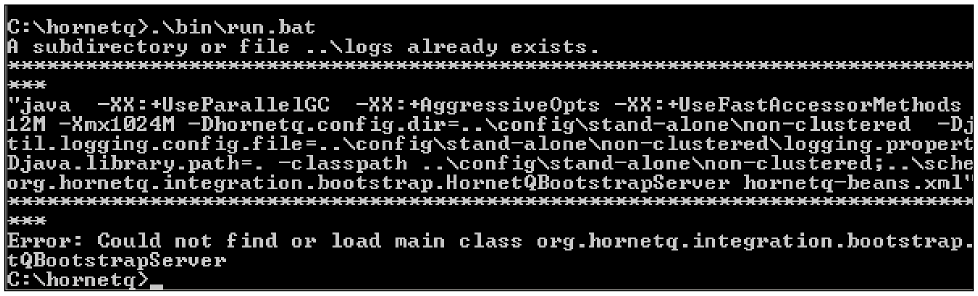
Now if you reboot the system you will have, in the next startup, your HornetQ server running.

In the case of a power failure, when the computer reboots we have a lock file, so we have a non-clear startup of the HornetQ instance. I suggest that the skilled or simply enthusiastic readers think of improving the script, by creating a script that will also manage the restart, even through a forced restart by removing the lock files in the `HORNETQ_ROOT\logs` folder.

Automating the HornetQ startup in Windows

Windows does not have a method to fully mute a `.bat` file into a service, being that the services are expected to be `.exe` files with particular management for the start and stop of the service itself. There are many third-party softwares that can use a `.bat` script file as a service, such as FireDaemon or RunAsService. This would be the clearest solution for doing such a task. However, if the reader does not want to use third-party software, we offer our "tips and tricks" solution for running the HornetQ standalone server as Windows boots.

A good practical solution is to launch the `run.bat` file in the `HORNETQ_ROOT\bin` folder when Windows starts up, using a **scheduled task**. However, creating a scheduled task that launches the `run.bat` file is not sufficient to guarantee the correct startup of the HornetQ standalone server. This is due to the fact that the `run.bat` file expects to be launched from the `HORNETQ_ROOT\bin` folder. The scheduled task has its own startup folder, so every path and Java classpath is configured for this folder. As a consequence, if you open a command prompt and launch the command from a different directory, you will get the error shown in the following screenshot:



```
C:\hornetq>.\bin\run.bat
A subdirectory or file ..\logs already exists.
*************************************************************************
***
"java  -XX:+UseParallelGC  -XX:+AggressiveOpts -XX:+UseFastAccessorMethods
12M -Xmx1024M -Dhornetq.config.dir=..\config\stand-alone\non-clustered  -Dj
til.logging.config.file=..\config\stand-alone\non-clustered\logging.propert
Djava.library.path=. -classpath ..\config\stand-alone\non-clustered;..\sche
org.hornetq.integration.bootstrap.HornetQBootstrapServer hornetq-beans.xml'
*************************************************************************
***
Error: Could not find or load main class org.hornetq.integration.bootstrap.
tQBootstrapServer
C:\hornetq>_
```

To solve this problem and use HornetQ as a Windows service, we are going to create a copy of the `run.bat` file, which can be launched anywhere. Next, we will use this file to create a scheduled Windows task that will run the modified file at Windows startup. The steps to follow are detailed as shown:

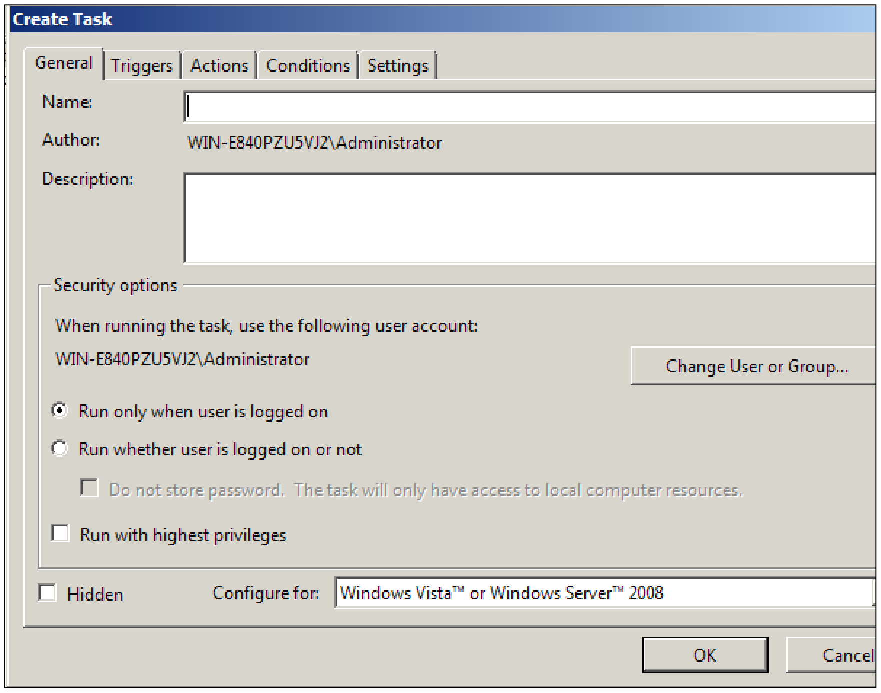
1. Create in the `HORNETQ_ROOT\bin` folder, a copy of the `run.bat` file named `runasservice.bat`.
2. Go to the line `set HORNETQ_HOME=..` and substitute the double dots with the full `HORNETQ_ROOT` folder without the final "\".
3. Go to the line `for /R ..\lib %%A in (*.jar) do (` and substitute it with `for /R %HORNETQ_HOME%\lib %%A in (*.jar) do (`.

Now if you launch `runasservice.bat` from outside the `HORNETQ_ROOT\bin` folder, you should see the server correctly started.

We need to point out that due to the fact that the `logging.properties` file refers to a relative path, when you launch `runasservice.bat` from outside the `HORNETQ_ROOT\bin`, the logfiles are not created in the `logs` folder of the `HORNETQ_ROOT` directory. The same thing happens for the `data` folder. To avoid this problem, you should set the full path to the `logs` and `data` folder into the corresponding files.

Apart from these issues, you are now ready to schedule a task with the `runasservice.bat` script. The operation is pretty simple:

1. Open **Administrative Tools** | **Task scheduler**.
2. From the right-hand side bar, click on **Create Task**; the following window will appear:



Name:

Author: WIN-E840PZU5VJ2\Administrator

Description:

Security options

When running the task, use the following user account:

WIN-E840PZU5VJ2\Administrator

[✓ Run only when user is logged on]

Run whether user is logged on or not

Do not store password. The task will only have access to local computer resources.

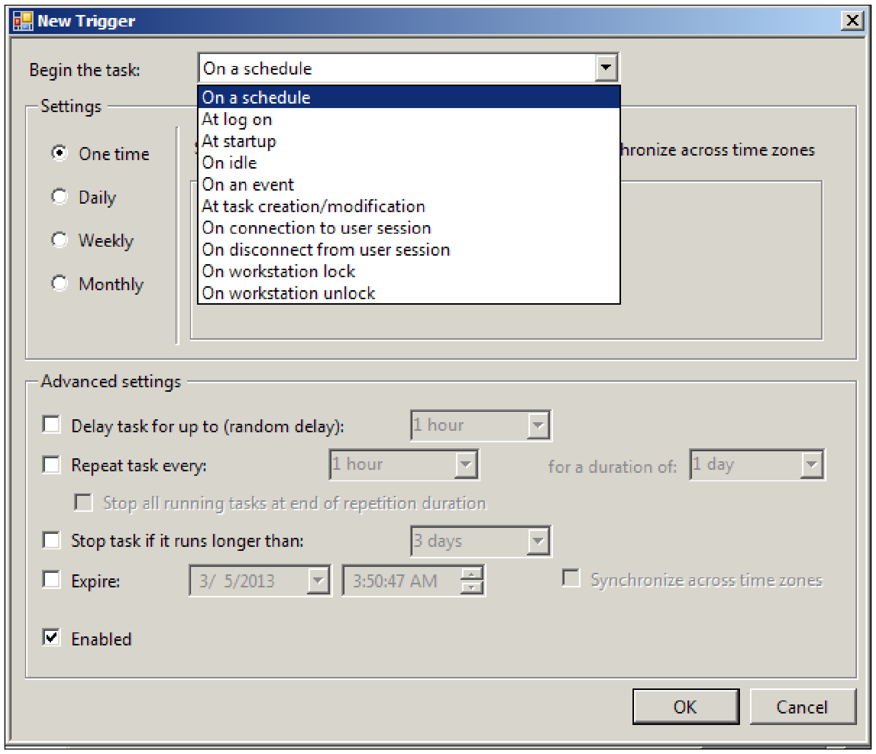
Run with highest privileges

Hidden

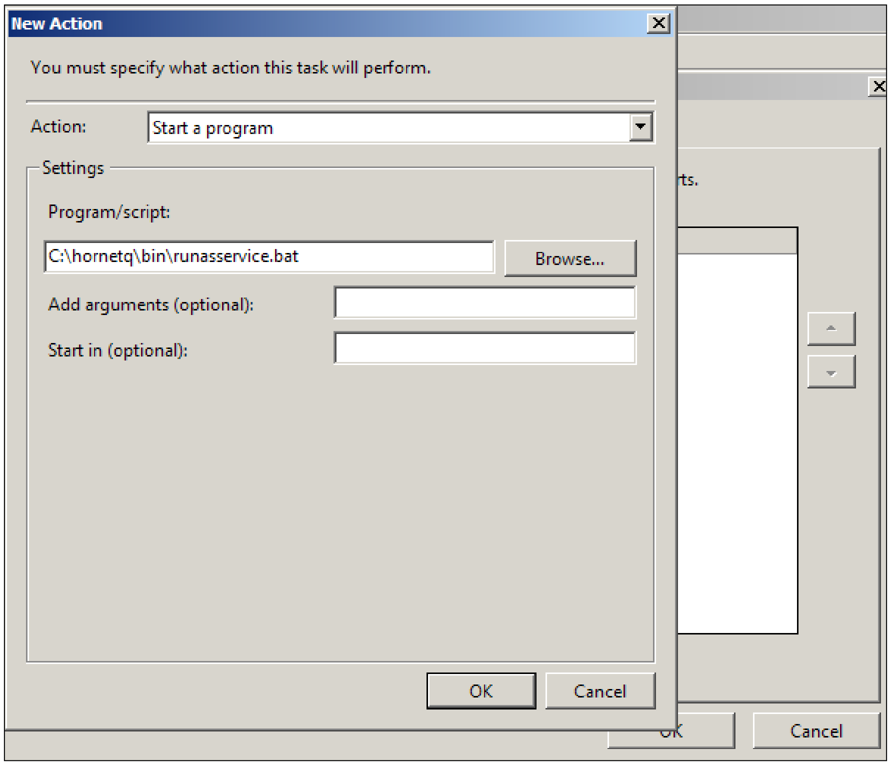
Configure for: Windows Vista™ or Windows Server™ 2008

3. Under the **General** tab, insert a name in the **Name** field.

4. Under the **Triggers** tab, click on **New trigger** and choose a startup trigger, as shown in the following screenshot:



5. Under the **Actions** tab, add a new **Action** as shown in the following screenshot:


New Action

You must specify what action this task will perform.

Action: Start a program

Settings

Program/script:

C:\hornetq\bin\runasservice.bat

Add arguments (optional):

Start in (optional):

6. After clicking on **OK** on the **Create Task** main window, you will have the HornetQ standalone server running at Windows startup.

Remember that this is only a trick solution and that shutting down Windows will cause a forced stopping of HornetQ, so at the next startup, you could have the problem of lock file to solve.

HornetQ and JBoss AS 7

Starting from JBoss Application Server 7, HornetQ has been integrated into the distribution as the JBoss default messaging provider. Armed with the HornetQ standalone application server concepts, we can move on to see how to configure HornetQ in JBoss AS 7. First you need to download the application server from `http://www.jboss.org/jbossas/downloads/`, and then you need to download the file release for 7.0.1 Thunder. If you downloaded a previous version, do not forget to download the full version, because the web version does not contain HornetQ as a module.

Once you download the ZIP file on Windows or Linux, you need to unzip it to have a `JBOSS_ROOT` folder. In our Linux environment, we have our `JBOSS_ROOT` folder under `/jboss-as-7.1.0.Final`.

JBoss uses XML files for configuring the optional module that should be launched with the application server when it is started. They are placed in the `JBOSS_ROOT\standalone\configuration\` folder. Unfortunately, running JBoss without parameters launches the `standalone.xml` configuration file, which does not use the messaging module.

So for our first test, we need to run the following command from the `JBOSS_ROOT\bin` folder:

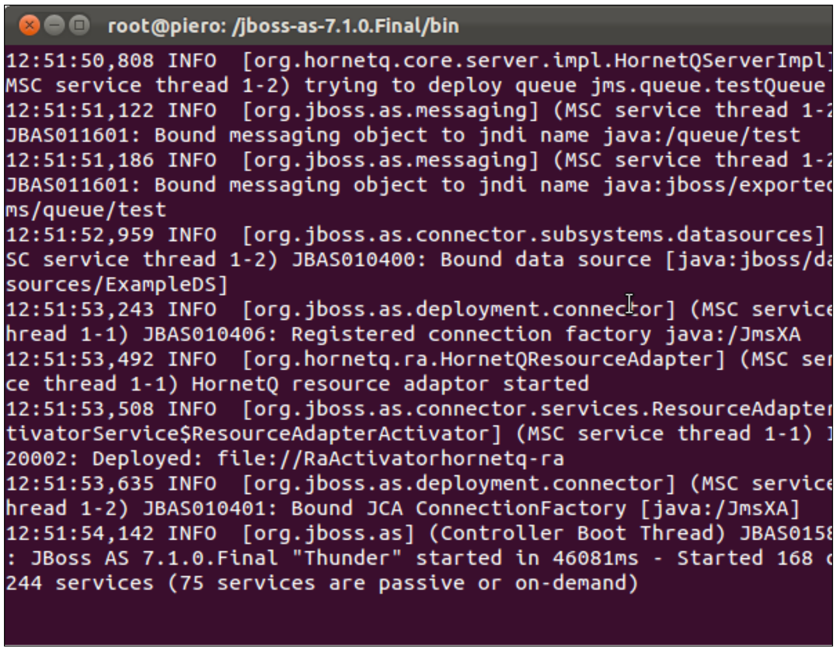
- In Linux:

  ```
  ./standalone.sh --server-config=standalone-preview.xml
  ```

- In Windows:

  ```
  standalone.bat --server-config=standalone-preview.xml
  ```

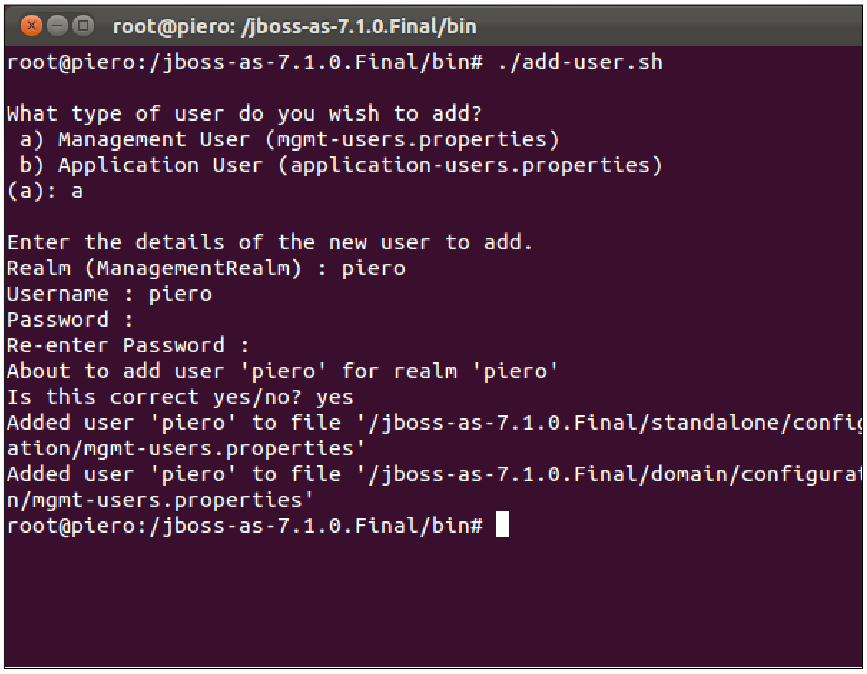
After some time, you should see the following screenshot:



```
12:51:50,808 INFO  [org.hornetq.core.server.impl.HornetQServerImpl
MSC service thread 1-2) trying to deploy queue jms.queue.testQueue
12:51:51,122 INFO  [org.jboss.as.messaging] (MSC service thread 1-2
JBAS011601: Bound messaging object to jndi name java:/queue/test
12:51:51,186 INFO  [org.jboss.as.messaging] (MSC service thread 1-2
JBAS011601: Bound messaging object to jndi name java:jboss/exported
ms/queue/test
12:51:52,959 INFO  [org.jboss.as.connector.subsystems.datasources]
SC service thread 1-2) JBAS010400: Bound data source [java:jboss/da
sources/ExampleDS]
12:51:53,243 INFO  [org.jboss.as.deployment.connector] (MSC service
hread 1-1) JBAS010406: Registered connection factory java:/JmsXA
12:51:53,492 INFO  [org.hornetq.ra.HornetQResourceAdapter] (MSC ser
ce thread 1-1) HornetQ resource adaptor started
12:51:53,508 INFO  [org.jboss.as.connector.services.ResourceAdapter
tivatorService$ResourceAdapterActivator] (MSC service thread 1-1) I
20002: Deployed: file://RaActivatorhornetq-ra
12:51:53,635 INFO  [org.jboss.as.deployment.connector] (MSC service
hread 1-2) JBAS010401: Bound JCA ConnectionFactory [java:/JmsXA]
12:51:54,142 INFO  [org.jboss.as] (Controller Boot Thread) JBAS0158
: JBoss AS 7.1.0.Final "Thunder" started in 46081ms - Started 168 o
244 services (75 services are passive or on-demand)
```

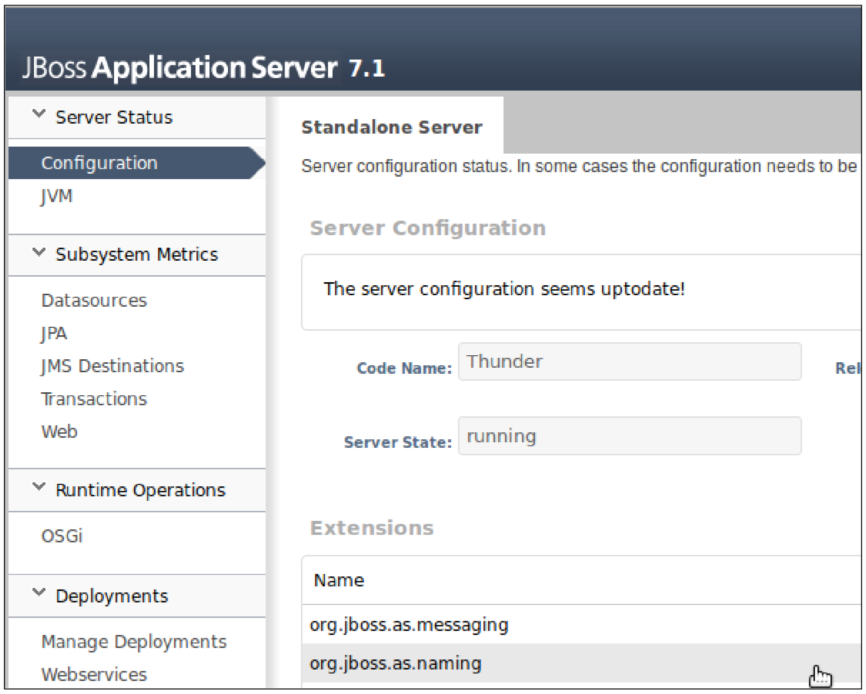
It states that your JBoss server has been successfully started, but what about HornetQ?

JBoss provides a management HTTP interface, but to use this, you need to add an admin user by using a script inside the `JBOSS_ROOT\bin` folder. To add the new admin user, type into the command prompt `./add-user.sh` for Linux, or in a Windows machine `add-user.bat`. The script will ask you some questions; we will not go into the details, however refer to the following screenshot, which is self explanatory:



```
root@piero:/jboss-as-7.1.0.Final/bin# ./add-user.sh

What type of user do you wish to add?
 a) Management User (mgmt-users.properties)
 b) Application User (application-users.properties)
(a): a

Enter the details of the new user to add.
Realm (ManagementRealm) : piero
Username : piero
Password :
Re-enter Password :
About to add user 'piero' for realm 'piero'
Is this correct yes/no? yes
Added user 'piero' to file '/jboss-as-7.1.0.Final/standalone/confi
ation/mgmt-users.properties'
Added user 'piero' to file '/jboss-as-7.1.0.Final/domain/configura
n/mgmt-users.properties'
root@piero:/jboss-as-7.1.0.Final/bin#
```

Now restart the JBoss server and point your browser to the URL `http://localhost:9990/console`; enter the credentials you provided for your user, and you should see the following interface:



Standalone Server

Server configuration status. In some cases the configuration needs to be

Server Configuration

The server configuration seems uptodate!

Code Name: Thunder

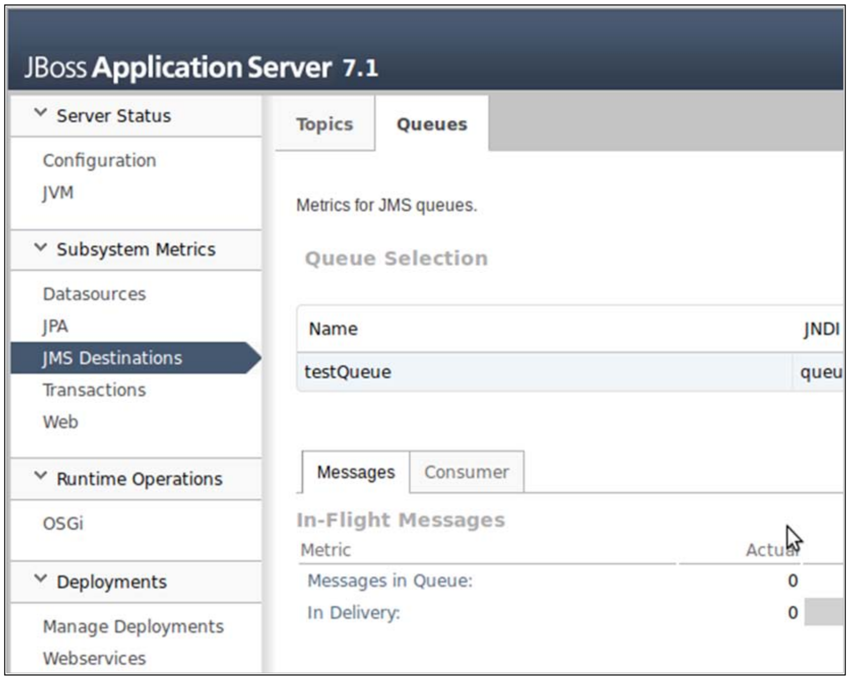
Rel

Server State: running

Extensions

Name
org.jboss.as.messaging
org.jboss.as.naming

As you can see, one of the extensions loaded is **org.jboss.as.messaging**, as you can use messaging systems other than HornetQ. If you click on the **JMS Destinations**, you will see the **JMS queues** as shown in the following screenshot:



So as you can see, you have successfully started HornetQ as the default message provider for JBoss, and there is a queue named **testQueue** waiting for storing the message. JBoss loads the modules it needs using some directive from its XML configuration file placed in the `HORNETQ_ROOT\standalone\configuration\` folder. If you open the file that we used (`standalone-full.xml`), you should see the following entries telling JBoss to load HornetQ as the messaging system. If the tag is commented, you should remove the comments and restart.

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

In the same file, you also have the section that starts with the following tag:

```
<subsystem xmlns="urn:jboss:domain:messaging:1.1">
            <hornetq-server>
```

As you can see in JBoss, the HornetQ configuration server is no more than a module that should be configured in the JBoss configuration files. However, detailing all the possibilities offered by JBoss is outside the scope of this book.

For a more detailed explanation refer to *JBoss AS 7 Configuration, Deployment and Administration, Francesco Marchioni, Packt Publishing*. But you could use this configuration for correctly setting up JBoss with HornetQ.

Checking your setup/installation

If you use JBoss AS integrated with HornetQ, as you have seen, you have an administration interface that you can use for monitoring purposes. However, for every setup of HornetQ, it is always good practice to refer to the logfiles generated during the HornetQ server startup. As we mentioned, the default location for the standalone server is the `HORNETQ_ROOT\logs` folder, but keep in mind that the location can be overridden by the `logging.properties` file; so check the configuration first if you ever have any trouble. In case you are running HornetQ in JBoss AS, all the HornetQ logs are redirected using the default logging framework adopted by JBoss. As we point out in this chapter, if you start the JBoss server, you need to look, by default, in the `JBOSS_ROOT\standalone\log\` folder for the `server.log` file. Refer to the JBoss documentation because even JBoss has a lot of configuration capabilities. So, for example, if you use it in a cluster environment, you should be aware of where the log files are stored.

Apart from the boot procedure, another important task to check is whether the HornetQ JNDI port is replying correctly. In a public environment, the default port is normally closed to incoming connections by the firewall's configuration, so using the following command could help check that everything is working soundly:

```
telnet <hornetq_ip> <jndi port>
```

As a further exercise, change the default IP port, remembering that the port configuration is written in the `hornetq-beans.xml` file. Also, consider referring to the JBoss documentation to find which is the same port for the HornetQ and JBoss integrated server.

Summary

After finishing this chapter the reader should be able to correctly install HornetQ both on Windows and Linux systems. We also have seen where the config files, the head of your HornetQ instance, are located. We have also seen a basic JBoss installation. As for other Apache software, the folder where the configuration files are located can be changed to some other location. But for the first-time user we suggest to leave them in the default configuration folder.

Obviously, any installation procedure could lead to some adjustments due to the specificity of the OS. But the cases we present should be general enough, even if they do not cover some specific topic like HornetQ in a cluster or HornetQ distributed over a WAN (World Area Network).

Now that we have seen how to set up HornetQ, it is time to prepare yourself for a more deeply focused coding phase chapter. In *Chapter 3*, *Basic Coding with HornetQ: Creating and Consuming Messages*, we will see how to use the HornetQ core API to send and receive messages within a HornetQ standalone server.

3
Basic Coding with HornetQ: Creating and Consuming Messages

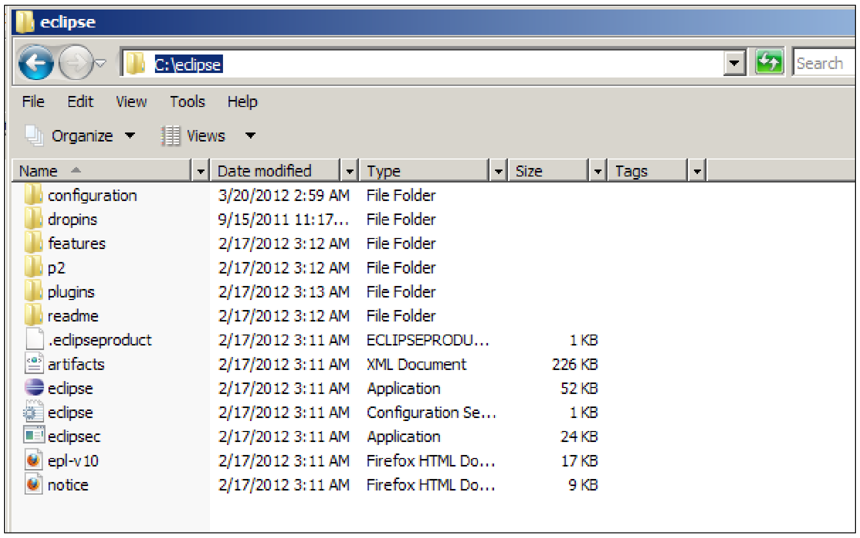
Starting from this chapter, you should now be ready to install and configure a HornetQ standalone server, and assign some queues that can be used for development and production purposes. We are now ready to move towards setting up a development environment, so that you can start coding with HornetQ. In this chapter, we will set up a full development environment using both Eclipse and NetBeans so that you will have a template project that you can use for developing with HornetQ. From now on we will assume that, on the machine on which you are developing there will be an instance of the HornetQ standalone server running. We will also assume that you have installed the latest **Java Development Kit** (**JDK**) in your OS.

In this chapter, we will go through a more detailed example of what we have done in *Chapter 1, Getting Started with HornetQ*. Using the Electrocardiography example string we will cover the following topics:

- Installing Eclipse and NetBeans for developing with HornetQ on both Windows and Ubuntu
- Setting up a development environment for working with HornetQ core API in Eclipse and NetBeans
- Creating an example for producing and consuming messages in HornetQ in both a synchronous and an asynchronous way
- Implementing some classes using the performance practice for managing core API connections, sessions, and clients

Installing Eclipse on Windows

As we saw in *Chapter 1, Getting Started with HornetQ*, you can download the Eclipse IDE for Java EE developers (in our case the ZIP file `eclipse-jee-indigo-SR1-win32.zip`) from `http://www.eclipse.org/downloads/`. Once downloaded, you have to unzip the `eclipse` folder inside the archive to the destination folder so that you have a folder structure like the one illustrated in the following screenshot:



Name	Date modified	Type	Size	Tags
configuration	3/20/2012 2:59 AM	File Folder		
dropins	9/15/2011 11:17...	File Folder		
features	2/17/2012 3:12 AM	File Folder		
p2	2/17/2012 3:12 AM	File Folder		
plugins	2/17/2012 3:13 AM	File Folder		
readme	2/17/2012 3:12 AM	File Folder		
.eclipseproduct	2/17/2012 3:11 AM	ECLIPSEPRODU...	1 KB	
artifacts	2/17/2012 3:11 AM	XML Document	226 KB	
eclipse	2/17/2012 3:11 AM	Application	52 KB	
eclipse	2/17/2012 3:11 AM	Configuration Se...	1 KB	
eclipsec	2/17/2012 3:11 AM	Application	24 KB	
epl-v10	2/17/2012 3:11 AM	Firefox HTML Do...	17 KB	
notice	2/17/2012 3:11 AM	Firefox HTML Do...	9 KB	

Now a double-click on the **eclipse.exe** file will fire the first run of Eclipse.

Installing NetBeans on Windows

NetBeans is one of the most frequently used IDE for Java development purposes. It mimics the Eclipse plugin module's installation, so you could download the J2EE version from the URL `http://netbeans.org/downloads/`. But remember that this version also comes with an integrated GlassFish application server and a Tomcat server. Even in this case you only need to download the `.exe` file (`java_ee_sdk-6u3-jdk7-windows.exe`, in our case) and launch the installer. Once finished, you should be able to run the IDE by clicking on the NetBeans icon in your Windows Start menu.

Installing NetBeans on Linux

If you are using a Debian-based version of Linux like Ubuntu, installing both NetBeans and Eclipse is nothing more than typing a command from the bash shell and waiting for the installation process to finish.

As we are using Ubuntu Version 11, we will type the following command from a non-root user account to install Eclipse:

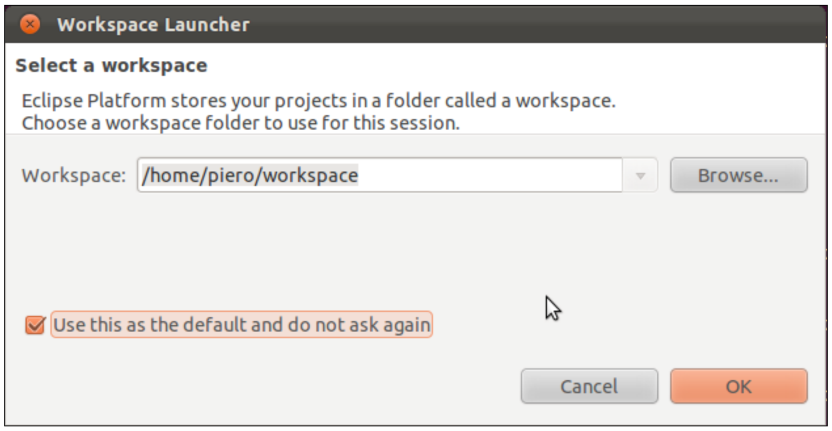
```
sudo apt-get install eclipse
```

The NetBeans installation procedure is slightly different due to the fact that the Ubuntu repositories do not have a package for a NetBeans installation.

So, for installing NetBeans you have to download a script and then run it. If you are using a non-root user account, you need to type the following commands on a terminal:

```
sudo wget http://download.netbeans.org/netbeans/7.1.1/final/bundles/
netbeans-7.1.1-ml-javaee-linux.sh
sudo chmod +x netbeans-7.1.1-ml-javaee-linux.sh
./netbeans-7.1.1-ml-javaee-linux.sh
```

During the first run of the IDE, Eclipse will ask which default workspace the new projects should be stored in. Choose the one suggested, and in case you are not planning to change it, check the **Use this as the default and do not ask again** checkbox for not re-proposing the question, as shown in the following screenshot:



Eclipse Platform stores your projects in a folder called a workspace.
Choose a workspace folder to use for this session.

Workspace: /home/piero/workspace

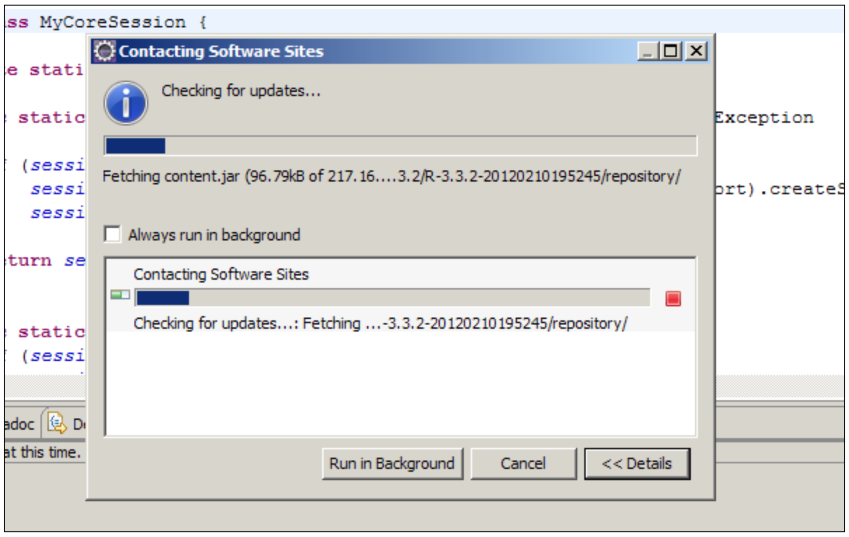
☑ Use this as the default and do not ask again

The same happens with NetBeans, but during the installation procedure.

Post installation

Both Eclipse and NetBeans have an integrated system for upgrading them to the latest version, so when you have correctly launched the first-time run, keep your IDE updated.

For Eclipse, you can access the **Update** window by using the menu **Help** | **Check for updates**. This will pop up the window, as shown in this screenshot:



Fetching content.jar (96.79kB of 217.16....3.2/R-3.3.2-20120210195245/repository/

Checking for updates...: Fetching ...-3.3.2-20120210195245/repository/

NetBeans has the same functionality, which can be launched from the menu.

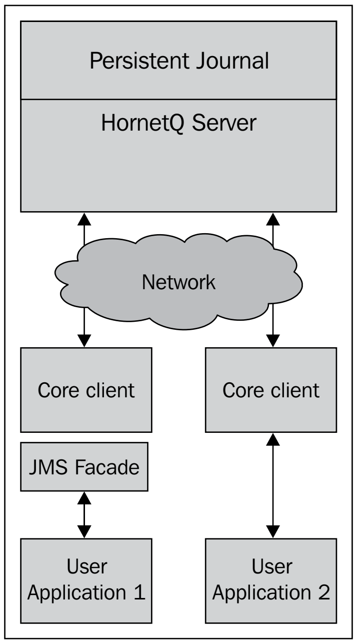
A 10,000 foot view of HornetQ

Before moving on with the coding phase, it is time to recover some concepts to allow the user and the coder to better understand how HornetQ manages messages.

In *Chapter 1*, *Getting Started with HornetQ*, we coded our first example of using JMS messages with HornetQ. The reader could be allowed to think of HornetQ only in terms of JMS messages. In fact, HornetQ is only a set of **Plain Old Java Objects** (**POJOs**) compiled and grouped into JAR files. The software developer could easily grasp that this characteristic leads to HornetQ having no dependency on third-party libraries. It is possible to use and even start HornetQ from any Java class; this is a great advantage over other frameworks.

HornetQ deals internally only with its own set of classes, called the **HornetQ core**, avoiding any dependency on JMS dialect and specifications. Nevertheless, the client that connects with the HornetQ server can speak the JMS language.

So the HornetQ server also uses a JMS to core HornetQ API translator. This means that when you send a JMS message to a HornetQ server, it is received as JMS and then translated into the core API dialect to be managed internally by HornetQ. The following figure illustrates this concept:



As we implemented the JMS facade from the client perspective in *Chapter 1*, *Getting Started with HornetQ*, in this chapter we are going to implement the core API specification to allow the client to connect to the queue and to produce/consume messages with the core classes. The core messaging concepts of HornetQ are somewhat simpler than those of JMS:

- **Message**: This is a unit of data that can be sent/delivered from a consumer to a producer. Messages have various possibilities that we will cover in the next chapter. But only to cite them, a message can have: durability, priority, expiry time, time, and dimension.
- **Address**: HornetQ maintains an association between an address (IP address of the server) and the queues available at that address. So the message is bound to the address.
- **Queue**: This is nothing more than a set of messages. Like messages, queues have attributes such as durability, temporary, and filtering expressions.

Thinking, then coding

HornetQ can handle messages in a very efficient way, but, as usual, lots of improvements can be managed by coding the message consumer and the message producer in a way that avoids any anti-pattern in performance and tuning. Remembering the HornetQ user manual, if you work on the code side of things, you need to avoid instantiating the producer, consumer, connections, or sessions for one single message. This is the most common performance anti-pattern, so you should try to avoid it.

We also need to alert the user that it's possible to improve performance by considering the following server-side check list:

- Tuning the journal
- Tuning JMS
- Tuning settings
- Tuning JVM
- Tuning your code

We will not cover each step, but in this chapter's example we will try to tune our code so that we will avoid having multiple connections, multiple sessions, and multiple consumer/producer objects at the same time.

The HornetQ core API example

We are now ready to move on from the example seen in *Chapter 1*, *Getting Started with HornetQ*, which used JMS to create and consume messages. We will re-code the same example using the core API. We will then code a message producer and a message consumer that will push/read ECG signals using core API messages to a HornetQ standalone non-clustered server. We will not cover the MongoDB interaction; that will be left to the willing reader as an exercise—it is only a simple refactor of the code we have seen in *Chapter 1*, *Getting Started with HornetQ*.

We will try to get only one connection object that is charged with connecting to the HornetQ standalone non-clustered server that is running on our test environment. Even the session that is shared between the consumer client and the producer client will be created with only one object.

Finally, we will detail how to code the message producer and the message consumer. If you use the core API objects, you will see that the message object has many methods to integrate objects into it. So we will take a look at some possible implementations and we will suggest some others to the user.

Preparing your development environment

This step is pretty easy to be arranged; if you use the HornetQ core API, you only need to add to your application classpath the following JAR files that can be found in the `HORNETQ_ROOT\lib` folder:

- `HornetQ-core.jar`
- `HornetQ-core-client.jar`
- `Netty.jar`

A HornetQ server can be created and used directly from Java code; from this it follows that the queues can also be created at runtime, but in this case we will detail the queue into the configuration file of HornetQ. Before going on we need to understand the difference between the JMS queue and the core API queue. In the JMS implementation there exists the idea of a message topic, which is absent in the core API. This means that in core API you have an address where the server is running on one or more queues bounded to this address. With core API it is possible to create a queue at runtime in your code or to use a queue mapped to your address. In our case, we will use a queue mapped to the localhost address.

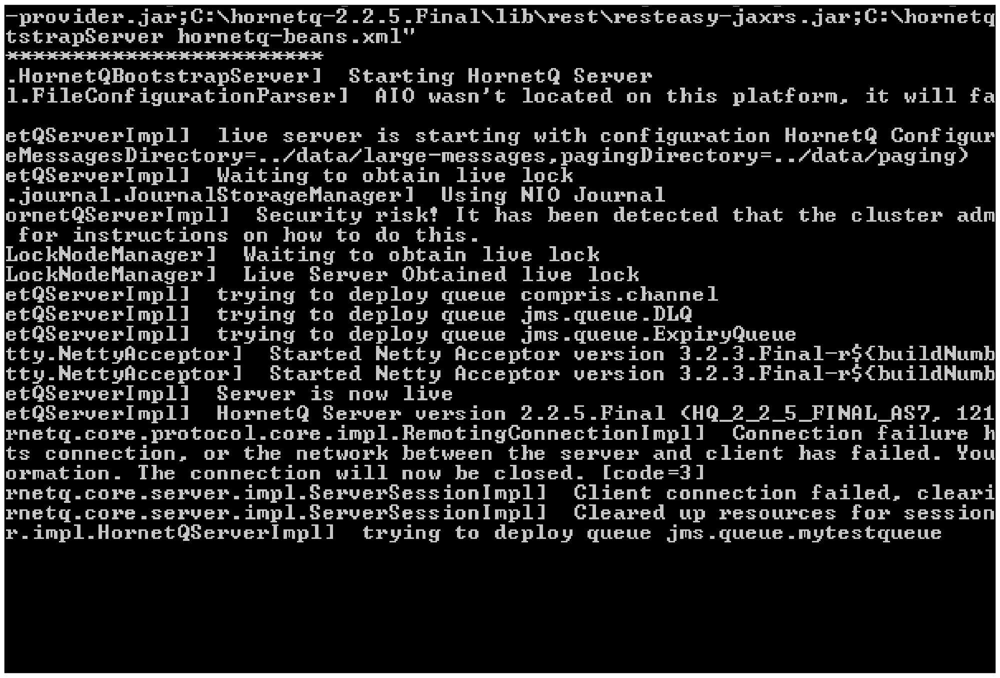
So, we will create a durable queue that will be used by both the producer client and the consumer client to deliver and use messages. The difference between durable and non-durable queue is pretty simple—non durable queues do not survive after a server restart, meaning that the messages that are not consumed are lost if the server crashes.

To configure such queues simply open, using your preferred text editor, the `hornetq-configuration.xml` file that you can find in the `HORNETQ_ROOT\config\standalone\non-clustered\` folder. Inside the `<configuration></configuration>` XML element, add the following element:

```
 <queues>
    <queue name="jms.queue.mytestqueue">
              <address>jms.queue.mytestqueue</address>
                     <durable>true</durable>
    </queue>
</queues>
```

This will create a predefined core API queue that is durable, meaning that all the messages bound to this queue will survive a server restart. By convention, all the predefined core queues have the `jms.queue` prefix before the name of the queue you want to assign.

To see for yourself how HornetQ works, at runtime you can change the `hornetq-configuration.xml` file and you will see that HornetQ will get the change and display something like the following screenshot:



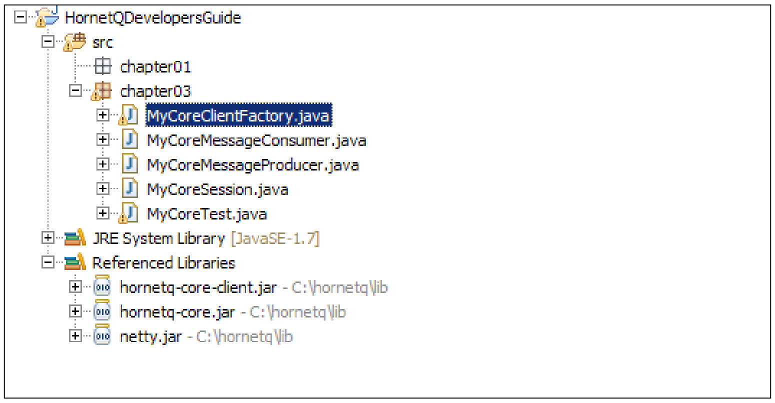
```
-provider.jar;C:\hornetq-2.2.5.Final\lib\rest\resteasy-jaxrs.jar;C:\hornetq
tstrapServer hornetq-beans.xml"
************************
.HornetQBootstrapServer]  Starting HornetQ Server
l.FileConfigurationParser]  AIO wasn't located on this platform, it will fa

etQServerImpl]  live server is starting with configuration HornetQ Configur
eMessagesDirectory=../data/large-messages,pagingDirectory=../data/paging)
etQServerImpl]  Waiting to obtain live lock
.journal.JournalStorageManager]  Using NIO Journal
ornetQServerImpl]  Security risk! It has been detected that the cluster adm
 for instructions on how to do this.
LockNodeManager]  Waiting to obtain live lock
LockNodeManager]  Live Server Obtained live lock
etQServerImpl]  trying to deploy queue compris.channel
etQServerImpl]  trying to deploy queue jms.queue.DLQ
etQServerImpl]  trying to deploy queue jms.queue.ExpiryQueue
tty.NettyAcceptor]  Started Netty Acceptor version 3.2.3.Final-r${buildNumb
tty.NettyAcceptor]  Started Netty Acceptor version 3.2.3.Final-r${buildNumb
etQServerImpl]  Server is now live
etQServerImpl]  HornetQ Server version 2.2.5.Final (HQ_2_2_5_FINAL_AS7, 121
rnetq.core.protocol.core.impl.RemotingConnectionImpl]  Connection failure h
ts connection, or the network between the server and client has failed. You
ormation. The connection will now be closed. [code=3]
rnetq.core.server.impl.ServerSessionImpl]  Client connection failed, cleari
rnetq.core.server.impl.ServerSessionImpl]  Cleared up resources for session
r.impl.HornetQServerImpl]  trying to deploy queue jms.queue.mytestqueue
```

Now we can move to our IDE to prepare the development environment.

Using the **New Java Project** wizard, create a new Java project called `Chapter03` with five classes in it:

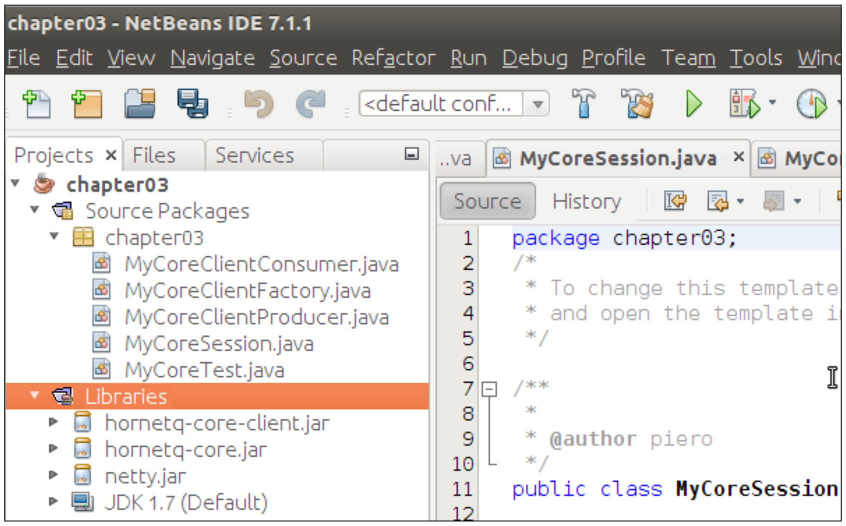
- `MyCoreClientFactory`: This will initialize the `CoreClientFactory` class
- `MyCoreSession`: This is in charge of returning a valid `CoreSession` class
- `MyCoreMessageProducer`: This is same for the `CoreMessageProducer` class
- `MyCoreMessageConsumer`: This is same for the `CoreMessageConsumer` class
- `MyCoreTest:` This is the class that will test the previous ones

So, in Eclipse, you will have the following configuration for the project:



HornetQDevelopersGuide
src
chapter01
chapter03
MyCoreClientFactory.java
MyCoreMessageConsumer.java
MyCoreMessageProducer.java
MyCoreSession.java
MyCoreTest.java
JRE System Library [JavaSE-1.7]
Referenced Libraries
hornetq-core-client.jar - C:\hornetq\lib
hornetq-core.jar - C:\hornetq\lib
netty.jar - C:\hornetq\lib

While in NetBeans (Ubuntu), you should have something like the following screenshot:



```
package chapter03;
/*
 * To change this template
 * and open the template i
 */

/**
 *
 * @author piero
 */
public class MyCoreSession
```

We are now ready to move on to the coding phase.

For the reader who would like to use a more object-oriented approach, we suggest, apart from the `MyCoreTest` class, the creation of an interface for the other classes in order to have every class as an implementation of a known interface.

Creating a shared core API connection

We are now ready to code the connection class that will be used by the session class to connect to a standalone HornetQ server using the core API. Basically, the `MyCoreConnectionFactory` class has the following static methods:

```
Public static ClientSessionFactory getConnectionFactory(string host,
int port)
Private static HashMap createSettings(String host,int port)
Public static void close();
```

The aim of this class is to manage one `ClientSessionFactory` HornetQ object, so that it will be possible for multiple session objects to use it. The `public main` method is declared as `static`, so we need the following static objects:

```
    private static ClientSessionFactory factory = null;
    static HashMap map = null;
    private static TransportConfiguration configuration;
    private static ServerLocator locator;
```

The code for the `main` method is:

```
public static ClientSessionFactory getConnectionFactory(string host,
int port) {
          if (factory == null){
                    configuration = new TransportConfiguration(NettyConne
ctorFactory.class.getName(), createSettings(host, port));
                    locator = HornetQClient.createServerLocatorWithoutHA(
configuration);
                    factory =  locator.createSessionFactory();
          }
          return factory;
}
```

Basically, the first time this method is called, the `factory` object, which is of type `ClientSessionFactory`, is null (because we choose to declare it as a static property of the class itself) so it will be instantiated and configured according to the IP address and the port where the core server is running using the `NettyConnector`. Then, the next time we need to access the connection, it will be re-used without any changes. We underline that we need to create a `ClientSessionFactory` object using a `ServerLocator` object. The `ServerLocator` object uses a `TransportConfiguration` object, which is an object used by a client to specify a connection to a server and its backup, if one exists.

Typically, its constructors take the class name and parameters needed to create the connection. These will be different and will depend on which connector is being used, `netty`, `InVM`, or something similar. The difference between the `IVM` (In Virtual Machine) connector and the `netty` connector is that the `netty` connector allows the client and the server to run on different virtual machines. Nevertheless, we strongly encourage using the `netty` connector, because it is the most configurable one.

The `netty` transport can be used in one of these ways; to use old (blocking) Java IO, NIO (non-blocking), to use straightforward TCP sockets like SSL, or to tunnel over HTTP or HTTPS. In addition to this we also provide a servlet transport.

This is why, even if we do not need the `netty.jar` library at compile time, we will need it at runtime, to allow the `netty` connector to work properly.

Having said that the `ClientSessionFactory` class can be closed, we implement a method named `close()`:

```
public static void close() {
      if (factory != null){
                factory.close();
                factory = null;
          }
}
```

This method will close the connection and set the `factory` object to null so that if some other resources need to access the `ClientSessionFactory` object, it will be reinitialized from scratch.

Creating and sharing a HornetQ session

Now that we have a connection factory at our disposal, it is time to move to a shared session that will be used by both the consumer and the producer of the message.

We follow the same methodology we used for the `ConnectionFactory` class. We will have only one method that will initialize a `ClientSession` object and return it after starting. The methods will be as follows:

```
public static void setSessionParameters(String host, int port)
public static ClientSession getSession()
public static void start()
public static void close()
```

So we have only one static `ClientSession` object that is mapped from the `ClientSessionFactory` class we have just coded. We have some methods to set up the connection and to create a consumer object on a queue defined by the user.

We first need to declare a static `org.hornetq.api.core.client.ClientSession` object that will be available and used every time we will call a method of the class in the following way:

```
private static ClientSession session = null;
```

The first method to be implemented is the `setSessionParameters(String host, int port)` method, which will be the one responsible for creating a connection within the `MyCoreClientFactory` object we just coded. The code is very simple:

```
            if (session == null) {
                    System.out.println("creating session at " + new java.
util.Date());
                    session = MyCoreClientFactory.
getClientSessionFactory(host, port).createSession(false, true, true);
            }
```

So we call the `getClientSessionFactory` method of the first `MyCoreClientFactory` class with the host and the IP address thus creating a `ClientSession` object. The three respective Boolean parameters are:

- `xa`: This parameter shows whether the session supports XA transaction semantics or not
- `autoCommitSends`: This parameter is set to true for automatically commiting message sends, and set to false for commiting manually
- `autoCommitAcks`: This parameter is set to true for automatically commiting message acknowledgement, and set to false for commiting manually

So, in this case our connection will support the transaction semantic that will automatically commit messages once the `send` method of the consumer is called, and will automatically acknowledge every message.

This is a simple case, but in a high-frequency message environment you will need to fine-tune these parameters. However, this will require more coding on the producer-consumer layer for effectively managing the commits and acknowledgements.

We also add a console output to show that the session will be created only once during our test even if we share it between the producer and the consumer.

The next simple method is the `getSession()` method—one that is in charge of returning to every call the `ClientSession` object just initialized. The following code is self-explanatory:

```
public static ClientSession getSession()
{
    return session;
}
```

We also need two more methods to start and stop the instantiated session. Here is the first one:

```
public static start()
{
if (session != null)
{
          session.start();

}
}

This is the second one:
public static close()
{
if (session != null)
{
          session.close();
          session = null;
}
}
```

To avoid a null object reference, the two methods control—in a very simple way—if the session object is first initialized. Armed with these two classes, we are now ready to move to the interesting part. We will now start to code an object that acts like a `MessageProducer` class.

Coding a client consumer

Now that we have the two static classes for managing the connection and the session, we are now ready to code a reusable object that can be used to send messages to a HornetQ queue.

The aim of this object, once created, is to send a message to the selected HornetQ queue by passing only three parameters—the IP address of the host where HornetQ is running, the port where the `netty` service is listening, and the name of the queue where we need to put the messages.

We need to instantiate two objects that will be delivering the messages:

```
private org.hornetq.api.core.client.ClientProducer producer;
private String queuename;
private org.hornetq.api.core.client.ClientMessage message;
```

The core `ClientMessage` object is equipped with lots of methods that can be used both for defining message properties, such as durability and timeout (among others), and for defining various ways to deliver a text message. We invite the reader to refer to the Javadoc API, provided at `http://docs.jboss.org/hornetq/<version>.Final/api/index.html`, to start training with the various possibilities offered by the core API implementation, which offers lots of functionalities/methods that surpass the JMS message implementation.

Clearly, we also need a producer that will be in charge of sending the message to the selected queue. In this case we will explicitly declare the queue name at this level.

Apart from the constructor of the class, we only have two methods:

```
public void setQueueName(String queuename) throws Exception
public void send(String _message) throws HornetQException,
Exception
```

The `setQueueName(String queuename)` method is in charge of defining the bounded queue for the `MessageProducer` class.

The `send` method somehow mimics the `send` method of the JMS `MessageProducer` class, which we have seen in *Chapter 1*, *Getting Started with HornetQ*.

The code for `setQueueName` is shown as follows:

```
            if (producer == null){
                    producer = MyCoreSession.getSession().
createProducer(queuename);
            }
```

So, this method is in charge of both creating the `MessageProducer` class from the session we have instantiated before and to bind it to the queue specified by the `queuename` string.

So, using a cascade method when you, for the first time, use the `MyCoreMessageProducer` class and you call the `setQueueName` method you will have the following backward calls to the previous static classes:

- A `MyCoreSession.getSession()` call to obtain the `CoreSession` object
- A backward call to the `getClientSessionFactory` method of the `MyCoreClientFactory` class

So, the first time the `MyCoreMessageProducer` class is called, all the core API objects are initialized correctly and then reused without creating new instances; this greatly reduces the creation of new objects and helps in controlling the information flow.

Once we have a `CoreProducer` object correctly set and bound to a queue, we only need to send the message using the `send` method.

The code is shown as follows:

```
message = MyCoreSession.getSession().createMessage(false);
message.putStringProperty("ecg", _message);
producer.send(message);
System.out.println("message sent successfully");
```

First the message needs to be assigned to a session, so we need to re-use our static `getSession` method of the `MyCoreSession` class. The `createMessage` method that we call at the end accepts at least a Boolean type that specifies if the message is durable or not. In our case we will not need the message to stay in the queue until a consumer parses it. Other parameters of the `createMessage` method can be durability, priority, and so on, but again, a good look at Javadoc can give the coder all the possibilities.

On the other hand, we know that it is time for the message to be filled with the text we want to transmit. As mentioned earlier, the `ClientMessage` core object comes with various methods to pass simple types as messages. In fact it is also possible to deliver the so-called large messages that accept an `InputStream` as the body of the message. This makes it possible to directly send the files as messages; we will see this functionality in *Chapter 5*, *Some More Advance Features of HornetQ*. As we saw in *Chapter 2*, *Setting Up HornetQ*, HornetQ uses the `HORNETQ_ROOT\data` folder to physically store the queue; in this case we have a queue named `large-message` where it is possible to deliver `InputStream`. When using this technique, the only limit to the dimension of the message delivered is the amount of disk space available to the `HORNETQ_ROOT\data` folder.

The `HornetQ` user manual claims to have tested the delivery of a message that was 8 GB in size on a machine running with 50 MB of RAM. We never tested a stressful situation and we alert the reader that other parameters should be fine-tuned for performances like this.

Considering such performances and considering that it is possible to use `InputStream` for storing the body of HTML pages via URLs, why not consider HornetQ for creating a fast, multithreaded, and high-performance web spider?

In our case we will use the possibility to store in a message and retrieve in a second moment a key/value pair, like in a Java hashtable structure. In our example the message will be a single three signal ECG measurement and our test message will be a string like the following one:

```
String theECG = "1;02/20/2012 14:01:59.010;1020,1021,1022";
```

We will assign this string as a key/value couple as follows:

```
"ecg"/"1;02/20/2012 14:01:59.010;1020,1021,1022";
```

Only as a reference, it is also possible to write directly to the body of the message using the `getBodyBuffer().writeString(String value)` method.

For large messages we invite the reader to take a look at the `setBodyInputStream` method of the `CoreMessage` class.

Testing the first message sent

Before coding the consumer class, let's test the classes we coded.

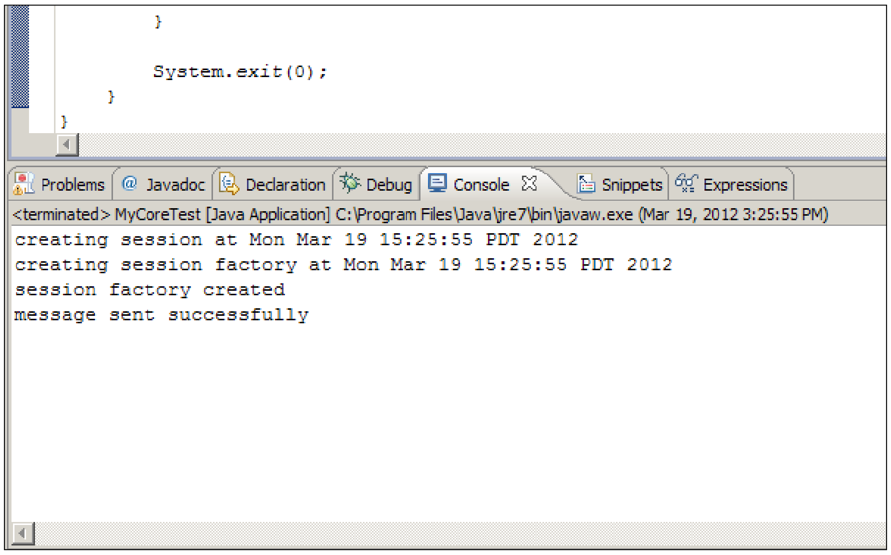
So before moving on, open the `MyCoreTest` class with your IDE and add the following code to the `main` method:

```
public static void main(String[] argvs) throws Exception{

          try {
                MyCoreSession.setSessionParameters("localhost",
5445);

                MyCoreMessageProducer m = new
MyCoreMessageProducer();
                m.setQueuename("jms.queue.myqueue");
                m.send("1;02/20/2012 14:01:59.010;1020,1021,1022");


          } catch (Exception e) {
                e.printStackTrace();
          }
          finally {
                MyCoreSession.close();
                MyCoreClientFactory.close();
          }

          System.exit(0);
    }
}
```

So, as you can see, we first set in the `MyCoreSession` class the host and the port where HornetQ is running; this will create the `ClientSessionFactory` and the `ClientSession` objects in cascade.

Having done that, we create a `MyCoreMessageProducer` object and bind it to a specific queue. Then we will send the message and close the session and connection. I suggest you close the session and the connection in a `finally` block because if you have a runtime exception the next time you re-run the program, the pending session will affect the way you can consume messages.

Our efforts are shown in the following screenshot:



```
            }

            System.exit(0);
        }
}
```

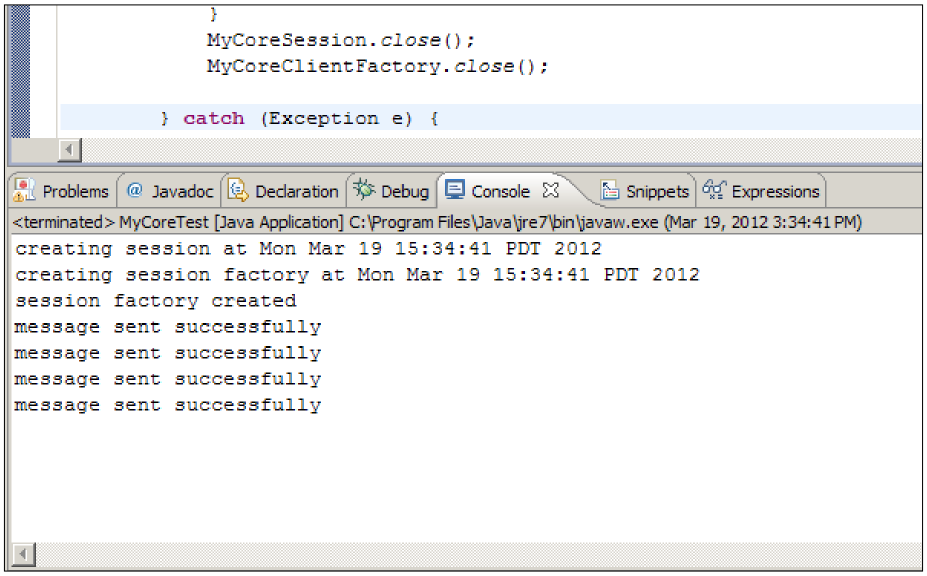
```
<terminated> MyCoreTest [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (Mar 19, 2012 3:25:55 PM)
creating session at Mon Mar 19 15:25:55 PDT 2012
creating session factory at Mon Mar 19 15:25:55 PDT 2012
session factory created
message sent successfully
```

Only to show that the objects are re-used correctly according to the code performance practice we underlined at the beginning, if you change the code to send, for example, five messages in the following way:

```
try {
                MyCoreSession.setSessionParameters("localhost",
5445);
                MyCoreMessageProducer m;

                m = new MyCoreMessageProducer();
                m.setQueuename("jms.queue.myqueue");


                for (int i = 1; i < 5; i++){
                      m.send("1;02/20/2012
14:01:59.010;1020,1021,1022");
                }
                MyCoreSession.close();
```

```
                    MyCoreClientFactory.close();
    } catch(Exception ex) {
        e.printStackTrace();
    }
```

You'll see the following output:



```
            }
            MyCoreSession.close();
            MyCoreClientFactory.close();

        } catch (Exception e) {
```

<terminated> MyCoreTest [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (Mar 19, 2012 3:34:41 PM)

```
creating session at Mon Mar 19 15:34:41 PDT 2012
creating session factory at Mon Mar 19 15:34:41 PDT 2012
session factory created
message sent successfully
message sent successfully
message sent successfully
message sent successfully
```

This confirms that the connection and the session have been created only once, from the inner object to the outermost one.

Coding the HornetQ core API consumer

Let's move to the end of our coding example, consuming the message we have just sent to the queue.

Our final class is the `MyCoreMessageConsumer` class, which is in charge of connecting to the queue and parsing the message that we pushed with the `MessageProducer` class.

In this case we will reuse the same `MyCoreSession` class and hence the `MyCoreClientFactory` class, too.

The class only has the following attributes:

```
private org.hornetq.api.core.client.ClientConsumer consumer;
private org.hornetq.api.core.client.ClientMessage message;
```

The consumer is the one that we will create to receive the messages and `ClientMessage` will be the object where the body of the message will be stored, containing the key/value for the ECG string that we used in the `MessageProducer` class.

The only methods we will implement are the following two:

```
public void getMessages(String queuename) throws HornetQException,
Exception

public void describeMessage() throws IllegalArgumentException,
IllegalAccessException
```

The `getMessage` method accepts the `queuename` and initializes a `MessageConsumer` object on that queue so that we can receive the message. The method is implemented as follows:

```
    public void getMessages(String queuename) throws HornetQException,
Exception{
            consumer = MyCoreSession.getSession().
createConsumer(queuename);
            message = consumer.receive();
            System.out.println("Received TextMessage:" + message.
getStringProperty("ecg"));
    }
```

As you can see, we call the same static method that we used in the `MyCoreProducer` class; so, according to the previous code the session is initialized within the same class and we get the same reference that we have for the message consumer class. With this session we create a consumer bound on the `queuename` queue that is passed as a parameter. While the `MyCoreProducer` class is able to work without exception, even for a queue not defined on the server, the consumer needs to be bound correctly with a queue name.

This behavior could seem strange, but HornetQ bounds queues to the address, so if you send the message to a queue that does not exist, the messages simply disappear. In case you are using the JMS implementation when you try to send a message to a queue that is not defined, you will get an `InvalidDestinationException` exception.

The `CoreClientConsumer` class is equipped with a `receive` method that can be called without a parameter or passing a millisecond integer parameter that will block the receiving of a message until the millisecond timeout has passed. In case the `MessageConsumer` class is blocked to the queue until something happens, the process is destroyed or the network is down.

So, for example, if you want to receive a message and set up a timeout of one second for each message, simply use the following lines along with the code:

```
MyCoreSession.getSession().createConsumer(queuename);
          message = consumer.receive(1000);
```

Once received, we need to parse the message to retrieve the correct key/value pair. This is done by using the `getStringProperty` method and recalling the correct key identifier, which in our case is the ECG string.

If you try to access a key not stored in the message object you will have a null value returned, so a better way to read a property of a message is to use the `containsProperty` method.

A more correct code follows:

```
          if (message.containsProperty("ecg"))
                System.out.println("Received TextMessage:" +
message.getStringProperty("ecg"));
```

We also add a method to describe all the fields of the message so that we have a fast method for logging some important attributes of the message to the console.

This method uses reflection to find the fields of the `ClientMessage` object inside our `MyCoreMessageConsumer` class. We will call the method on the superclass, after setting the visibility of every field to avoid problems in accessing protected/private attributes.

Reflection is one of the great possibilities that Java offers to the developer, so we will only give a simple example and leave the reader any possible future implementation that they want to make.

The code for the `describeMessage()` method is the following:

```
          Class<?> parentClass = this.message.getClass().
getSuperclass();

          Field[] fields = parentClass.getDeclaredFields();
          for (Field field : fields) {
                field.setAccessible(true);
                if (field.get(this.message) != null)
                      System.out.println("field: " + field.getName()
+ " : " + field.get(this.message).toString()  );
                else
                      System.out.println("field: " + field.getName()
+ " : null");
          }
```

Some fields can have a null value, so we need to control them to avoid null reference exceptions at runtime.

Considering the field property that we can set on a `CoreMessage` object, we need to point out the most useful ones in a transactional environment:

- `messageID`: This is a unique identifier that is automatically assigned when the message is produced
- `servertimestamp`: This is the long timestamp value corresponding to the time this message was handled by a HornetQ server
- `UserID`: This is an optional user-specified long value that can be set to identify the message and will be passed around with the message
- `Priority`: This is a byte-valued field that ranges from 0 (less priority and default value) to 9 (more priority), inclusive of both

Putting everything together

Now that we have also coded the `MyCoreMessageConsumer` class, we are ready to put everything together to have a fully working example. So we reopen our `MyCoreTest` class to add the functionality of the `MyCoreMessageConsumer` class.

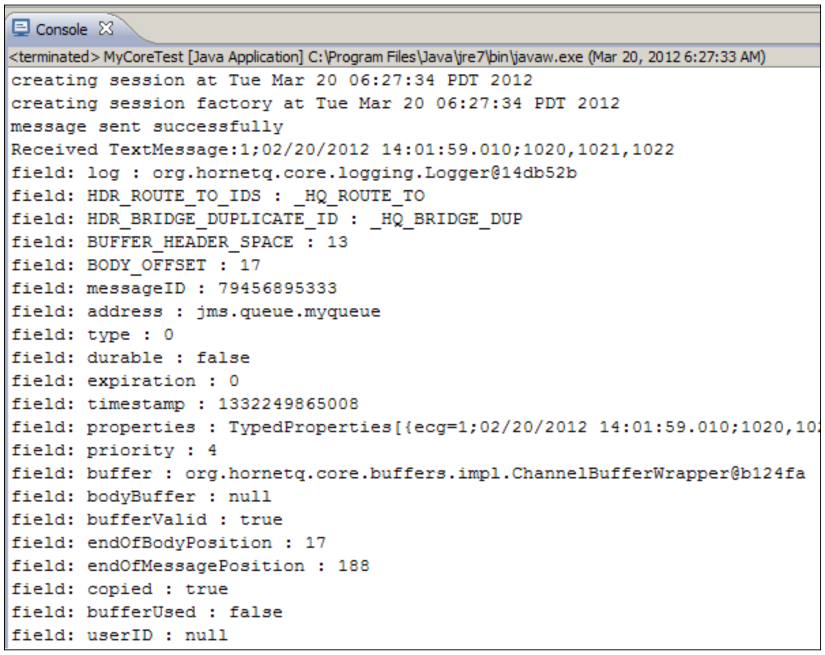
The complete code for the `main` method is shown as follows:

```
    public static void main(String[] argvs) throws Exception{

            try {
                    MyCoreSession.setSessionParameters("localhost",
5445);

                    MyCoreMessageProducer m = new
MyCoreMessageProducer();
                    m.setQueuename("jms.queue.myqueue");
                    m.send("1;02/20/2012 14:01:59.010;1020,1021,1022");

                    MyCoreSession.start();
                    MyCoreMessageConsumer c = new
MyCoreMessageConsumer();
                    c.getMessages("jms.queue.myqueue");
                    c.describeMessage();

            } catch (Exception e) {
                    e.printStackTrace();
            }
            finally {
                    MyCoreSession.close();
                    MyCoreClientFactory.close();
            }

            System.exit(0);
    }
```

The following are the steps to follow:

1. First we start the `ClientSession` object bind inside the `MyCoreSession` class.
2. Then we create the `MyCoreMessageConsumer` object (so that it will be possible to use) and call the `getMessages` method on the queue we used to produce the messages.
3. Lastly, we describe the message received and close the `ClientSession` and `ClientFactory` objects.

The output of our efforts is the following console:



```
<terminated> MyCoreTest [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (Mar 20, 2012 6:27:33 AM)
creating session at Tue Mar 20 06:27:34 PDT 2012
creating session factory at Tue Mar 20 06:27:34 PDT 2012
message sent successfully
Received TextMessage:1;02/20/2012 14:01:59.010;1020,1021,1022
field: log : org.hornetq.core.logging.Logger@14db52b
field: HDR_ROUTE_TO_IDS : _HQ_ROUTE_TO
field: HDR_BRIDGE_DUPLICATE_ID : _HQ_BRIDGE_DUP
field: BUFFER_HEADER_SPACE : 13
field: BODY_OFFSET : 17
field: messageID : 79456895333
field: address : jms.queue.myqueue
field: type : 0
field: durable : false
field: expiration : 0
field: timestamp : 1332249865008
field: properties : TypedProperties[{ecg=1;02/20/2012 14:01:59.010;1020,10
field: priority : 4
field: buffer : org.hornetq.core.buffers.impl.ChannelBufferWrapper@b124fa
field: bodyBuffer : null
field: bufferValid : true
field: endOfBodyPosition : 17
field: endOfMessagePosition : 188
field: copied : true
field: bufferUsed : false
field: userID : null
```

Our example using only HornetQ core API is over, but we need to underline some remarks.

Final considerations

There are some remarks that we should note to prevent you from possibly misunderstanding how the core messages work.

First we need to point out that, in the implementation done in this chapter, we assume implicitly that the messages are produced in a synchronous fashion by the consumer. This means that even if we produce messages in a series, only the last one is consumed.

To arrange a `MessageConsumer` class that works in an asynchronous way, we need to detail how the message is consumed in a better way.

If a `MessageConsumer` class needs to access a queue in an asynchronous way, the solution is to have a Java `EventHandler` class that somehow fires up when a new message is read.

So the `MessageConsumer` class has the `setMessageHandler` method. This is the method that accepts as a parameter a class that implements the `MessageHandler` interface. The `MessageHandler` interface has only one method that needs to be implemented and that will be fired up once a new message is consumed asynchronously.

To set the asynchronicity for a generic consumer, you can refer to the following code:

```
ClientConsumer consumer = session.createConsumer("jms.queue.myqueue");
consumer.setMessageHandler(new myMessageHandler());
```

As you can see, we created a new class that implements the `MessageHandler` interface. One possible implementation is:

```
import java.util.Date;

import org.hornetq.api.core.client.ClientMessage;
import org.hornetq.api.core.client.MessageHandler;

public class myMessageHandler implements MessageHandler {

   @Override
   public void onMessage(ClientMessage message) {
          // TODO Auto-generated method stub
          System.out.println("received message " + message.
getMessageID() + " at  " + new Date(message.getTimestamp()));
   }

}
```

Once a new message is consumed in the `onMessage` method, it is fired by passing the new `ClientMessage` object. In our implementation, we have to only write to the console, the `messageID`, and the `timestamp` of the message received by the HornetQ server.

Only to display some information, we convert to a `Date` format, but you can do anything you need for implementing the logic layer of your application.

Have you done your math?

Lastly, we would like to suggest some possible improvements on the code, hoping that fighting against some first-use difficulties could help with a deeper understanding of the layer interaction in HornetQ.

First, an obvious implementation of the code done previously is to add the possibility in the `MyCoreMessageConsumer` class to consume messages asynchronously by adding a new method and a new `MessageHandler` implementation. Using the class we coded, it would be nice to create a client producer that sends messages and a client consumer that runs on a separate JVM that reads messages asynchronously.

Another good exercise is to add to the consumer the MongoDB interaction that we worked on in *Chapter 1*, *Getting Started with HornetQ*. This can be done by implementing the `getMessages` method of the `MyCoreMessageConsumer` class, both synchronously and asynchronously.

The code we had can be improved in several ways. We invite the reader to change it in the spirit of the open source community.

JMS messaging is implemented by HornetQ but core API is much more efficient because it works with low level machine capacity but has some limitations; for example, core messages are just able to send byte messages with aggregated properties. So you need to think which will be the best solution for your project.

Summary

In this chapter we learned how to install a development environment, create a set of reusable classes for managing messages using the core API, and saw some specific properties of a core API message.

We are now ready to cover the management of the HornetQ server using the management API. We will understand how it is possible to use code to control and view what is happening to a HornetQ server object like a queue while it is running.

4
Monitoring HornetQ

We have seen in the previous chapter how to push/read messages in HornetQ using JMS or the core API. But apart from these two basic operations we have not seen any other interaction.

If you are reading this book you are probably planning to adopt HornetQ because you need to manage lots of messages/queues in a fast-updating environment. For example, counting how many messages there are in a queue is another simple task that overpasses the knowledge that we have.

This chapter is devoted to illustrating how it is possible to control the state and the behavior of the objects of a HornetQ server.

This chapter will cover the following topics:

- Managing HornetQ: In this topic we will see what are the core APIs devoted for monitoring purposes
- Interacting using JMS: In this topic we will see the JMS counterpart of the monitoring API
- Managing HornetQ using JMX: In this topic we will use the standard management API provided by the JDK to do the same thing

Each of these points will be covered using an example that we will describe in detail. We assume for the moment that you have a standalone non-clustered HornetQ server running on your computer, as for the previous chapters. In *Chapter 3, Basic Coding with HornetQ: Creating and Consuming Messages*, we focused on Eclipse; in this chapter we will use NetBeans under Ubuntu. The code can be ported to an Eclipse environment without heavy modifications. The reader could refer to the book's code for the Eclipse environment. So let us start with the core API management.

Managing HornetQ using the core API

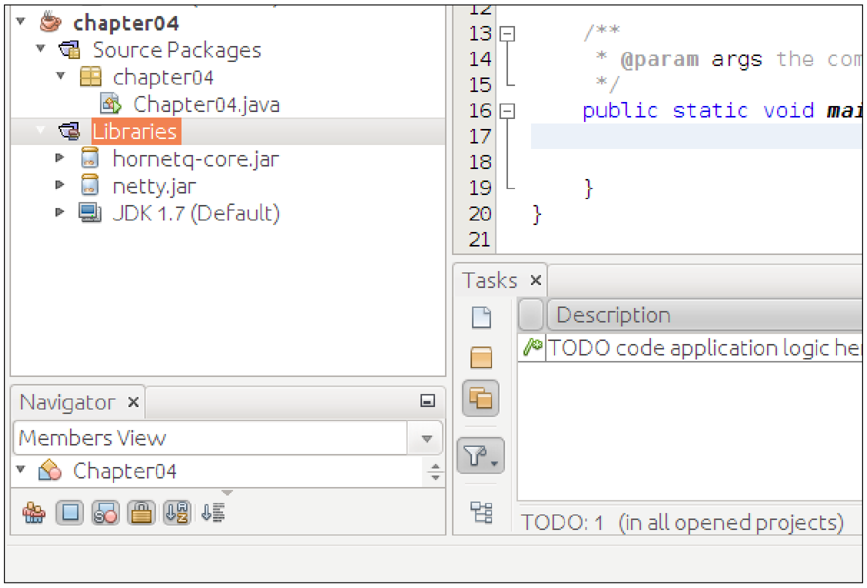
The first example that we will illustrate is for creating a simple class that will use the core HornetQ management API to count how many messages are stored on a queue of a local HornetQ standalone server. In this class we will also create and deploy a new queue at runtime.

Configuring the environment

To set up your environment, in our case NetBeans, for developing the class we will only need to add to the project classpath, the `hornetq-core.jar` file and the `netty.jar` file, that can be found in the `HORNETQ_ROOT\lib` folder as usual.

To do this on NetBeans you can follow these steps:

1. Once you have fired up NetBeans, go to **File** | **New Project** | **Java application** and enter `chapter04` as the name.
2. Also add a `main` method.
3. Right-click on the **Libraries** item of the `chapter04` project just created and choose from the popup menu **Add** | **jar/folder**.
4. Select the library `hornetq-core.jar`, and the `netty.jar` file in the `HORNETQ_ROOT\lib` folder, so you should have the following project screenshot:



```
    /**
     * @param args the com
     */
    public static void mai

    }
}
```

Before using the management core API classes

Before moving on to the coding phase we need to change some default configurations. This is because management operations are not allowed to be done by every client that connects to the HornetQ instance. Besides this, HornetQ interacts with a particular queue for managing different operations so we need to make some changes in the `hornetq-configuration.xml` file.

We will cover in detail what is the correct way to configure the management queue to avoid any non-authorized access in *Chapter 8*, *Controlling Message Flow*. For the moment, we will use a brute-force approach by disabling any HornetQ controls about security. Otherwise we could have runtime issues when connecting to the management queue to read information about the queue itself.

So for having your HornetQ standalone non-clustered server instance correctly configured you need to open the `hornetq-configuration.xml` file using your text editor in the `HORNETQ_ROOT\config\stand-alone\non-clustered\` folder. Once opened, add the following code before the `</configuration>` tag:

```
<security-enabled>false</security-enabled>
<management-address>jms.queue.hornetq.management</management-address>
```

Using this directive we told HornetQ not to bother about the client's security settings.

So you are now ready to fire up the HornetQ server using the `run.sh` file in the `HORNETQ_ROOT\bin` folder. Let us start coding.

First example using the core

To manage operations within the core API we first need some description that will also be useful while dealing with JMS. All the management operations are arranged by a meta-queue called **management queue**. On this queue you perform the usual operations like pushing and reading messages, but in this case the message that comes out as a reply after a push operation is a message containing information on other HornetQ queue addresses. In our example, we will use the management queue to find some information on the number of messages present on a queue specified in the main configuration file.

The operation that will be performed will be the following:

- Connecting to the HornetQ standalone non-clustered server
- Producing and sending some messages on a queue

- Connecting to the management queue
- Pushing some requests to the management queue
- Reading the results

So let us get started:

1. We will reuse the same code from *Chapter 3, Basic Coding with HornetQ: Creating and Consuming Messages* using the `TransportConfiguration` class. So with the following lines of code you should have a valid session for your HornetQ localhost instance:

```
        ServerLocator serverLocator = null;
        ClientSessionFactory factory = null;
        ClientSession session = null;

        HashMap map = new HashMap();
        map.put("host", "localhost");
        map.put("port", 5445);
        try{
        TransportConfiguration configuration = new TransportConfig
uration(NettyConnectorFactory.class.getName(), map);

        serverLocator = HornetQClient.createServerLocatorWithoutHA
(configuration);
        factory = serverLocator.createSessionFactory();

        session = factory.createSession(false, false, false);
session.start();
```

2. Now it is time to push some messages; we will use a simple `for` loop to push a lot of messages in a small amount of time so that we can start looking at how HornetQ is performing. Remember to configure the core queue name in the `hornetq-configuration.xml` file (in our case `queue.exampleQueue`).

```
        ClientMessage message = session.createMessage(false);

        final String queueName = "queue.exampleQueue";
        ClientProducer producer = session.
createProducer(queueName);

        for (int i = 0; i < 1000;i++){
            message.getBodyBuffer().clear();
            message.getBodyBuffer().writeString("new message sent
at " + new Date());
            System.out.println("Sending the message.");
            producer.send(message);
        }

        session.close();
```

Please notice that we use the `getBodyBuffer()` method to load the string into the message without using the `puproperty` method, as we did in *Chapter 3, Basic Coding with HornetQ: Creating and Consuming Messages*. This is another opportunity offered by the core API.

3. Now that we are connected to HornetQ and we have loaded some messages for testing purposes, it is time to go through and see how it is possible to query the management queue to get some data. The following code illustrates how to do this:

```
        ClientSession managedsession = factory.createSession();
        ClientRequestor requestor = new
ClientRequestor(managedsession, "hornetq.management");
        ClientMessage messagemanaged = managedsession.
createMessage(false);
        ManagementHelper.putAttribute(messagemanaged, queueName,
"messageCount");
        ClientMessage reply = requestor.request(messagemanaged);
        int count = (Integer) ManagementHelper.getResult(reply);
        System.out.println("There are " + count + " messages in
exampleQueue");
```

Let us go through the code step by step:

1. Before doing anything you need to create a session to connect to the HornetQ instance, so we reuse the `connectionFactory` object, which was just instantiated.
2. Once we connect, we instantiate the `ClientRequestor` class that is the class in charge of associating the session with the management queue and will be in charge of sending the meta-request about the queue to monitor. The constructor is given as follows:

```
        ClientRequestor requestor = new
ClientRequestor(managedsession, "hornetq.management");
```

3. After this we create a `ClientMessage` object to store the request that we need to send to the management queue.
4. The next step is the most important. Using the static method of the `ManagementHelper` class, we put a property that is the `messageCount` property and the value is the queue we want to monitor using the code:

```
        ManagementHelper.putAttribute(messagemanaged,
queueName, "messageCount");
```

5. We will create a message that will be put on the `hornetq.management` queue asking for the messages present on the `queue.exampleQueue`.
6. We now need to push the message into the `hornetq.management` queue. This is done using the following code:

```
ClientMessage reply = requestor.request(messagemanaged);
```

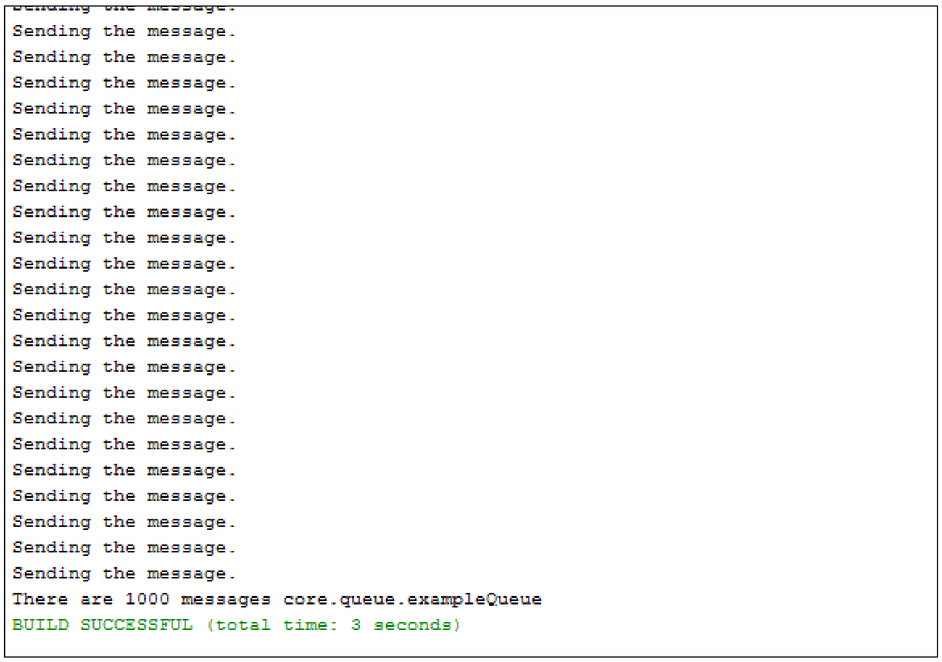
Notice that the reply to the request is another `ClientMessage` object that you can manipulate and consume using the method, provided in this case, we pass the reply object to the `ManagementHelper` class to get the reply to the request we have pushed.

4. The `ManagementHelper` object provides a more structured way to arrange requests and error messages as you can see in the following lines of code:

```
        if (reply != null) {
                if (ManagementHelper.hasOperationSucceeded(reply))
{

                        int count = (Integer) ManagementHelper.
getResult(reply);
                        System.out.println("There are " + count +
" messages in exampleQueue");
                }
                else
                {
                        String error = (String) ManagementHelper.
getResult(reply);
                        System.out.println("exception : " +
error);
                }

        } else {
                System.out.println("Server did not replly see log
for errors");
        }
```

So if the `ClientMessage reply` object has been successfully created, you can go through to see if the request operation succeeded or otherwise get a more detailed error message.

If you have performed all the steps correctly the message should be similar to the one displayed in the following screenshot:



```
Sending the message.
Sending the message.
Sending the message.
Sending the message.
Sending the message.
Sending the message.
Sending the message.
Sending the message.
Sending the message.
Sending the message.
Sending the message.
Sending the message.
Sending the message.
Sending the message.
Sending the message.
Sending the message.
Sending the message.
Sending the message.
Sending the message.
Sending the message.
Sending the message.
Sending the message.
There are 1000 messages core.queue.exampleQueue
BUILD SUCCESSFUL (total time: 3 seconds)
```

We underline that the reader should get familiar with the `ManagementHelper` class and the `ResourceNames` class (`http://docs.jboss.org/hornetq/2.2.14.Final/api/index.html`). For example, if you want to query the management queue for having information on the server version, you could use the following code:

```
ManagementHelper.putAttribute(messagemanaged,ResourceNames.CORE_
SERVER, "version");
```

We can now move on to using the HornetQ management API to JMS management API.

JMS management API

The general settings that we had for managing HornetQ using the core API are nearly the same for JMS settings. As we explained in *Chapter 2, Setting Up HornetQ*, HornetQ uses core APIs internally to manage the JMS requests so the only thing that changes is how to call the management API in the correct way.

So we are now ready to re-code the previous example in the JMS dialect only to underline the differences.

Before doing anything we need to create the queue used for testing and to tell HornetQ how to treat the security of the management queue:

1. Fire up your preferred text editor and open the `hornetq-configuration.xml` file located in the `HORNETQ_ROOT\config\stand-alone\non-clustered` folder, and add the following tag into the `<security-settings>` tag:

    ```
    <security-setting match="jms.queue.hornetq.management">
       <permission type="manage" roles="guest" />
    </security-setting>
    ```

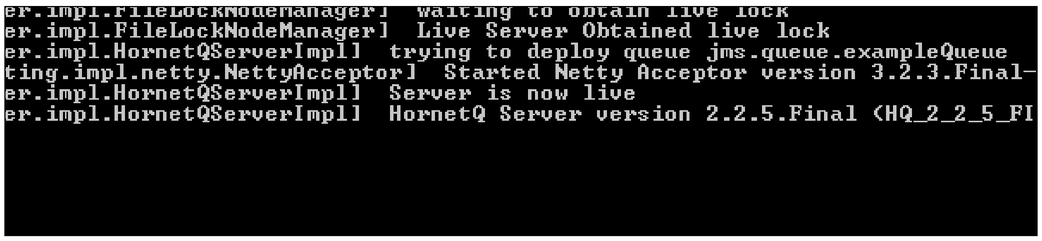
    The lowest trusted role of security is the guest role. In this case we provide the possibility to access and use the `hornetq.management` queue to every client that will be connected to HornetQ. Remember that forgetting to insert the XML tag `<security-setting>`, as shown previously, will cause a runtime authorization problem on the code. In this case, following the specification we add the prefix `jms.queue`.

2. We also need to define the queue that we will monitor, so open the `hornetq-jms.xml` file and add the following XML tag:

    ```
    <queue name="exampleQueue">
       <entry name="/queue/exampleQueue"/>
    </queue>
    ```

    Remember that you can also add this tag in the `hornetq-configuration.xml` file, but it is always a good idea in a production environment to separate the core queue definitions from JMS queue definitions.

    Now that we are ready to fire up your HornetQ instance, you should see the example queue deployed as the following screenshot:



```
er.impl.FileLockNodeManager]  Waiting to obtain live lock
er.impl.FileLockNodeManager]  Live Server Obtained live lock
er.impl.HornetQServerImpl]  trying to deploy queue jms.queue.exampleQueue
ting.impl.netty.NettyAcceptor]  Started Netty Acceptor version 3.2.3.Final-
er.impl.HornetQServerImpl]  Server is now live
er.impl.HornetQServerImpl]  HornetQ Server version 2.2.5.Final (HQ_2_2_5_FI
```

3. Now fire up NetBeans and create a new class inside the `chapter04` package that you created previously. Call this class `JMSManagementExample` with a `main` method.

4. Next, do not forget to add the `hornetq-jms.jar` file to the external JAR library, otherwise you will have a lot of compile-time errors.

Now that we are ready, we can code an example to demonstrate the use of JMS for management purposes.

Calling the JMS management API

The steps we are going to follow for our example are:

1. Connecting to the JMS queue.
2. Producing a message and sending it to the queue.
3. Counting how many messages are stored in the queue.

We will also perform some more advanced tasks like removing some messages from the queue itself before they are consumed by some `MessageConsumer`.

1. So let us start with the connection to the `exampleQueue` queue. We will add the following lines of code in the `hornetq-jms.xml` file:

```
      QueueConnection connection = null;
      InitialContext initialContext = null;
      try
      {

java.util.Properties p = new java.util.Properties();

p.put(javax.naming.Context.INITIAL_CONTEXT_FACTORY,
"org.jnp.interfaces.NamingContextFactory");

p.put(javax.naming.Context.URL_PKG_PREFIXES,
"org.jboss.naming:org.jnp.interfaces");

p.put(javax.naming.Context.PROVIDER_URL, "jnp://localhost:1099");

                    initialContext = new javax.naming.
InitialContext(p);


         Queue queue = (Queue)initialContext.lookup("/queue/
exampleQueue");

         QueueConnectionFactory cf = (QueueConnectionFactory)
initialContext.lookup("/ConnectionFactory");

         connection = cf.createQueueConnection();
         connection.start();

         QueueSession session = connection.
createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
```

As we have seen in *Chapter 1, Getting Started with HornetQ*, we need to use the JNDI lookup to find the queue name.

2. After creating the connection we create a session with the `Session.AUTO_ACKNOWLEDGE` mode.
3. Next we can push some fake messages on the queue, only for the purpose of demonstration, as we did in the previous session.

```
         MessageProducer producer = session.createProducer(queue);

         TextMessage message = session.createTextMessage("This is 
a text message");

               for (int i = 0; i<1000; i++) {
                     message = session.createTextMessage("This is a 
text message");
                     producer.send(message);
                     System.out.println("Sent message: " + message.
getText() + " " + new Date());
               }
```

4. We can now retrieve the `messageCount` property of these messages:

```
         Queue managementQueue = HornetQJMSClient.
createQueue("hornetq.management");

         QueueRequestor requestor = new QueueRequestor(session, 
managementQueue);

         Message m = session.createMessage();

         JMSManagementHelper.putAttribute(m, "jms.queue.
exampleQueue", "messageCount");

         Message reply = requestor.request(m);

         int messageCount = (Integer)JMSManagementHelper.
getResult(reply);
         System.out.println(queue.getQueueName() + " contains " + 
messageCount + " messages");
```

As we saw for the core API, even for JMS we push a message into the `HornetQ.management` queue. The message will ask HornetQ the `messageCount` property of a specific JMS queue.

It is important to notice that even if the management queue accepts messages as "normal", queues have some different publishing behaviors. One difference is that you do not need to use JNDI lookup to find the management queue, but you can instantiate and use it directly.

5. We are now ready to see some more advanced operations for the JMS queue. We will remove the last message that we produce on the `jms.queue.exampleQueue` queue. The idea is to read the message's unique identifier, which is provided when it is pushed into the queue, use it and the `putOperationInvocation` method of the `JMSManagementHelper` class, and tell HornetQ to remove the message from the queue with that ID. The code is only a development of the idea explained. Here it is:

```
         m = session.createMessage();
         JMSManagementHelper.putOperationInvocation(m,
                                                  "jms.queue.
exampleQueue", "removeMessage",
                                                message.
getJMSMessageID());
reply = requestor.request(m);
       if (reply != null){
             if (JMSManagementHelper.hasOperationSucceeded(reply))
{
                   System.out.println("operation invocation has
succeeded");
                   System.out.println("The message with ID" +
message.getJMSMessageID() +" have been successfully deleted");
             }
             else
             {
                   String error = (String) ManagementHelper.
getResult(reply);
                   System.out.println("exception : " + error);
             }
       } else {
             System.out.println("the operation invoked reply with
null");
       }
```

The first operation is to create another message that will handle the next request, then we simply call the static method `putOperationInvocation` of the `JMSManagementHelper` class.

The parameters are self-explanatory; apart from the message `m`, we define the queue where we want to perform the operation. Next we define the operation we want to do in the previous object and then the eventual parameter needed by the operation.

We need to point out that the `putOperationInvocation` method acts using a general object, in this case a `JMSQueueControl` interface, and the operations are the methods provided by the interface itself. The reader should refer to the HornetQ management API doc (`http://docs.jboss.org/hornetq/2.2.14.Final/api/org/hornetq/api/jms/management/JMSQueueControl.html`) to view all the possibilities. In this case the invocation corresponds to the calling of the method `removeMessage(String messageID)`, if called on a class implementing the `JMSQueueControl` interface. As we did in the previous example we manage the reply of the HornetQ management queue by controlling if it is a valid and instantiated `ClientMessage` object, and in this case by controlling if the operation has succeeded and by printing the `messageID`. Otherwise we print the error to the standard output console.

6. Next we close all the resources in a controlled way using the portion of the following code:

```
if (initialContext != null)
{
   initialContext.close();
}
if (connection != null)
{
   connection.close();
}
```

Only one final consideration on using the `putOperationInvocation` method—considering that a queue can be monitored while the messages are consumed, both synchronously and asynchronously by a `ClientConsumer`, some operations could lead to runtime errors. In our case everything went well because we removed the message while no other resources were consuming it on the same queue.

So in most production environments even removing a message could be an action that needs the queue to be suspended from consuming messages.

Considering that the `JMSQueueControl` interface defines a method called `paused()` that pauses the queue so that the messages are no longer delivered to its consumers, are you able to implement a safe remove message on one queue using the previous code?

The reader should exercise on recoding the same functionality using the core example.

Managing clients using JMX

The last example that we will describe in this chapter will answer a common problem when we have a software that dials between them using a client/server approach—who are the clients connected to a server and what are they doing at the moment?

HornetQ provides JMX interfaces to allow the management of the clients connected to a HornetQ server. The JMX can be accessed as HornetQ exposes an `MBeanServer` that can be run on the same virtual machine.

So in our final example we will explain how to list all the clients connected to our standalone HornetQ server and how to disconnect each one of them.

Setting up the server

We need some configuration for enabling the `MBeanServer` to work correctly:

1. First you need to check whether the `hornetq-configuration.xml` file contains the following line:

   ```
   <jmx-management-enabled>false</jmx-management-enabled>
   ```

   Because if it does, the management is disabled (by default in HornetQ it is enabled).

2. As the `MBeanServer` runs within the JVM machine that launches HornetQ, you need to provide the following JVM system properties:
   - `-Dcom.sun.management.jmxremote`
   - `-Dcom.sun.management.jmxremote.port=3000`
   - `-Dcom.sun.management.jmxremote.ssl=false`
   - `-Dcom.sun.management.jmxremote.authenticate=false`

3. To allow this, open the `run.bat` file (in Windows) or `run.sh` file (in Linux) and modify the system variable `JVM_ARGS`. In Windows, we have the following code:

   ```
   set JVM_ARGS=%CLUSTER_PROPS% -XX:+UseParallelGC
   -XX:+AggressiveOpts -XX:+UseFastAccessorMethods -Xms512M -Xmx1024M
   -Dhornetq.config.dir=%CONFIG_DIR%  -Djava.util.logging.config.
   file=%CONFIG_DIR%\logging.properties -Djava.library.path=. -Dcom.
   sun.management.jmxremote -Dcom.sun.management.jmxremote.port=3000
   -Dcom.sun.management.jmxremote.ssl=false
   Dcom.sun.management.jmxremote.authenticate=false
   ```

In Linux we have the following code:

```
export JVM_ARGS="$CLUSTER_PROPS -XX:+UseParallelGC
-XX:+AggressiveOpts -XX:+UseFastAccessorMethods -Xms512M -Xmx1024M
-Dhornetq.config.dir=$CONFIG_DIR -Djava.util.logging.config.
file=$CONFIG_DIR/logging.properties -Djava.library.path=."
-Dcom.sun.management.jmxremoteDcom.sun.management.jmxremote.
port=3000Dcom.sun.management.jmxremote.ssl=false
Dcom.sun.management.jmxremote.authenticate=false
```

4. In the last step, create a JMS queue in the `hornetq-jms.xml` file and launch the server.

Listing all the clients connected to a queue using JMX

We are ready to code the Java class that will list all the clients connected to our HornetQ server. To do this perform the following steps:

1. First create a new class in the `chapter04` package with a static `main` method.
2. Then add the following JAR files from the `HORNETQ_ROOT\lib` folder: `hornetq-core.jar`, `HornetQ-jms.jar`, `jnp-client.jar`, `netty.jar`, and `jboss-jms-api.jar`.
3. Now, from your class, connect to the server using JNDI and create a connection. You can do this using the following code:

```
        java.util.Properties p = new java.util.Properties();

        p.put(javax.naming.Context.INITIAL_CONTEXT_FACTORY,
        "org.jnp.interfaces.NamingContextFactory");

        p.put(javax.naming.Context.URL_PKG_PREFIXES,
        "org.jboss.naming:org.jnp.interfaces");

        p.put(javax.naming.Context.PROVIDER_URL, "jnp://
localhost:1099");


        initialContext = new javax.naming.InitialContext(p);
        QueueConnectionFactory cf = (QueueConnectionFactory)
initialContext.lookup("/ConnectionFactory");
        connection = cf.createQueueConnection();
connection.start();
```

4. To display the client's IP address we need to have a valid `HornetQServerControl` object. The code to do this is the following one:

```
        String JMX_URL = "service:jmx:rmi:///jndi/rmi://
localhost:3000/jmxrmi";

        ObjectName on = ObjectNameBuilder.DEFAULT.
getHornetQServerObjectName();
        JMXConnector connector = JMXConnectorFactory.connect(new
JMXServiceURL(JMX_URL), new HashMap<String, String>());
        MBeanServerConnection mbsc = connector.
getMBeanServerConnection();
        HornetQServerControl serverControl =
(HornetQServerControl)MBeanServerInvocationHandler.newProxyInstanc
e(mbsc,on,HornetQServerControl.class,false);
```

   The operations in sequence are:

   1. Instantiating a `JMXConnector` object using the `JMX_URL` string provided by the JMX string connection syntax.
   2. Instantiating a `HornetQServerControl` object using a `MBeanServerConnection`.

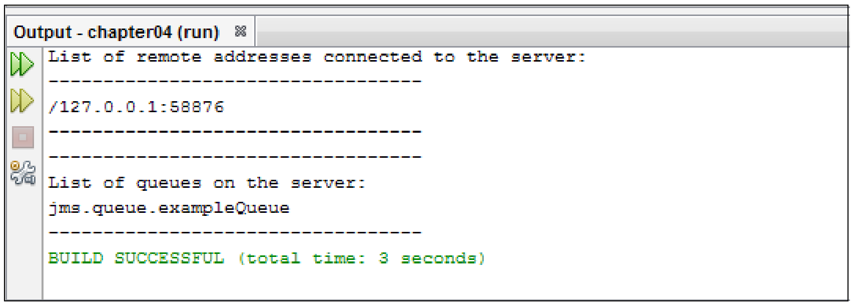
5. The final result is an object where we can get information on many HornetQ objects. For example, using the `listRemoteAddresses` method we can have some information on the client that is connected, as shown in the following lines of code:

```
         System.out.println("List of remote addresses connected to
the server:");
         System.out.println("----------------------------------");
         String[] remoteAddresses = serverControl.
listRemoteAddresses();
         for (String remoteAddress : remoteAddresses)
         {
            System.out.println(remoteAddress);
         }
         System.out.println("----------------------------------");
```

   Or get every queue managed by the server using the following lines:

```
         System.out.println("----------------------------------");
         System.out.println("List of queues on the server:");

         String[] queueNames = serverControl.getQueueNames();
         for (String queueName : queueNames)
         {
            System.out.println(queueName);
         }
         System.out.println("----------------------------------");
```

The standard output shows the following results:



```
List of remote addresses connected to the server:
----------------------------------
/127.0.0.1:58876
----------------------------------
----------------------------------
List of queues on the server:
jms.queue.exampleQueue
----------------------------------
BUILD SUCCESSFUL (total time: 3 seconds)
```

As a final exercise we invite you to use both the JMX and JMS management API to develop a class that lists all the queues on our standalone non-clustered server and displays all the messages present in these queues.

Summary

In this chapter, we have seen that there are different possibilities for monitoring the HornetQ server. We suggest you try using Javadoc to train yourself on combining JMX and JMS management to see the possible combinations. For example, even if we have not covered this possibility, it is also possible to trigger some events on the server using JMX management API, such as suspending or activating some events when something happens on a queue.

Now that we have seen how to manage the queue and the messages within them, we can move ahead to see some more advanced features available in HornetQ. However, I hope that this chapter will inspire the reader to go deeper into the thematics of management.

Considering that HornetQ can also be used in a cluster environment for high-performance purposes, detecting what happens on a single or multiple instance of the server is an essential task to be addressed.

Now that we have seen how to control our queues and the HornetQ server instance, we are ready to move to some more advance features concerning messages. The next chapter will illustrate how to send files as messages and how to manage the delivery of messages.

5 Some More Advanced Features of HornetQ

The last chapter was a necessary break for the reader. In this chapter we will continue with the argument treated in *Chapter 3*, *Basic Coding with HornetQ: Creating and Consuming Messages*. HornetQ implements a rich set of features concerning the managing of the messages themselves, so this chapter will illustrate how to use this feature using, as usual, our book's example.

We will briefly recall the main scenario: we have an **Electrocardiography** (**ECG**) device that sends messages containing the three-signal ECG measurements of the patient's heart and stores them into a HornetQ queue. On the consumer's backend there is an application that reads the messages and stores them in the MongoDB database.

In this chapter, we will implement our example using HornetQ features to treat some real world needs:

- Managing large messages
- Managing re-delivery
- Managing priority

At this point the reader should be aware that for each one of these features there will be the HornetQ core API and the HornetQ JMS example. We will cover both of them underlining the differences and the configuration issues.

We will use the NetBeans development environment under Ubuntu but the examples are also coded in an Eclipse project available at this book's code download page.

Let us begin with the large messages.

Managing large messages with HornetQ

Up until now we have managed messages that were simple strings like the following one:

```
"ecg"/"1;02/20/2012 14:01:59.010;1020,1021,1022";
```

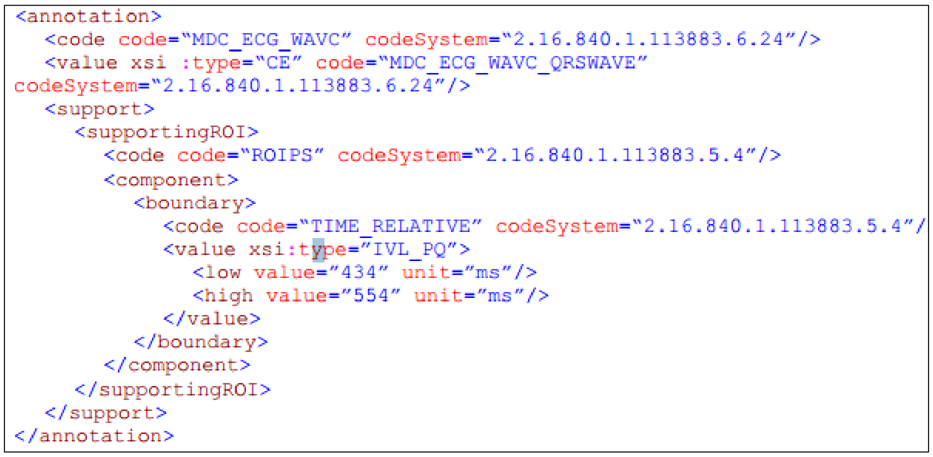
But as we pointed out in *Chapter 3, Basic Coding with HornetQ: Creating and Consuming Messages*, HornetQ is able to manage messages that can reach the magnitude of gigabytes using its internal storing system.

In this section, we will see how to pass an entire file to a HornetQ queue so that the message is stored in the file itself and not a single string.

Using our reference example, the need arises from the fact that modern medical equipments neither store nor send ECG messages directly into string format but use some predefined medical format.

We will not go into further details but even if there is no agreed format for the ECG measurement, we will use the most widely used format that is also an ANSI standard for medical measurements named **HL7** (`www.hl7.org`). The main idea behind HL7 is to store the information concerning the medical application (so not limited to ECG) in an agreed format that can be implemented in different scenarios. Using such an agreed standard, it is possible for different stakeholders to implement their software in a way in which it is possible to share information with others with minimal effort.

From the developer's side, an HL7 annotated ECG message is nothing more than an XML file that follows the HL7 standard. See the following ECG QRS wave annotation as an example:



```
<annotation>
  <code code=“MDC_ECG_WAVC” codeSystem=“2.16.840.1.113883.6.24”/>
  <value xsi :type=“CE” code=“MDC_ECG_WAVC_QRSWAVE”
codeSystem=“2.16.840.1.113883.6.24”/>
  <support>
    <supportingROI>
      <code code=“ROIPS” codeSystem=“2.16.840.1.113883.5.4”/>
      <component>
        <boundary>
          <code code=“TIME_RELATIVE” codeSystem=“2.16.840.1.113883.5.4”/
          <value xsi:type=”IVL_PQ”>
            <low value=”434” unit=”ms”/>
            <high value=”554” unit=”ms”/>
          </value>
        </boundary>
      </component>
    </supportingROI>
  </support>
</annotation>
```

The QRS wave represents three different waves that when combined show the center peak of the ECG signal, so it codes the most important information for a clinician. Again we point out that in this example we will only show how to store a message and read it, and this message will be the XML file shown previously, without going so much into clinical details.

We are now ready to illustrate two of our examples. So let us start with JMS.

Managing large messages using JMS

HornetQ stores large messages into the file system of the server where the instance is running using the path configured in the `HornetQ-configuration.xml` file. If you open it with your preferred text editor you should see the following tag:

```
<large-messages-directory>${data.dir:../data}/large-messages</large-
messages-directory>
```

In this case the folder where the large message's file will be stored will be the `HORNETQ_ROOT\data\large-messages` folder, but you can also input the absolute path of the folder.

Considering that large messages need IO consumption for performance issues, the large message directory should be chosen with a different physical volume than the message journal or paging directory.

HornetQ decides if a message is normal or large by looking at the size of the message. In case you are using JMS queues, the threshold value for considering a message as large is configured in the `HornetQ-jms.xml` file present in the `HORNETQ_ROOT\config` folder. The tag is as follows:

```
<min-large-message-size>value</min-large-message-size>
```

The previous tag is present inside the `connection-factory` tag. So if you are using the `NettyConnectionFactory` transport, and you want to define that a message is large if its size is more than 1 MB, you should have the following configuration:

```
   <connection-factory name="NettyConnectionFactory">
      <xa>false</xa>
     <min-large-message-size>1048576</min-large-message-size>
      <connectors>
         <connector-ref connector-name="netty"/>
      </connectors>
      <entries>
         <entry name="/ConnectionFactory"/>
      </entries>
```

The value for the size is expressed in bytes. If no tag/value is specified, HornetQ uses 100 KB as the limit threshold value.

We can now move on to the coding phase. Using NetBeans, follow these steps:

1. From the menu go to **File** | **New Project** | **Java Application**.
2. Name the new project chapter05 and check the checkbox **Create Main Class** and name it chapter05.JMSLargeMessageExample.
3. Also check the flag **Set as Project Main** and click on **Finish**.
4. Once done right-click on the **Libraries** icon of chapter05 project and select **Add jar/folder**.
5. Go to the HORNETQ_ROOT\lib folder and select the hornetq-jms.jar, hornetq-jms-client.jar, jboss-jms-api.jar, and the netty.jar files.
6. Click on **Finish**.

We can now move to the configuration of the large message. To do this we need to follow these steps:

1. In the hornetq-jms.xml file, inside the HORNETQ_ROOT\config\standalone\non-clustered folder, create a queue named queue/exampleQueue.
2. In the same folder, open the hornetq-jms.xml file and set the tag <min-large-message-size> to the value of 10240 bytes in the nettyconnectionfactory tag.
3. Save every change to the configuration files and start the HornetQ server.

Now moving to the code, we connect to the exampleQueue queue and create a producer using the following code:

```
Properties p = new java.util.Properties();
p.put(javax.naming.Context.INITIAL_CONTEXT_FACTORY,"org.jnp.
interfaces.NamingContextFactory");
p.put(javax.naming.Context.URL_PKG_PREFIXES,"org.jboss.naming:org.jnp.
interfaces");
p.put(javax.naming.Context.PROVIDER_URL, "jnp://localhost:1099");

ic = new javax.naming.InitialContext(p);
cf = (javax.jms.ConnectionFactory)ic.lookup("/ConnectionFactory");
queue = (javax.jms.Queue)ic.lookup("queue/exampleQueue");
connection = cf.createConnection();
session = connection.createSession(false, javax.jms.Session.AUTO_
ACKNOWLEDGE);
Message producer = session.createProducer(queue);
```

Now we can move to the interesting part. Now that we have configured the queue to receive large messages, we will read an XML file from the disk and send it as a message to the queue. The following is the code to do this:

```
BytesMessage message = session.createBytesMessage();
FileInputStream fileInputStream = new FileInputStream("/tmp/hl7.xml");
BufferedInputStream bufferedInput = new BufferedInputStream(fileInput
Stream);
message.setObjectProperty("JMS_HQ_InputStream", bufferedInput);
System.out.println("Sending the message ." + new Date());
producer.send(message);
System.out.println("Large Message sent on " + new Date());
```

The first thing to do is to create a `ByteMessage` object that is a class derived from the `HornetqMessage` class using this code:

```
BytesMessage message = session.createBytesMessage();
```

While in our previous JMS examples we used the `TextMessage` object to store the messages that is another extension of the `HornetqMessage`.

Then we read the HL7 XML file using a `FileInputStream` class that will be passed to a `BufferedInputStream` class. As the `BufferedInputStream` object will be the final object to be stored into the message, we could have also used a URL as the source.

The code is easily understandable by Java coders; just remember to put the file in the correct folder:

```
FileInputStream fileInputStream = new FileInputStream(fileInput);
BufferedInputStream bufferedInput = new BufferedInputStream(fileInput
Stream);
```

This is a standard way to read a filestream where the `fileinput` string variable is the absolute path to the file.

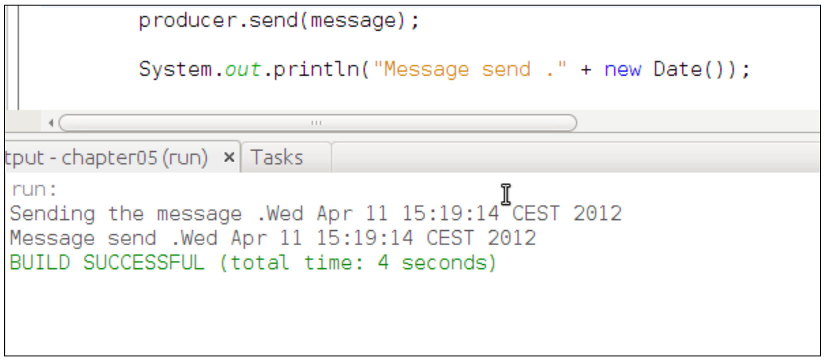
Next we need to set the `JMS_HQ_InputStream` message property:

```
message.setObjectProperty("JMS_HQ_InputStream", bufferedInput);
```

Using this code we tell the message which kind of stream is stored. Then we simply send the message:

```
System.out.println("Sending the message ." + new Date());
producer.send(message);
System.out.println("Large Message sent on " + new Date());
```

If you now run the class, you will see a screenshot similar to the following one on the NetBeans output console:

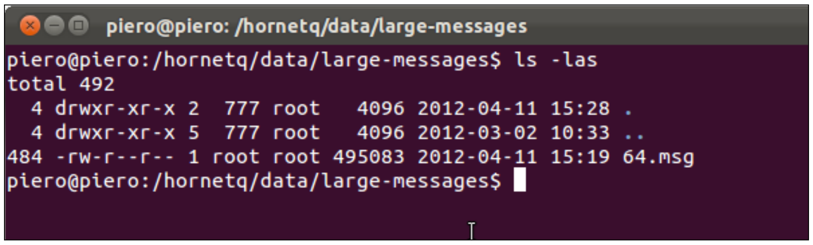


```
        producer.send(message);

        System.out.println("Message send ." + new Date());
```

```
run:
Sending the message .Wed Apr 11 15:19:14 CEST 2012
Message send .Wed Apr 11 15:19:14 CEST 2012
BUILD SUCCESSFUL (total time: 4 seconds)
```

Now to verify that the message has been sent to the large message queue and that it is stored as a physical file, you could perform the following actions:

1. Stop the HornetQ server.
2. Browse to the `HORNETQ_ROOT\data\large-messages` (in our case `/hornetq/data/large-messages/`) and see that there is a file with the `.msg` extension like in the following screenshot:



```
piero@piero:/hornetq/data/large-messages$ ls -las
total 492
  4 drwxr-xr-x 2  777 root   4096 2012-04-11 15:28 .
  4 drwxr-xr-x 5  777 root   4096 2012-03-02 10:33 ..
484 -rw-r--r-- 1 root root 495083 2012-04-11 15:19 64.msg
piero@piero:/hornetq/data/large-messages$
```

If you open it you should see the same content as previously shown.

Now we could reuse the `JMSLargeMessageExample` class to code the consumer. Parsing and storing the HL7 XML file is out of our purpose so we will only save the message to another folder.

To do this we first need to create the consumer with the following code:

```
BytesMessage messageReceived = (BytesMessage)messageConsumer.
receive(120000);
```

In this case as the large messages survive the server restart, we consume it in an asynchronous way.

Now that we have our `BytesMessage` object we can extract its content.

First we create a `BufferedOutputStream` class that will be in charge of storing the content received from the `messageReceived` object.

```
File outputFile = new File("/tmp/reveic");
FileOutputStream fileOutputStream = new FileOutputStream(outputFile);
BufferedOutputStream bufferedOutput = new BufferedOutputStream(fileOu
tputStream);
```

Now as we did for the producer message we will use the following code to assign the content of the message to `bufferedOutput`:

```
messageReceived.setObjectProperty("JMS_HQ_SaveStream",
bufferedOutput);
```

Then we simply close the resources.

Managing large messages using the core API

Handling large messages using the core API is similar to the JMS case but the main difference in this case is that, as the core API does not refer directly to a queue as JMS does, the maximum size of a message to be considered as large is done at the code level.

To understand the case we will create a class that streams the same HL7 file as we did in the previous section. Using NetBeans (or Eclipse) you could reuse the same package and add a new class called `CoreApiLargeMessageExample`. To do this in NetBeans you should perform the following steps:

1. Right-click on the project icon and from the pop up menu go to **New | Java class**.
2. Call the class as `CoreApiLargeMessageExample` and then check the main method box.
3. Create the class inside the `chapter05` package.

Now that our class has been created, we move on by inserting the code after creating a `ClientSessionFactory` object. To do this, open the `CoreApiLargeMessageExample` file and in the `main` method, you can add the following lines of code:

```
HashMap map = new HashMap();
map.put("host", "localhost");
map.put("port", 5445);
ClientSessionFactory factory = null;
```

```
ClientSession session = null;
ServerLocator serverLocator = null;
TransportConfiguration configuration = new TransportConfiguration(Nett
yConnectorFactory.class.getName(), map);
serverLocator = HornetQClient.createServerLocatorWithoutHA(configurat
ion);
```

As we did in *Chapter 3, Basic Coding with HornetQ: Creating and Consuming Messages*, we connect to the HornetQ server using the core API. At the end we create a new `serverlocator` object. Now that we have a `serverLocator` object, you can set on this the size for a message to be considered as large by using the following line of code:

```
serverLocator.setMinLargeMessageSize(25 * 1024);
```

From now on the `ClientSessionFactory` object that will be created will have the large message property on every message sent as 25 KB.

The code is now the counterpart of the core API code with respect to the JMS by using a `BufferedInputStream` class as shown:

```
factory = serverLocator.createSessionFactory();
session = factory.createSession();
ClientProducer producer = session.createProducer("ECGQueue");
ClientMessage message = session.createMessage(true);
String fileInput = "c:\\tmp\\qrs.xml";
FileInputStream fileInputStream = new FileInputStream(fileInput);
BufferedInputStream bufferedInput = new BufferedInputStream(fileInput
Stream);
message.setBodyInputStream(bufferedInput);
producer.send(message);
```

You should see that we have performed the same action as we did in the previous case. The `setBodyInoutStream` method of the `ClientMessage` object always accepts a `BufferedIputStream` object. So we need to read if from the disk.

In this case we point out to this table to demonstrate the symmetricity of the core API large message client with respect to the JMS:

Name	Description	JMS equivalent property
`setBodyInputStream (InputStream)`	Set the `InputStream` used to read a message body when sending it.	`JMS_HQ_InputStream`
`setOutputStream (OutputStream)`	Set the `OutputStream` that will receive the body of a message. This method does not block.	`JMS_HQ_OutputStream`

Name	Description	JMS equivalent property
`saveOutputStream (OutputStream)`	Save the body of the message to the `OutputStream`. It will block until the entire content is transferred to the `OutputStream`.	`MS_HQ_SaveStream`

We also challenge the reader to implement the consumer remembering that we have also, in this case as the JMS case, the possibility of reading the entire message until the stream has finished, or not to wait until the end of the stream.

Managing undelivered messages

Up until now we have supposed that every message has been successfully consumed by the consumer layer. However in a real high-frequency transactional environment this is not always possible. Remember from *Chapter 1*, *Getting Started with HornetQ*, we could have a connection error due to the fact that the MongoDB server is down.

So there are two possibilities when consuming a message, if it is consumed in the correct way, you can commit the session so HornetQ will remove the message from the queue where it was stored. If something goes wrong you can roll back the session in this case the message is put on the queue some other time where it was stored.

This means that if the error persists it is possible that the message is redelivered again and again. To avoid this, HornetQ offers two main possibilities:

- **Delayed redelivery**: You can configure some delay time to let the client return in a successfully delivered state
- **Dead letter address**: All messages that are not delivered correctly could be stored in a particular queue, so they can be parsed in a second time

Delay in redelivery

Configuring a delayed redelivery is pretty easy because it is only a configuration task and does not involve client interaction.

If you remember on the `hornetq-configuration.xml` file we have a special tag that is the `<address-settings>` tag. If you take a look at the default file present in the installation folder you should see the following tag:

```
<address-settings>
   <!--default for catch all-->
   <address-setting match="#">
```

```
            <dead-letter-address>jms.queue.DLQ</dead-letter-address>
            <expiry-address>jms.queue.ExpiryQueue</expiry-address>
            <redelivery-delay>0</redelivery-delay>
            <max-size-bytes>10485760</max-size-bytes>
            <message-counter-history-day-limit>10</message-counter-
history-day-limit>
            <address-full-policy>BLOCK</address-full-policy>
         </address-setting>
      </address-settings>
```

As you can see from the highlighted code, every message independent of the queue where it is stored (`match="#"`) is redelivered without waiting, using the following tag:

```
<redelivery-delay>0</redelivery-delay>
```

The match property is a wildcard. These wildcards are used to prefilter some types of messages on the deployed queues. In this case we tell HornetQ not to filter the messages. In some more advanced configurations that we will see in *Chapter 6, Clustering with HornetQ*, it is possible to use some kind of query syntax to deliver messages to different queues.

This is a very simple case but in a more realistic example, instead of using a non-waiting redelivery, we will create a JMS (so also a core API) queue. On this queue we will define that every message not delivered successfully will wait one minute before trying to be redelivered. So we will give it a second chance of being redelivered after a minute. This case is closer to the real software system. Considering our example with MongoDB, if the database is down for example, it is possible that in a minute it will become alive again.

For our example we will show how to delay a re-delayed message with a delay time of five seconds.

To do this, first open the `hornetq-configuration.xml` file with your text editor located, in our case, inside the `HORNETQ_ROOT\config\stand-alone\non-clustered\` folder and replace the `<address-settings>` tag with the following one:

```
<address-settings>
   <address-setting match="jms.queue.ECGQueue">
          <redelivery-delay>5000</redelivery-delay>
   </address-setting>
</address-settings>
```

Now go to the `hornetq-jms.xml` configuration file located in the same folder as shown previously and create the settings for the `jms.queue.ECGQueue`, by adding the following tag inside the `<configuration>` tag:

```
<queue name="ECGQueue">
   <entry name="/queue/ECGQueue"/>
</queue>
```

You can leave the `<address-setting match="#">` tag inside the `<address settings>` tag because HornetQ uses the most filtering options so you can define every queue created to have no wait for the redelivery, and you can also define a particular queue to have a delay.

So we set up a delay time of five seconds to the `ECGQueue`. Now moving to the code we first need to connect to the queue and correctly instantiate a `javax.jms.Connection` object. As we saw in *Chapter 4, Monitoring HornetQ*, we could do this by using the following lines of code:

```
Properties p = new Properties();
p.put(Context.INITIAL_CONTEXT_FACTORY,"org.jnp.interfaces.
NamingContextFactory");
p.put(Context.URL_PKG_PREFIXES,"org.jboss.naming:org.jnp.interfaces");
p.put(Context.PROVIDER_URL, "jnp://localhost:1099");
initialContext  = new javax.naming.InitialContext(p);
Queue queue = (Queue)initialContext.lookup("/queue/ECGQueue");
ConnectionFactory cf = (ConnectionFactory)initialContext.lookup("/
ConnectionFactory");
connection = cf.createConnection();
```

Now we are ready to create a transactional session, so we can roll back the message which has not been successfully consumed. The code is pretty simple:

```
Session session = connection.createSession(true, 0);
```

The first Boolean parameter tells that the session is transacted with no acknowledgement. Now we can produce a simple message and send it to the `ECGQueue`.

```
MessageProducer producer = session.createProducer(queue);
TextMessage message = session.createTextMessage(body);
producer.send(message);
```

In this code, the `body` string variable is the usual ECG string coming from our device. In this case as the session is transacted we need to explicitly commit the sending of the message otherwise nothing will happen, while with the non-transacted session the `send` method directly commits the message to the queue.

We now need to consume the message and create a managed runtime exception so we could redeliver the message itself. To do this we will try the easy way first by converting the ECG string to a number, so as to generate a `NumberFormatException`. First we will create the consumer:

```
MessageConsumer messageConsumer = session.createConsumer(queue);
connection.start();
```

Now we receive the message but we generate a runtime exception, which we will manage in the `catch` block:

```
try {
    messageReceived = (TextMessage)messageConsumer.receive(5000);
    System.out.println("1st delivery from " + queue.getQueueName() +
": " + messageReceived.getText());
    int value = Integer.parseInt(messageReceived.getText());
} catch (Exception ex) {

    session.rollback();

    messageReceived = (TextMessage)messageConsumer.receive(1000);

    System.out.println("Redelivery has been delayed so received
message is " + messageReceived);

    Thread.sleep(8000);

    messageReceived = (TextMessage)messageConsumer.receive(1000);

    System.out.println("2nd delivery from " + queue.getQueueName() +
": " + messageReceived.getText());

}
```

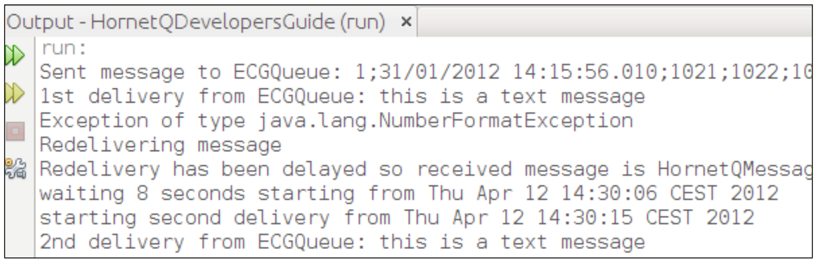
As you can see the consumption of the message is done the same way as we did in *Chapter 1, Getting Started with HornetQ*. In this case, the second line of code after the `try` line triggers a runtime exception, because you are trying to transform a string into a number. At this point the bytecode fires an exception and the code in the block is executed.

First we need to roll back the session, this forces the message to be redelivered into the queue where it was first consumed (the `ECGQueue`).

Next we try to receive the message again, but in this case as the redelivery is delayed in five seconds trying to get the message again in one second (1000 in the `receive` method, will return a null message). So we wait for eight seconds by suspending the execution of the code and then we once again try to receive the message that is now on the queue as five delayed seconds have passed.

The timeline is the following one:

1. The message is delivered correctly the first time so it is removed from its original queue.
2. The exception is raised.
3. The rollback on the session re-puts the message in the original queue.
4. A new delivery is done.
5. The output should be the one displayed in the following screenshot:



```
run:
Sent message to ECGQueue: 1;31/01/2012 14:15:56.010;1021;1022;10
1st delivery from ECGQueue: this is a text message
Exception of type java.lang.NumberFormatException
Redelivering message
Redelivery has been delayed so received message is HornetQMessag
waiting 8 seconds starting from Thu Apr 12 14:30:06 CEST 2012
starting second delivery from Thu Apr 12 14:30:15 CEST 2012
2nd delivery from ECGQueue: this is a text message
```

Dead letter address

Even if the delay in redelivery could be useful, dealing with backend problems, like in our case MongoDB going down, is not a good technique in high-frequency messaging environments because the message is redelivered again and again, potentially increasing the usage of the queue.

A more robust approach is the one of dead letters. The basic idea of the dead letter approach is to redelay the message only a fixed number of times and in case the consumer layer is not able to deliver the message, it is sent to a JMS queue where it is stored.

Fixing a maximum number of deliveries allows the undelivered messages to be re-received not for a long time. The JMS dead letter queue can be managed and read using the HornetQ core API allowing the coder to deeply control every step of the delivery allowing offline analysis.

Configuring our test environment is nothing more than configuring the dead letter JMS queue. Open the `hornetq-configuration.xml` file and add the following lines of code inside the `<address-setting>` tag:

```
<address-setting match="jms.queue.ECGQueue">
         <dead-letter-address>jms.queue.ECGDeadLetterQueue</dead-
letter-address>
         <max-delivery-attempts>3</max-delivery-attempts>
      </address-setting>
```

So we define that for `ECGQueue` the dead letter address queue is the JMS queue named `ECGDeadLetterQueue` and the maximum redelivery number before putting the undelivered message into the queue is three.

Save and open the `hornetq-jms.xml` file in the same folder (`HORNETQ_ROOT\config\standalone\non-clustered`) and create the `ECGDeadLetterQueue` by adding the following tag before the closing tag of the file:

```
<queue name=" ECGDeadLetterQueue ">
   <entry name="/queue/ ECGDeadLetterQueue "/>
</queue>
```

This queue has no particular configuration and we will not use this queue in the code. However, in a production environment a backend service that controls this queue will be a good solution. We challenge you to code, as an exercise, a monitor client on the dead letter queue, so that when a new message is inserted an alert is raised.

Now fire up NetBeans (or Eclipse) and add a new `DeadLetterExample` Java class with a `main` method. After this you could reuse the same code as before to create a message that is successfully sent to the `ECGQueue`.

Now let us roll back the stored message three times. You could do this using the following code:

```
TextMessage messageReceived = (TextMessage)messageConsumer.
receive(5000);
System.out.println("1st delivery from " + queue.getQueueName() + ": "
+ messageReceived.getText());
session.rollback();
messageReceived = (TextMessage)messageConsumer.receive(5000);
System.out.println("2nd delivery from " + queue.getQueueName() + ": "
+ messageReceived.getText());
session.rollback();
```

```
messageReceived = (TextMessage)messageConsumer.receive(5000);
System.out.println("3rd delivery from " + queue.getQueueName() + ": "
+ messageReceived.getText());
session.rollback();
```

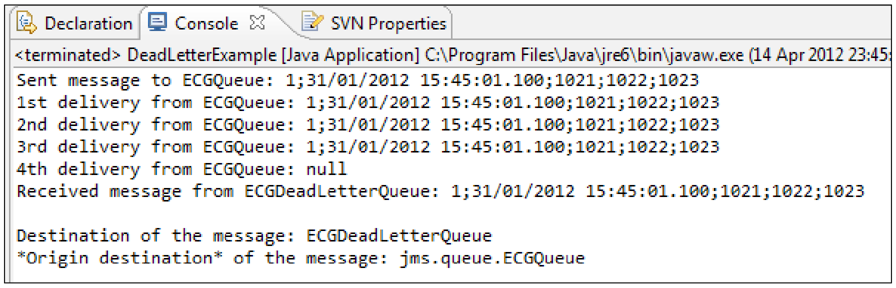
You could do the same thing by calling the `session.rollback()` code three times.

Now we move to the dead letter queue to receive the message that was refused by the ECG queue:

```
Queue deadLetterQueue = (Queue)initialContext.lookup("/queue/
ECGDeadLetterQueue");

MessageConsumer deadLetterConsumer = session.createConsumer(deadLette
rQueue);

messageReceived = (TextMessage)deadLetterConsumer.receive(5000);

System.out.println("Received message from " + deadLetterQueue.
getQueueName() + ": " + messageReceived.getText());

System.out.println("Destination of the message: " + ((Queue)
messageReceived.getJMSDestination()).getQueueName());

System.out.println("*Origin destination* of the message: " +
messageReceived.getStringProperty("_HQ_ORIG_ADDRESS"));

session.commit();
```

As you can see we reused the same session but in this case we connect to the `ECGDeadLetterQueue` queue and receive our message. The content of the message is the same as the original message but the headers change and in this case we read the property telling what the original queue was where the message should have been consumed. Remember that a dead letter queue can be associated with different queues.

We suggest the curious reader to refer to the book's code and by compiling it you should obtain the following console output:


Declaration | Console | SVN Properties

```
<terminated> DeadLetterExample [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (14 Apr 2012 23:45:
Sent message to ECGQueue: 1;31/01/2012 15:45:01.100;1021;1022;1023
1st delivery from ECGQueue: 1;31/01/2012 15:45:01.100;1021;1022;1023
2nd delivery from ECGQueue: 1;31/01/2012 15:45:01.100;1021;1022;1023
3rd delivery from ECGQueue: 1;31/01/2012 15:45:01.100;1021;1022;1023
4th delivery from ECGQueue: null
Received message from ECGDeadLetterQueue: 1;31/01/2012 15:45:01.100;1021;1022;1023

Destination of the message: ECGDeadLetterQueue
*Origin destination* of the message: jms.queue.ECGQueue
```

Message priority

In the last section of this chapter we will focus our attention on another important task of JMS messages.

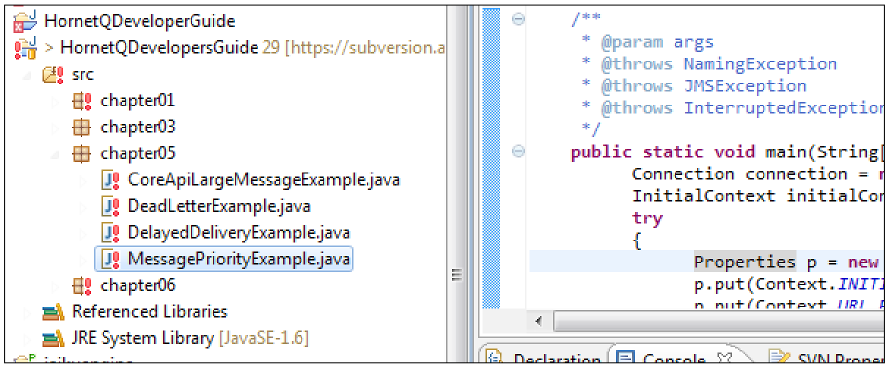
In our fictional case, imagine that in the normal streaming of the patient's ECG device we find that the heart rate goes above 180 (heartbeats per minute). We remember that having a heart rate of 180 is potentially dangerous for the cardiac muscles, so we have a patient in a potential life risk situation.

In this case if we share the same queue that stores all the messages coming from different patients, the messages with a heart rate above 180 should be consumed prior to the other messages.

We are in fact dealing with the arrival order on a JMS queue inside HornetQ. Messages of equal priorities are delivered in the natural order of their arrival at their queue. This is basically done by considering the timestamp stored in the headers of the messages.

In this section we will investigate the possibility of assigning different priorities to the JMS messages in a queue. From the server side we do not need any particular configuration, but only need to have a JMS queue ready to store messages. Considering our example we will code a producer that assigns different priorities on some messages.

Now moving to the coding side, open NetBeans or Eclipse and create a new Java class with a `main` method. Do not forget to assign to it the right package side (see the following screenshot):


HornetQDeveloperGuide
> HornetQDevelopersGuide 29 [https://subversion.a
src
chapter01
chapter03
chapter05
CoreApiLargeMessageExample.java
DeadLetterExample.java
DelayedDeliveryExample.java
MessagePriorityExample.java
chapter06
Referenced Libraries
JRE System Library [JavaSE-1.6]
Declaration
Console
SVN Proper

```
/**
 * @param args
 * @throws NamingException
 * @throws JMSException
 * @throws InterruptedException
 */
public static void main(String[
        Connection connection = n
        InitialContext initialCo
        try
        {
                Properties p = new
                p.put(Context.INITI
```

Now connect to the queue `ECGQueue` using the `netty` connector, and create a producer.

In this case we will create a non-transacted session, because we do not need to commit or roll back the transactions.

The code to do this is pretty similar to the one we used in the previous sections.

```
Queue queue = (Queue)initialContext.lookup("/queue/ECGQueue");

ConnectionFactory cf = (ConnectionFactory)initialContext.lookup("/
ConnectionFactory");

connection = cf.createConnection();

Session session = connection.createSession(false, Session.AUTO_
ACKNOWLEDGE);

MessageProducer producer = session.createProducer(queue);
```

Now we push three different messages with three different priorities. Before doing this we output the default priority of every single message, to show the reader that every message is created with default priority.

```
TextMessage[] messagewithPriorities = new TextMessage[3];
messagewithPriorities[0] =  session.createTextMessage("1;31/01/2012
19:35:34.000;180");
messagewithPriorities[1] = session.createTextMessage("1;31/01/2012
19:35:35.000;200");
messagewithPriorities[1] = session.createTextMessage("1;31/01/2012
19:35:36.000;220");

for (int i = 0; i<3;i++){
   System.out.println("message " + (i+1) + " created with priority "
+ messagewithPriorities[i].getJMSPriority());
}
```

Now we can assign a priority to every message prior to inserting it into the `ECGQueue`. We point out that the priority of a message is a number that spans from 0 (the lowest) to 9 (the highest). In our example, following what we have described about heart rate, we will assign the higher priority to the higher value of the heart rate.

```
producer.send(sentMessages[0]);
 System.out.println("Message sent: " + sentMessages[0].getText() +
                   " with priority: " +
                   sentMessages[0].getJMSPriority());
 producer.send(sentMessages[1], DeliveryMode.NON_PERSISTENT, 5, 0);
   System.out.println("Message sent: " + sentMessages[1].getText() +
                   " with priority: " +
                   sentMessages[1].getJMSPriority());
 producer.send(sentMessages[2], DeliveryMode.NON_PERSISTENT, 9, 0);
 System.out.println("Message sent: " + sentMessages[2].getText() +
                   " with priority: " +
                   sentMessages[2].getJMSPriority());
```

As the reader can see we send the first message as usual; the other two are sent by defining three other parameters respectively.

The delivery mode is the JMS specification and can be:

- Non-persistent
- Persistent

We will not go into the details; we invite you to refer to the JMS Javadoc, but persistence is the way HornetQ internally stores the message. This means that if the message is marked as persistent it means that it will be saved to the storage log.

The second parameter is the priority of the message. And the third one is the message's lifetime in milliseconds. In this case messages are marked with no lifetime.

We can now move to the consumer layer to see what happens when the messages are received.

```
MessageConsumer Consumer = session.createConsumer(queue);
Consumer.setMessageListener(new SimpleMessageListener());
connection.start();
Thread.sleep(5000);
```

In this case we have also coded a `MessageListener` interface implementing the `javax.jms.MessageListener` event to match the fact that we fire up an event every time a new message is received no matter when. The code for the `MessageListener` interface is pretty easy as described in the following lines of code:

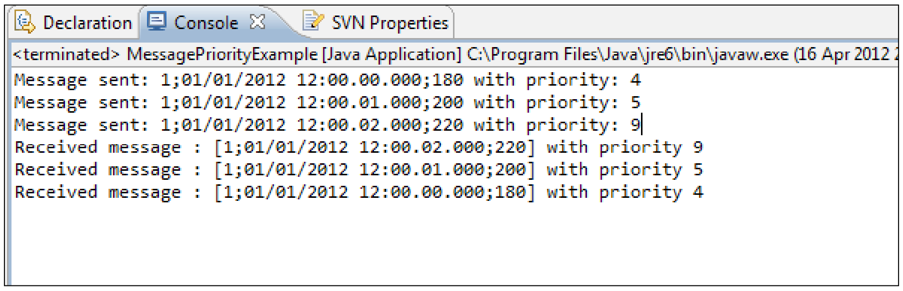
```
   public class SimpleMessageListener implements MessageListener
   {

public class SimpleMessageListener implements MessageListener
{

  public SimpleMessageListener()
  {
  }

  public void onMessage(final Message msg)
  {
    TextMessage textMessage = (TextMessage)msg;
    try
    {
          System.out.println("Received message : [" + textMessage.
getText() + "] with priority " + textMessage.getJMSPriority());
    }
```

```
            catch (JMSException e)
            {
                   result = false;
            }

         }

      }
```

The `OnMessage` event is fired up every time a new message is received and creates a new JMS message object that, in this case, we use only to display the priority and the text showing that the priority on the delivery has been changed.

We add a `Thread.sleep(5000)` method only to prevent the fact that the programs end before receiving all the messages.

If you run the code you should see an output similar to the following screenshot:



```
<terminated> MessagePriorityExample [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (16 Apr 2012
Message sent: 1;01/01/2012 12:00.00.000;180 with priority: 4
Message sent: 1;01/01/2012 12:00.01.000;200 with priority: 5
Message sent: 1;01/01/2012 12:00.02.000;220 with priority: 9
Received message : [1;01/01/2012 12:00.02.000;220] with priority 9
Received message : [1;01/01/2012 12:00.01.000;200] with priority 5
Received message : [1;01/01/2012 12:00.00.000;180] with priority 4
```

As you can see, the priority has been changed according to the order we set up.

Even if you are using the core API it is possible for a message to be set with a priority. In this case the coder can set the priority by simply using the methods `setPriority` or `getPriority` on a message.

In this case, once you have a message producer you can set the priority from 0 to 9 with the following code:

```
ClientProducer producer = session.createProducer("ECGQueue");
ClientMessage message = session.createMessage(true);
message. setPriority(0);
producer.send(message)
```

Again we invite the reader to recode the example shown in this section for the JMS priority to see whether the translation into the core API dialect is possible.

Summary

As a final point to this chapter we have also seen that it is possible to implement some events on the messages sent/received so as to have a better code-oriented management of the messages.

We have also learned how to manage large messages and how to configure a message to be intended as large, manage redelivery, and manage priorities on messages.

We invite the readers to take a look at the HornetQ Javadoc (`http://docs.jboss.org/hornetq/2.2.5.Final/api/index.html`) to understand this feature better or try to test their knowledge by referring to the book's code.

In the last three chapters we have seen how to deal with various HornetQ features, but we limited our code to only one standalone server that is running on a single machine.

But HornetQ is designed to be deployed in mission-critical environments. So we will see in *Chapter 6*, *Clustering with HornetQ*, how to set up HornetQ in a cluster environment to have multiple server instances that act like a single entity.

6
Clustering with HornetQ

The current definition of a **computer cluster** from Wikipedia is as follows:

A computer cluster consists of a set of loosely-connected computers that work together so that in many respects they can be viewed as a single system.

This definition does not help in understanding what a cluster is and how to manage a cluster in a software system.

Considering the HornetQ case, clustering in HornetQ in its simpler meaning is the ability of HornetQ to have different instances acting like a single entity to an enquiry, both from the consumer or producer side.

The HornetQ cluster derives itself directly from JBoss' cluster feature. In this chapter we will focus on the different possibilities that we have of configuring HornetQ to act in a clustered way. We won't be able to cover any single configuration from a matter of length but we hope we can provide the basic concepts so that the reader can use this feature of HornetQ.

For now, we need to define some concepts to allow the reader to better understand the configuration that we will illustrate. We consider the following definitions:

- **Cluster**: This is a group of HornetQ servers that are able to share the messaging process
- **Node**: This is an active HornetQ server that manages its own messages and handles its own connections
- **Host**: This is an active OS (both Linux or Windows) that can be a physical machine or a virtual machine

In our case we have different possibilities on grouping HornetQ servers according to the different needs that are in the minds of the coder/administrator.

In this chapter we will take a look at the following configurations:

- Basic configuration for two or more HornetQ instances
- Different nodes that share the same queue

Considering our test environment as a single Windows or Ubuntu machine, we will create every node inside the same machine. But in the real world, two or more machines with a single node running inside them will be configured. The negative part of using this configuration, as we will see, is the need for configuring different ports to avoid a runtime error during startup.

In every case, we will also give the reader a suggestion that will use the example as a starting point for implementation on more machines.

In our examples we will use JMS to illustrate the cases, the core API implementation is left to the willing reader.

First cluster with two different nodes

Our first example is one of the simplest, yet one of the most needed ones when we need to create a cluster. Till now we have seen a single queue deployed on a single node. So we basically interacted with only one instance of HornetQ, both from the consumer, rather than from the consumer and producer layer.

In this case we change our scenario. We create two nodes for running different HornetQ server instances. Then we will use JMS to interact both at a producer level and at a consumer level to demonstrate that the cluster will share information between the two nodes.

JMS topics are part of the JMS specification and are basically a distribution mechanism for publishing messages that are delivered to multiple subscribers. From our point of view, you could think of a topic as a distributed queue where we can produce consumer messages.

From the code point of view, this will not affect anything except for the fact that we will use a topic instead of a queue to create the sessions.

Before going through the code we need to describe how HornetQ should be configured to act like a cluster. The main part resides in the `hornetq-configuration.xml` file where there is a group of XML tags that define how to group and cluster different HornetQ servers.

The mechanism used to discover a node in a cluster group is transparent to the reader, but only to point out that we need to define the following:

- A cluster group that is a logical name that maps every request from the client
- A cluster security setting stating what different nodes are allowed to do in queue connection and sharing information
- A cluster discovery, which is the mechanism that tells the cluster how to include nodes

In our example we will start with two nodes of HornetQ acting like a cluster. From the management point of view, our example is not so meaningful because we will create everything in one machine, so the user could not be aware of every configuration problem that one could have by running two nodes in two different machines.

The steps for configuring our cluster are as follows:

1. Create two `HORNETQ_ROOT` folders.
2. Configure both the HornetQ instances to act like a cluster and to share a queue.
3. Start every single node.
4. Code an example that will produce a message on the shared queue and will be consumed in both the nodes.

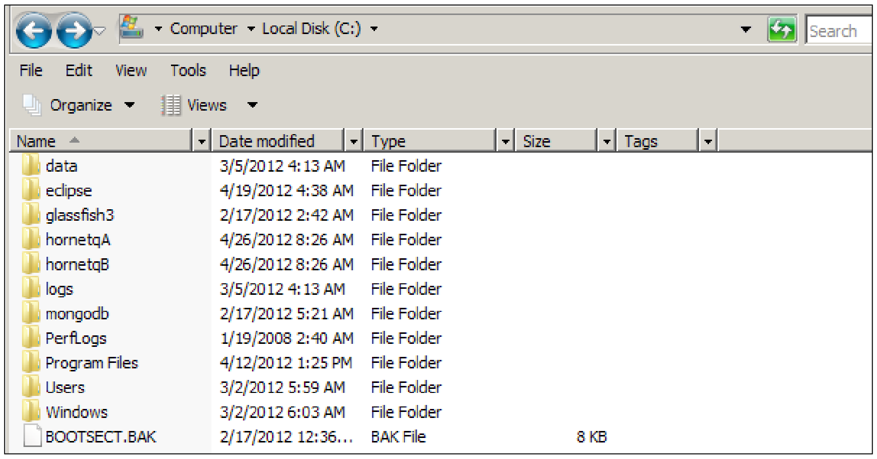
So let us start with it.

Creating two nodes

This step is simple and we will detail it only for the first-time user. To avoid any previous configuration issues remove the `HORNETQ_ROOT` folder and download a new `zip/tar.gz` archive of your HornetQ server from `http://www.jboss.org/hornetq/downloads`.

1. After this, uncompress your archive to have the normal `HORNETQ_ROOT` folder as done in *Chapter 1, Getting Started with HornetQ*.
2. Now rename the `HORNETQ_ROOT` folder to `HORNETQA`.
3. Redo the previous step to create a `HORNETQB` folder.

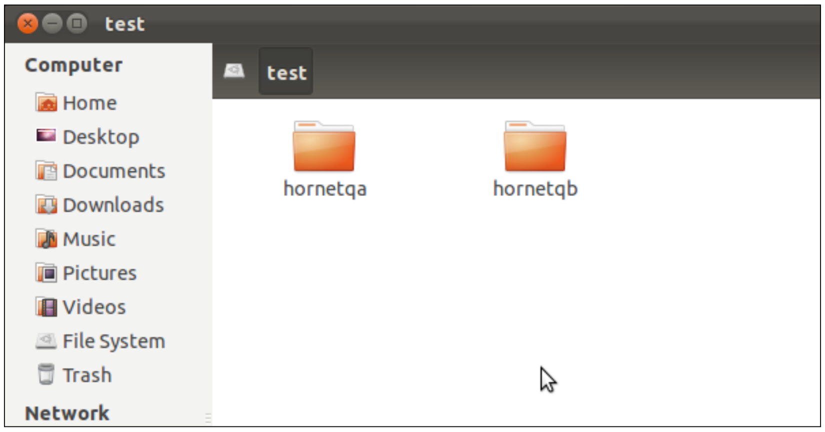
So now you should have two `HORNETQ_ROOT` folders, with the same subfolder structure. From now on we will refer to the `HORNETQA_ROOT` folder and to the `HORNETQB_ROOT` folder respectively as the first and the second decompressed folders, that we will also name node A and node B. The following screenshot is the configuration that you should have on a Windows system:



Name	Date modified	Type	Size	Tags
data	3/5/2012 4:13 AM	File Folder		
eclipse	4/19/2012 4:38 AM	File Folder		
glassfish3	2/17/2012 2:42 AM	File Folder		
hornetqA	4/26/2012 8:26 AM	File Folder		
hornetqB	4/26/2012 8:26 AM	File Folder		
logs	3/5/2012 4:13 AM	File Folder		
mongodb	2/17/2012 5:21 AM	File Folder		
PerfLogs	1/19/2008 2:40 AM	File Folder		
Program Files	4/12/2012 1:25 PM	File Folder		
Users	3/2/2012 5:59 AM	File Folder		
Windows	3/2/2012 6:03 AM	File Folder		
BOOTSECT.BAK	2/17/2012 12:36...	BAK File	8 KB	

In this case we will refer to the `HORNETQA_ROOT` folder with the absolute path `c:\hornetqa` and to the `HORNETQB_ROOT` folder with the absolute path `c:\hornetqb`.

In our Ubuntu system we have the folders shown in the following screenshot:



In this case the `HORNETQA_ROOT` folder has the absolute path `/test/hornetqa` and the `HORNETQB_ROOT` folder has the absolute path `/test/hornetqb`.

Now that you have both the nodes let us go through the configuration steps.

Configuring the nodes

Basically, now we have two HornetQ nodes that can be started. Now we have three more configuration steps as follows:

1. Configuring the `hornetq-configuration.xml` file.
2. Configuring the `run.sh` (`run.bat`) file.
3. Configuring the `hornetq-jms.xml` file to have a topic for producing and consuming messages.

So far we have always used the configuration files stored in the `HORNETQ_ROOT\config\standalone\non-clustered` folder. To create a cluster, the main point is to start every node using a particular configuration file.

The folder structure of `HORNETQ_ROOT` allows both to start a node in standalone mode or in cluster mode. So the first thing to do is to go to every node and tell the node itself to start using the cluster configuration files.

So in our example you first need to go to the `HORNETQA_ROOT\bin` folder and open the `run.sh` file using your preferred text editor if you are in Ubuntu, or the `run.bat` file if you are on a Windows system. In this case we will illustrate the changes needed for the Ubuntu system. The Windows counterpart is pretty much the same thing.

So open the `run.sh` file using your preferred text editor in the `HORNETQA_ROOT\bin` folder. You should see a line of code like the following:

```
if [ a"$1" = a ]; then CONFIG_DIR=$HORNETQ_HOME/config/stand-alone/
non-clustered; else CONFIG_DIR="$1"; fi
```

This line manages the first parameter launched from the terminal when you use the `run.sh` file. In the default case (no parameter passed), the nodes use the XML files inside the `HORNETQA_ROOT\config\standalone\non-clustered` folder.

In our case, we replace the previous line with the following one:

```
if [ a"$1" = a ]; then CONFIG_DIR=$HORNETQ_HOME/config/stand-alone/
clustered; else CONFIG_DIR="$1"; fi
```

So when you launch the `run.sh` command from the terminal, the configuration files that HornetQ looked at are the ones stored in the `HORNETQ_ROOT\config\standalone\clustered` folder.

> Without changing the `run.sh` file it is also possible to do this by using the command line. So for example if you want to launch HornetQ using the clustered `config` folder, you can do this by using the terminal command `./run.sh ../config/stand-alone/clustered` in Ubuntu or the `.\run.bat ..\config\stand-alone\clustered` command in Windows.

Now that you have made changes to the `run.sh` file of the `HORNETQA_ROOT\bin` folder, redo the same changes in the `HORNEQB_ROOT\bin` folder. Now you have two nodes configured to run as a cluster.

We now need to make one more change in the `run.sh` file of the `HORNETQB_ROOT\bin` folder that will be the second node to be started. This change is needed because HornetQ uses the `rmi` server and the `netty` server to allow two nodes to communicate with each other. Every node should also expose a port. The default ports that are bound on the first node when you start it are:

- 5445 for the `netty` server
- 5455 for the `batch netty` server

So for the second node we need to change these ports otherwise we will have a "port already in use exception" when starting the second node. These parameters are configurable using XML or property files, but in our case it is possible to configure them using a system variable.

To do this on the second node go to the `HORNETQA_ROOT\bin` folder and open the `run.sh` file (or `run.bat` on windows) you should see the following line:

```
#export CLUSTER_PROPS="-Djnp.port=1099 -Djnp.rmiPort=1098 -Djnp.
host=localhost -Dhornetq.remoting.netty.host=localhost -Dhornetq.
remoting.netty.port=5445"
```

You should change this line with the following line:

```
export CLUSTER_PROPS="-Djnp.port=2099 -Djnp.rmiPort=2098 -Dhornetq.
remoting.netty.port=6445 -Dhornetq.remoting.netty.batch.port=6455"
```

As you can see we have only added 1000 to the default port that was used by HornetQ in the default configuration. We stressed that this is only a choice; the important thing is to have no other service listening on the default port, and because we are using the same machine. If the reader is using two different machines this part is not necessary.

In Windows, changing the `run.bat` file is easy as shown in the following lines of code. The line to be changed is:

```
REM set CLUSTER_PROPS="-Djnp.port=1099 -Djnp.rmiPort=1098 -Djnp.
host=localhost -Dhornetq.remoting.netty.host=localhost -Dhornetq.
remoting.netty.port=5445"
```

With the following one:

```
set CLUSTER_PROPS="-Djnp.port=2099 -Djnp.rmiPort=2098 -Dhornetq.
remoting.netty.port=6445 -Dhornetq.remoting.netty.batch.port=6455"
```

We changed these ports because otherwise when the second node of our cluster will start, we will have a runtime error as the default ports are already bound by the first node of the cluster.

If you open the `hornetq-configuration.xml` file that is used to create the cluster you should see the following XML tag section:

```
<broadcast-groups>
   <broadcast-group name="bg-group1">
      <group-address>231.7.7.7</group-address>
      <group-port>9876</group-port>
      <broadcast-period>5000</broadcast-period>
      <connector-ref>netty</connector-ref>
   </broadcast-group>
</broadcast-groups>

<discovery-groups>
   <discovery-group name="dg-group1">
      <group-address>231.7.7.7</group-address>
      <group-port>9876</group-port>
      <refresh-timeout>10000</refresh-timeout>
   </discovery-group>
</discovery-groups>

<cluster-connections>
   <cluster-connection name="my-cluster">
      <address>jms</address>
      <connector-ref>netty</connector-ref>
       <discovery-group-ref discovery-group-name="dg-group1"/>
   </cluster-connection>
</cluster-connections>
```

The three tags describe to the HornetQ instance how it is clustered:

- Broadcast-group describes the virtual IP address of the group of nodes and which connector is to be used
- Discovery-group describes the group of instances of the virtual port where the cluster is responding as a whole and the refresh time
- Cluster-connections describe the virtual name of the cluster and the connector to be used

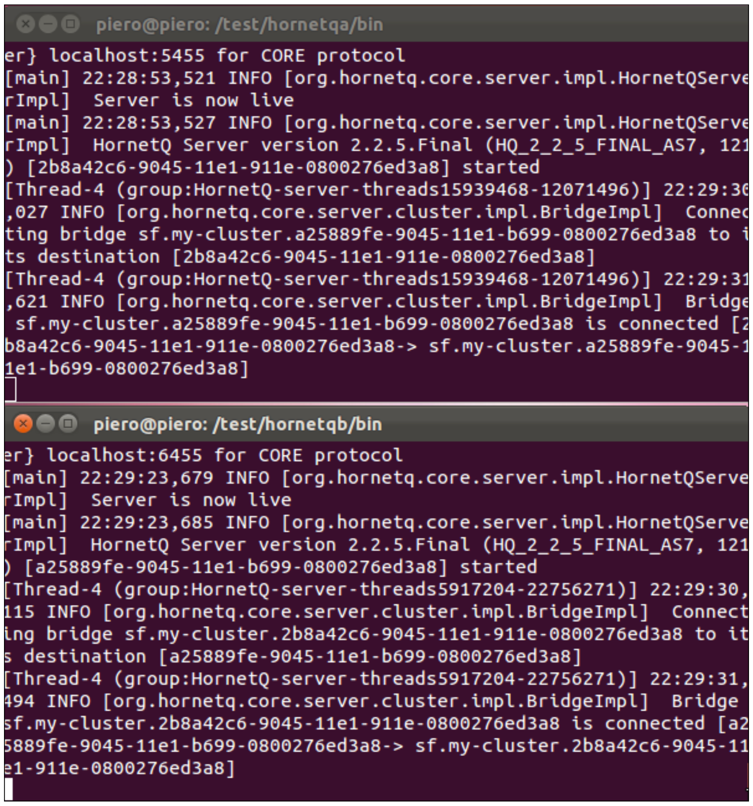
As the last step we add the following tag into the `hornetq-jms.xml` file that defines the topic that will be used to send and receive messages.

```
<topic name="ECGTopic">
   <entry name="/topic/ECGTopic"/>
</topic>
```

This tag should be added to the file defining queues in both nodes of our cluster otherwise we will get a runtime error when trying to consume the messages.

Running both the nodes

Now that everything is configured we can start both the nodes. Using Ubuntu there is nothing more to do than open a terminal on the `HORNETQA_ROOT\bin` folder and type the `./run.sh` command, open another terminal on the `HORNETQ_ROOT\bin` folder and type again the `./run.sh` command. As result you should see the following screenshot:



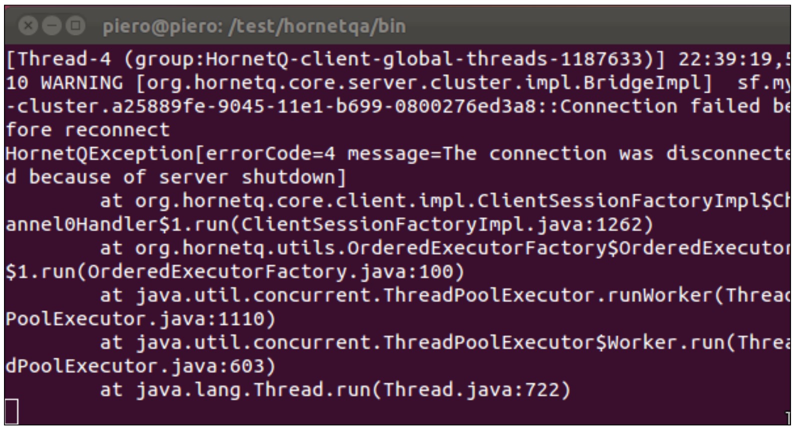
piero@piero: /test/hornetqa/bin

```
er} localhost:5455 for CORE protocol
[main] 22:28:53,521 INFO [org.hornetq.core.server.impl.HornetQServe
rImpl]  Server is now live
[main] 22:28:53,527 INFO [org.hornetq.core.server.impl.HornetQServe
rImpl]  HornetQ Server version 2.2.5.Final (HQ_2_2_5_FINAL_AS7, 121
) [2b8a42c6-9045-11e1-911e-0800276ed3a8] started
[Thread-4 (group:HornetQ-server-threads15939468-12071496)] 22:29:30
,027 INFO [org.hornetq.core.server.cluster.impl.BridgeImpl]  Connec
ting bridge sf.my-cluster.a25889fe-9045-11e1-b699-0800276ed3a8 to i
ts destination [2b8a42c6-9045-11e1-911e-0800276ed3a8]
[Thread-4 (group:HornetQ-server-threads15939468-12071496)] 22:29:31
,621 INFO [org.hornetq.core.server.cluster.impl.BridgeImpl]  Bridge
 sf.my-cluster.a25889fe-9045-11e1-b699-0800276ed3a8 is connected [2
b8a42c6-9045-11e1-911e-0800276ed3a8-> sf.my-cluster.a25889fe-9045-1
1e1-b699-0800276ed3a8]
```

piero@piero: /test/hornetqb/bin

```
er} localhost:6455 for CORE protocol
[main] 22:29:23,679 INFO [org.hornetq.core.server.impl.HornetQServe
rImpl]  Server is now live
[main] 22:29:23,685 INFO [org.hornetq.core.server.impl.HornetQServe
rImpl]  HornetQ Server version 2.2.5.Final (HQ_2_2_5_FINAL_AS7, 121
) [a25889fe-9045-11e1-b699-0800276ed3a8] started
[Thread-4 (group:HornetQ-server-threads5917204-22756271)] 22:29:30,
115 INFO [org.hornetq.core.server.cluster.impl.BridgeImpl]  Connect
ing bridge sf.my-cluster.2b8a42c6-9045-11e1-911e-0800276ed3a8 to it
s destination [a25889fe-9045-11e1-b699-0800276ed3a8]
[Thread-4 (group:HornetQ-server-threads5917204-22756271)] 22:29:31,
494 INFO [org.hornetq.core.server.cluster.impl.BridgeImpl]  Bridge
sf.my-cluster.2b8a42c6-9045-11e1-911e-0800276ed3a8 is connected [a2
5889fe-9045-11e1-b699-0800276ed3a8-> sf.my-cluster.2b8a42c6-9045-11
e1-911e-0800276ed3a8]
```

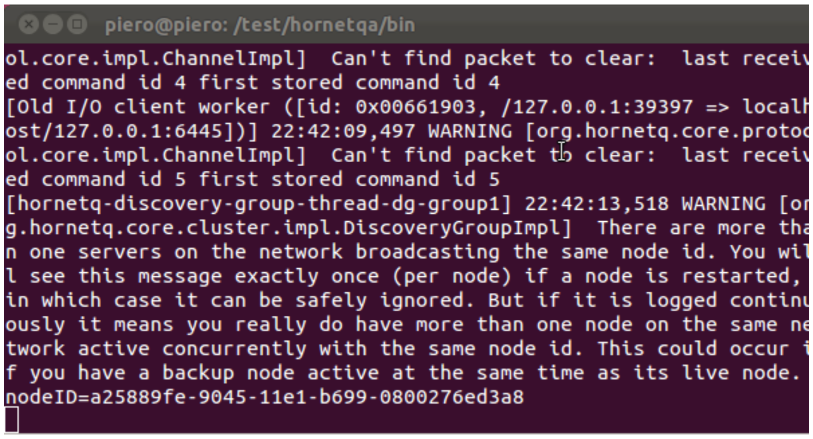
Eureka! As you can see from the last lines of both the terminals, a cluster between the two nodes has been formed correctly. It is interesting to note that if you now stop the second node, the first one (after some delay) receives the notification and displays the following output:



piero@piero: /test/hornetqa/bin

```
[Thread-4 (group:HornetQ-client-global-threads-1187633)] 22:39:19,5
10 WARNING [org.hornetq.core.server.cluster.impl.BridgeImpl]  sf.my
-cluster.a25889fe-9045-11e1-b699-0800276ed3a8::Connection failed be
fore reconnect
HornetQException[errorCode=4 message=The connection was disconnecte
d because of server shutdown]
        at org.hornetq.core.client.impl.ClientSessionFactoryImpl$Ch
annel0Handler$1.run(ClientSessionFactoryImpl.java:1262)
        at org.hornetq.utils.OrderedExecutorFactory$OrderedExecutor
$1.run(OrderedExecutorFactory.java:100)
        at java.util.concurrent.ThreadPoolExecutor.runWorker(Threac
PoolExecutor.java:1110)
        at java.util.concurrent.ThreadPoolExecutor$Worker.run(Threa
dPoolExecutor.java:603)
        at java.lang.Thread.run(Thread.java:722)
```

Finally, if you again run the `run.sh` file of the second node, you should see the first node advising the second node to join the cluster as shown in the following screenshot:



```
ol.core.impl.ChannelImpl]  Can't find packet to clear:  last receiv
ed command id 4 first stored command id 4
[Old I/O client worker ([id: 0x00661903, /127.0.0.1:39397 => localh
ost/127.0.0.1:6445])] 22:42:09,497 WARNING [org.hornetq.core.protoc
ol.core.impl.ChannelImpl]  Can't find packet to clear:  last receiv
ed command id 5 first stored command id 5
[hornetq-discovery-group-thread-dg-group1] 22:42:13,518 WARNING [or
g.hornetq.core.cluster.impl.DiscoveryGroupImpl]  There are more tha
n one servers on the network broadcasting the same node id. You wil
l see this message exactly once (per node) if a node is restarted,
in which case it can be safely ignored. But if it is logged continu
ously it means you really do have more than one node on the same ne
twork active concurrently with the same node id. This could occur i
f you have a backup node active at the same time as its live node.
nodeID=a25889fe-9045-11e1-b699-0800276ed3a8
```

Now we are ready to move to the coding part of our example.

Coding a message producer and consumer on a cluster

After the first long introduction that probably bored you with a coding experience, we are now ready to move to the coding part. In this example, we will create a producer on the cluster that we configured that will push a message, and we will create two consumers, one per node, so that it will be possible to see that the message is consumed on both nodes while it has been produced only on one of the two.

As done in the previous chapters using NetBeans or Eclipse, you should create a class named `TwoNodesExample` with a main method under the `chapter06` package.

In our example we will be:

- Creating a connection with the cluster just formed
- Connecting on this session and creating a producer
- Sending to the topic the ECG message string
- Connecting both nodes and seeing whether both the nodes consume the message

In our example, we will create two connections, one for each node of our cluster.

To do this we reuse the code for the connection as shown in the following lines of code:

```
Connection connectiona = null;
Connection connectionb = null;

InitialContext initialContexta = null;
InitialContext initialContextb = null;

String ECG_TEXT = "1;31/01/2012 15:45:01.100;1021;1022;1023";

Properties p = new Properties();
p.put(Context.INITIAL_CONTEXT_FACTORY,"org.jnp.interfaces.
NamingContextFactory");
p.put(Context.URL_PKG_PREFIXES,"org.jboss.naming:org.jnp.interfaces");
p.put(Context.PROVIDER_URL, "jnp://localhost:1099");
initialContexta  = new javax.naming.InitialContext(p);

ConnectionFactory cfa = (ConnectionFactory)initialContexta.lookup("/
ConnectionFactory");

p = new Properties();
p.put(Context.INITIAL_CONTEXT_FACTORY,"org.jnp.interfaces.
NamingContextFactory");
p.put(Context.URL_PKG_PREFIXES,"org.jboss.naming:org.jnp.interfaces");
p.put(Context.PROVIDER_URL, "jnp://localhost:2099");
initialContextb  = new javax.naming.InitialContext(p);

ConnectionFactory cfb = (ConnectionFactory)initialContexta.lookup("/
ConnectionFactory");
```

Using the same code of the previous chapters we create two `connectionFactory` objects, one for every node.

As the reader can see the only difference with the code for instantiating one single connection factory is the fact that we point to port 2099 for the second connection factory with the second node. The code to do this is the following:

```
p.put(Context.PROVIDER_URL, "jnp://localhost:2099");
```

This is the same port on which we have started the second node.

Now that we have the connection factories, we need to create two `connection` objects and two `session` objects. This is nothing more than duplicating the code we have already used in the first five chapters as the reader can see in the following lines of code:

```
Topic topic = (Topic)initialContexta.lookup("/topic/ECGTopic");
connectiona = cfa.createConnection();
connectionb = cfb.createConnection();

connectiona.start();
connectionb.start();

Session sessiona = connectiona.createSession(false, Session.AUTO_
ACKNOWLEDGE);
Session sessionb = connectionb.createSession(false, Session.AUTO_
ACKNOWLEDGE);
```

Now we have the interesting part where we connect to the same `ECGQUE` topic with the following configuration:

- On node A we have a producer and a consumer
- On node B we have only a consumer

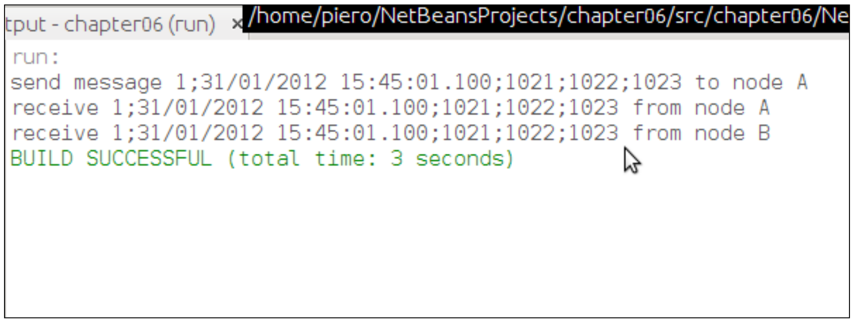
Now we will send the ECG message on node A and then we will consume the message on node A and on node B. Then we will see the magic where only one message is sent but it is received on both nodes. So let us proceed with the producer and consumer on node A:

```
MessageConsumer messageConsumera = sessiona.createConsumer(queue);

MessageProducer producer = sessiona.createProducer(topic);

TextMessage message = sessiona.createTextMessage(ECG_TEXT);

producer.send(message);
```

Now we create only the consumer on node B.

```
MessageConsumer messageConsumerb = sessionb.createConsumer(topic);
```

So we now normally consume the message on node A and node B.



```
run:
send message 1;31/01/2012 15:45:01.100;1021;1022;1023 to node A
receive 1;31/01/2012 15:45:01.100;1021;1022;1023 from node A
receive 1;31/01/2012 15:45:01.100;1021;1022;1023 from node B
BUILD SUCCESSFUL (total time: 3 seconds)
```

Now from the output shown in the previous screenshot you can see that the message has been managed internally by HornetQ as it is on cluster mode. So when the queue on the first node has received the message it has been delivered automatically on node B where it is ready to be consumed by the consumer attached to `ECGTopic`.

It is important to notice that in a real cluster working environment you could think of assigning a machine to a node. In this case, it is not necessary to change any port but only to be sure that the ports are open in the firewall of every machine involved in the cluster.

In this configuration, we have pointed to the nodes using JMS and we use JNDI to instantiate the connections on every node.

This could be a little tricky to manage from a code point of view because as you add a new node to your cluster you need to recode in order to include the new connection into your code.

You could also use the discovery method that will look for a connection using the logical IP address assigned to the cluster itself. So it is possible to connect to a cluster as a whole using the IP address-port pair contained in the broadcast-groups tag of the `hornetq-configuration.xml` file. We will give a snippet of code to illustrate the case:

```
final String groupAddress = "231.7.7.7";

final int groupPort = 9876;

ConnectionFactory jmsConnectionFactory =
        HornetQJMSClient.createConnectionFactory(groupAddress,
groupPort);

Connection jmsConnection1 = jmsConnectionFactory.createConnection();

Connection jmsConnection2 = jmsConnectionFactory.createConnection();
```

In this case the logical IP address is assigned in the `hornetq-configuration.xml` file in the following tag:

```
<discovery-groups>
  <discovery-group name="my-discovery-group">
    <group-address>231.7.7.7</group-address>
    <group-port>9876</group-port>
    <refresh-timeout>10000</refresh-timeout>
  </discovery-group>
</discovery-groups>
```

Now that we have seen how to create a cluster we will move to our second example.

Creating a sharing queue on a HornetQ cluster

In our second example we will use a tricky method, that is, use a normal queue to share messages between two nodes. The difficult task in this case is the configuration of the environment considering that we are running a cluster with two nodes on the same machine, while in a real high-performance environment, as we stated earlier, we could think of a univocal assignment of machines to cluster nodes.

The reader, however, should test himself to see if he has a clear understanding of where to put his hands in order to make everything work fine.

In this case we will produce more than one `TextMessage` object to see whether the delivery is fast using HornetQ.

Configuring the environment

In this example we assume that we have two nodes that will form our cluster. The first one is located in the `HORNETQA_ROOT` folder and the second one is located in the `HORNETQB_ROOT` folder. We need to start each node separately and after this we will code our example. If you refer to the book's code you will find the configuration files ready to be used by simply copying them into the correct configuration folder.

To configure the first node we need to modify the following files, in both nodes:

- `hornetq-configuration.xml`: In this file we need to define how to create a cluster for sharing a queue
- `hornetq-jms.xml`: In this file we define the `ecg` queue that will be used
- `jndi.properties`: This file defines the ports we need to use

Considering the last point, we will define the ports to be used, not as command-line parameters as we did previously, but we will define them in configuration files, which is a more correct way to approach such problems.

We emphasize the point that changing the default ports is something that we need to do in order to avoid any "port already in use" Java exception at runtime. But if we are running the nodes on two separate machines (one machine one node) this is not necessary so the reader could run every node by simply changing only the `hornetq-configuration.xml` and the `hornetq-jms.xml` file.

Returning to our example, open the configuration file (`hornetq-configuration.xml`) and add the following tags for configuring the cluster:

```
<broadcast-groups>
   <broadcast-group name="bg-group1">
          <group-address>231.7.7.7</group-address>
          <group-port>9876</group-port>
          <broadcast-period>5000</broadcast-period>
          <connector-ref>netty</connector-ref>
   </broadcast-group>
</broadcast-groups>

<discovery-groups>
   <discovery-group name="dg-group1">
          <group-address>231.7.7.7</group-address>
          <group-port>9876</group-port>
          <refresh-timeout>10000</refresh-timeout>
   </discovery-group>
</discovery-groups>

<cluster-connection name="my-cluster">
   <address>jms</address>
   <connector-ref>netty-connector</connector-ref>
   <retry-interval>500</retry-interval>
   <use-duplicate-detection>true</use-duplicate-detection>
   <forward-when-no-consumers>true</forward-when-no-consumers>
   <max-hops>1</max-hops>
   <discovery-group-ref discovery-group-name="my-discovery-group"/>
</cluster-connection>
```

The `<cluster-connection>` tag tells that the JMS queue will be shared and that the connector used will be the netty connector. The node will retry to connect to the cluster every 500 milliseconds and `use-duplicate-detection` will warn every node not to duplicate the messages on the queue.

Now we need to open the `hornetq-jms.xml` configuration file and add the following tag for the queue:

```
<queue name="ECGQueue">
   <entry name="/queue/ECGQueue"/>
</queue>
```

Now we add the port configuration for the `jndi.properties` file. This is done by adding the following line to the `jndi.properties` file in the `HORNETQA_ROOT\config\standalone\clustered` folder:

```
java.naming.provider.url=jnp://localhost:1099
```

Once we are done with the first cluster settings, we need to move to the second `config` folder located at `HORNETQB_ROOT\config\standalone\clustered`.

Now we open the `hornetq-configuration.xml` folder and add the same cluster settings tags that we added on the `hornetq-configuration.xml` file of the first node of the cluster.

1. Open the `hornetq-jms.xml` file and add the same tag for creating the `ecg` queue.
2. Open the `jndi.properties` file and add the following properties:
   ```
   java.naming.provider.url=jnp://localhost:2099
   ```
3. As a final step we browse to the `HORNETQA_ROOT\bin` folder and execute the `./run.sh` command (`run.bat` in Windows).

In the same way we browse to the `HORNETQB_ROOT\bin` folder and execute the `./run.sh` command.

[Do not forget to remove the setting of the `CLUSTER_PROPS` variable in the `run.sh` or `run.bat` files, in case you are reusing the same nodes that you used in the previous example.]

Coding the clustered queue example

We are now ready to move to the example that is not so complicated compared to the previous one. Before doing anything, using your IDE, create a Java class called `ClusteredQueueExample` inside the `chapter06` package of the project.

Even for this case we will code the following steps:

1. Create two JMS connections, one for every node of the cluster.
2. Create two message consumers, one for every node of the cluster.
3. Create only one producer for the first node.
4. Send one message to the first node of the cluster.
5. Consume the message on both the nodes.

Moving on, we can start with the session creation.

Creating the connections

As we did before, we code the connection and the session in the following way:

```
Connection connectiona = null;
Connection connectionb = null;

InitialContext initialContexta = null;
InitialContext initialContextb = null;

String ECG_TEXT = "1;31/01/2012 15:45:01.100;1021;1022;1023";

Properties p = new Properties();
p.put(Context.INITIAL_CONTEXT_FACTORY,"org.jnp.interfaces.
NamingContextFactory");
p.put(Context.URL_PKG_PREFIXES,"org.jboss.naming:org.jnp.interfaces");
p.put(Context.PROVIDER_URL, "jnp://localhost:1099");
initialContexta  = new javax.naming.InitialContext(p);

ConnectionFactory cfa = (ConnectionFactory)initialContexta.lookup("/
ConnectionFactory");

p = new Properties();
p.put(Context.INITIAL_CONTEXT_FACTORY,"org.jnp.interfaces.
NamingContextFactory");
p.put(Context.URL_PKG_PREFIXES,"org.jboss.naming:org.jnp.interfaces");
p.put(Context.PROVIDER_URL, "jnp://localhost:2099");
initialContextb  = new javax.naming.InitialContext(p);

ConnectionFactory cfb = (ConnectionFactory)initialContexta.lookup("/
ConnectionFactory");
```

```
Session sessiona = connectiona.createSession(false, Session.AUTO_
ACKNOWLEDGE);
Session sessionb = connectionb.createSession(false, Session.AUTO_
ACKNOWLEDGE);

connectiona.start();
connectionb.start();
```

The code is the same as the previous one. We should point out that in this case the 2099 port is read not as a command-line parameter but is read from the `jndi.properties` file.

Creating the producer and the consumers

Now there is a big difference from the previous example, in this case we use the JMS message queue to create the producer and the consumers.

The code is pretty much the same as we used in *Chapter 1*, *Setting Up HornetQ*, so you should be familiar with it.

```
Queue queue = (Queue)initialContexta .lookup("/queue/ECGQueue");

MessageConsumer consumera = sessiona.createConsumer(queue);
MessageConsumer consumerb = sessionb.createConsumer(queue);

MessageProducer producer = sessiona.createProducer(queue);
```

As you can see here, there is a difference with the topic example, but this is something nearly transparent for the coder as the `createConsumer` and `createProducer` methods accept both a `javax.jms.Queue` and a `javax.jms.Topic` object.

We are now ready to see the final step of our example.

Producing and consuming the messages on the clustered queue

As a final step we will send 10 messages of the same type to node A of our cluster's queue on the `ECGQueue`. After this we will consume these 10 messages on both the nodes.

The code is nothing more than some `for` loops; let us start with the producer:

```
final int numMessages = 10;

for (int i = 0; i < numMessages; i++)
```

```
{
   TextMessage message = sessiona.createTextMessage(ECG_TEXT);

   producer.send(message);

   System.out.println("Sent message: " + message.getText() + new
Date());
}

Now, we can consume the ten messages on both nodes, we do the
consuming on the same for loop:
for (int i = 0; i < numMessages; i++)
{
   TextMessage messagea = (TextMessage)consumera.receive(5000);

   System.out.println("Got message: " + messagea.getText() + " from
node A");

   TextMessage messageb = (TextMessage)consumerb.receive(5000);

   System.out.println("Got message: " + messageb.getText() + " from
node B");
}
```

After we are done with everything we close every connection. Considering that we are working in a cluster environment this step is not mandatory, as having a connection open on a node or a cluster could heavily affect the performance of the node itself.

```
if (connectiona != null)
{
   connectiona.close();
}

if (connectionb != null)
{
   connectionb.close();
}

if (initialContexta  != null)
{
   initialContexta.close();
}

if (initialContextb != null)
{
   initialContextb.close();
}
```

As a final result we have seen that in this case we push 10 messages on the same queue and both of the consumers receive 10 messages. So every node receives the same messages. In the topic example, when the message is consumed by one client, it is consumed for all the other ones without duplication.

Final considerations

While concluding this chapter, we noticed that you could be very upset about configuring and using a HornetQ cluster. However, I would like to advise that all the tricky configuration tasks that we did in this chapter were necessary only because we used the same machine for running two nodes.

However I would like to suggest to the more willing readers to try a more a more complicated exercise that could be done to better understand that configuring a cluster can be a not-so-difficult task.

We suggest you download VirtualBox (`https://www.virtualbox.org/`) to emulate two HornetQ nodes on two machines. Considering that installing a Ubuntu OS can be a long operation, we suggest the reader to use a DamnSmall linux (`www.damnsmalllinux.org`) distribution that is Debian-based, so you can reuse the same commands of *Chapter 2*, *Setting Up HornetQ*, within only 50 MB of RAM.

Once we have both the machines running, the reader could simply install an fresh copy of the HornetQ server on every machine using *Chapter 2*, *Setting Up HornetQ* as reference.

So now after running both the nodes the reader should see that the cluster is formed without any other changes and that the code we provided works correctly.

Summary

This was a long chapter where we covered some aspects of HornetQ clustering. We have seen how to create a cluster with two nodes on the same machine and we saw how to configure and use topics on clusters and distributed queues. The reader should also refer to and study the examples provided with the HornetQ installation so as to understand all the possible configurations.

We are now ready to move on to a more intensive subject in *Chapter 7*, *Divert and Filter Messages*, where we will learn how to filter messages and to direct the stream of messages in different ways. So we will use different nodes but without using them as a cluster.

7
Divert and Filter Messages

In *Chapter 6*, *Clustering with HornetQ*, we learnt how to set up and dial with a cluster formed by one or more machines. We have also seen that on the same standalone HornetQ server, it is possible to create and use different queues (both core and JMS).

Such a configuration setup allows great flexibility and expands the different possibilities that an administrator or a coder has; to implement a high frequency message delivery system. Nevertheless, for the moment, if we want to send different types of messages to different queues you are forced to create sessions and producers on different queues.

HornetQ offers the possibility of setting up filters that work on the server side, so that it is possible to apply filters based on the messages, without any changes in the client logic.

The JMS specification allows the selector to act this way so that the HornetQ implementation works in this way, but the HornetQ filtering functionality, based on the implementation of JMS selectors, are based on a subset of SQL92, so they allow to write SQL-like expressions.

In this chapter we will learn how to filter messages and divert them. We will also learn the following concepts in detail:

- How to filter messages using the configuration files
- How to divert messages so as to send messages from one queue to another

HornetQ's divert allows to divert messages that were originally routed from one address to some other address, without making changes to the client code or logic.

In high frequency environments this is something that is essential. Imagine we have a single HornetQ instance that is running and it accepts messages produced by some client.

If for some reason we need to redirect some messages because, for example, the consumer is becoming slow, we could do this without having to update the code pertaining to every client.

As we have seen in our fictional example involving medical monitoring, it is very important to define thresholds and alerts considering that a higher heart rate could lead to **CHF** (**Chronic Heart Failure**), but it is also important to monitor any technical issues concerning devices. For example, if we are dealing with a three-line ECG, it is possible that a single ECG line disconnects, so there's a possible alarm that the device generates and it sends a message to the HornetQ server.

In this case, if a message with some different body text arrives, it is stored on the same queue that has been used to store ECG messages. In our example, we would like to intercept the non-ECG messages—even in a production environment—without changing the code on the device.

In the example that will be described during this whole chapter, we will create two servers that have different queues. These two servers are like bridges, so they act like a unique entity but without being clustered. The first queue will receive all the messages but in this case if one of the messages contains the pattern "alarm", the message is automatically moved to the other queue bound on a different server, where we will store only the alarm messages.

A more detailed description

Diverts are server-side configurations that allow to transparently route messages from one queue to another.

Diverts can be of the following two types:

- **Exclusive**: This means that the message is diverted from one address to another without being visible to the consumer of the first address
- **Non-exclusive**: In this, a copy is routed from the source address to the target address

In our example we will do something more; we will route messages from a queue on a HornetQ node to another queue on another target node. To divert our messages from one instance to another, we will create a JMS bridge between the two instances.

Bridges are logical software applications that allow us to connect to HornetQ instances so that it is possible to consume a message from a source queue or topic and then send them to a target queue or topic on another server.

Bridges are resilient to source or destination unavailability so they can be used specifically on a **World Area Network** (**WAN**).

The difference between the divert and bridge concept is that divert is designed to move messages from queues on the same instances of the HornetQ node.

Thus, you can use a divert for moving messages from one queue to another on the same node. While if you have different queues/topics on different HornetQ nodes you can use the divert coupled with the bridges.

Let us start with the configuration of the servers.

Configuring the servers

As we have seen in the previous chapter, we have the same problem by running both the HornetQ servers on the same machine. Once you have started the second server, the address is bound by the first instance. In this case we will not create a cluster but only create two HornetQ servers so that it is possible for the reader to see that the messages are filtered and diverted from the first server to the second server.

In this case we will create two standalone HornetQ severs by launching the `run.sh` (or `run.bat`) file twice with some alternative parameters. This method could also be successfully applied to some other configurations that need more HornetQ instances running on the same machine.

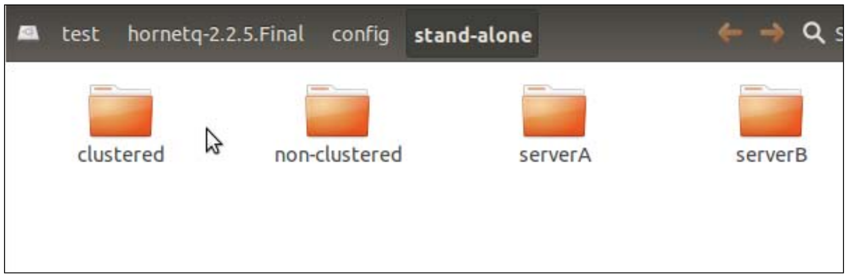
In that case, we need to start with a fresh new environment, remove all previous HornetQ installations and again run through the procedures as detailed in *Chapter 6, Clustering with HornetQ*.

So first of all, start by downloading a fresh version of HornetQ from the usual HornetQ download URL (`http://www.jboss.org/hornetq/downloads`).

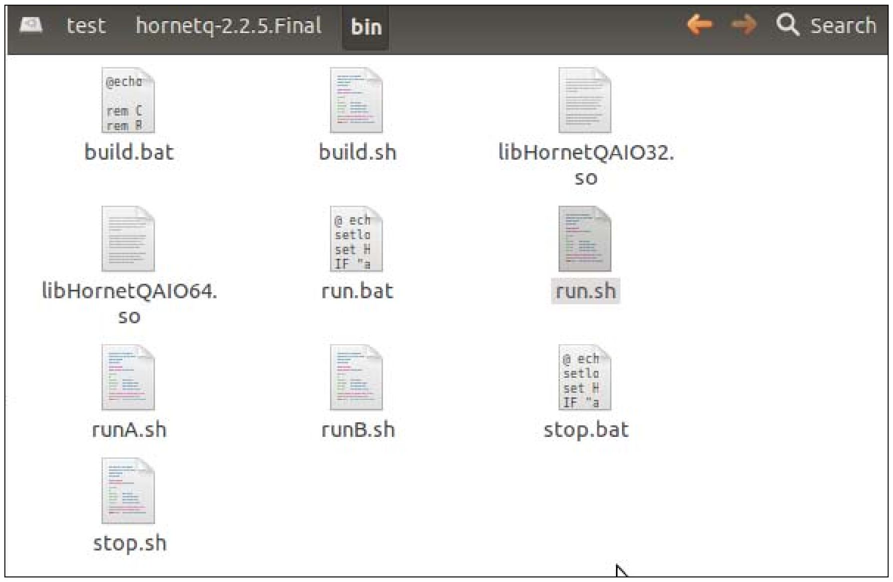
The main idea can be summarized in the following points:

- Duplicate the `config` folder where all the configuration files are stored
- Create another `HornetQ run.bat/run.sh` file so that it will be possible to launch two instances using the same `HORNETQ_ROOT` folder

Now, copy the `HORNETQ_ROOT\config\standalone\non-clustered` folder into two new folders, that is, `HORNETQ_ROOT\config\standalone\serverA` and `HORNETQ_ROOT\config\standalone\serverB`, respectively, so that the configuration should look like the one shown in the following screenshot on an Ubuntu system:



Now that we have both the folders, let us move to the `HORNETQ_ROOT\bin` folder and again create two copies of the `run.sh` file, and name them `runA.sh` and `runB.sh`. We only remind you that in Windows the file to be copied is `run.bat`. In the following screenshot the `HORNETQ_ROOT\bin` folder of our Ubuntu machine is shown with the changes described:



You should now be aware of what we are going to do. We will execute the `runA.sh` file and create an instance of HornetQ by pointing to the `HORNETQ_ROOT\config\standalone\serverA` folder and execute the `runB.bat` file by pointing to the `HORNETQ_ROOT\config\standalone\serverB` folder.

Recalling what we did in *Chapter 6, Clustering with HornetQ*, now open the `runA.sh` file and change the line that follows:

```
set CONFIG_DIR=%HORNETQ_HOME%\config\stand-alone\non-clustered
```

Replace this with the following one:

```
set CONFIG_DIR=%HORNETQ_HOME%\config\stand-alone\serverA
```

Now save the file and open the `runB.sh` file and replace the following:

```
set CONFIG_DIR=%HORNETQ_HOME%\config\stand-alone\non-clustered
```

With the following:

```
set CONFIG_DIR=%HORNETQ_HOME%\config\stand-alone\serverB
```

We also remind you that this configuration is possible without creating different copies of the `run.sh` file; it is also possible to launch the `run.sh` command from a terminal window with the path set to the new configuration folders as the first parameter.

Configuring the XML

Before configuring the XML, we need to better define what our example will do. Basically what we will do is divert messages from one server to another, and pass filtered messages from one queue to another. So we will work only on the server side without touching the client producer to route to different queues.

It is important to note that HornetQ allows us to divert messages in two ways, which are as follows:

- **Exclusive**: In this context the message to be diverted is intercepted prior to being inserted in the queue. So once diverted, the message is not present in the queue but only in its destination.
- **Non-exclusive**: Using such a configuration, the message is received by the first address, and then a copy is sent to the destination address.

In our example we will imagine that with every message associated to the heart rate that comes to the device randomly, a message containing an alert from the device itself can also be sent to the same queue. Nevertheless, every time an alert message from a device comes into the queue, it is diverted to a special queue where another consumer is listening, so as to allow the system to send an alert saying a particular device of a patient has encountered an unexpected problem.

In our configuration the server will act like the first large basin where all the messages are stored. But when an alert message gets filtered it is diverted to a queue on the second server.

To achieve this we need to perform the following steps:

1. Create the queue on the first server `ECGQueue` where all the messages arrive.
2. Create the filtering expression of the second queue.
3. Create the bridges that connect both the servers.

The last point, concerning the creation of bridges, is a key point. How does the first server know where to divert the messages from its queue to another address? We need to explicitly declare the destination.

To configure the first server, go to `HORNETQ_ROOT\config\stand-alone\serverB` and open the `hornetq-configuration.xml` file. Change this file using the following code (refer to this chapter's book code):

```
<configuration xmlns="urn:hornetq"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="urn:hornetq /schema/hornetq-
configuration.xsd">

   <clustered>true</clustered>

   <connectors>
      <connector name="netty-connector">
         <factory-class>org.hornetq.core.remoting.impl.netty.
NettyConnectorFactory</factory-class>
         <param key="port" value="5445"/>
      </connector>


      <connector name="alert-connector">
         <factory-class>org.hornetq.core.remoting.impl.netty.
NettyConnectorFactory</factory-class>
         <param key="port" value="5446"/>
      </connector>
   </connectors>

   <!-- Acceptors -->

   <acceptors>
      <acceptor name="netty-acceptor">
```

```
         <factory-class>org.hornetq.core.remoting.impl.netty.
NettyAcceptorFactory</factory-class>
         <param key="port" value="5445"/>
      </acceptor>
   </acceptors>

   <!-- Divert configuration -->

   <queues>
   <queue name="jms.queue.AlertQueue">
       <address>jms.queue.AlertQueue</address>
   </queue>
   </queues>

   <diverts>
      <divert name="alert-divert">
         <address>jms.queue.ECGQueue</address>
         <forwarding-address>jms.topic.AlertQueue</forwarding-address>
         <filter string="alert"/>
             <exclusive>true</exclusive>
      </divert>

   </diverts>

   <!-- Bridge configuration -->

   <bridges>
      <bridge name="alert-forward-bridge">
         <queue-name>jms.queue.ECGQueue</queue-name>
         <forwarding-address>jms.topic.AlertQueue</forwarding-address>
         <reconnect-attempts>-1</reconnect-attempts>
         <static-connectors>
            <connector-ref>alert-connector</connector-ref>
         </static-connectors>
      </bridge>
   </bridges>

   <!-- Other config -->

   <security-settings>
      <!--security for example -->
      <security-setting match="jms.#">
         <permission type="createDurableQueue" roles="guest"/>
         <permission type="deleteDurableQueue" roles="guest"/>
```

```
            <permission type="createNonDurableQueue" roles="guest"/>
            <permission type="deleteNonDurableQueue" roles="guest"/>
            <permission type="consume" roles="guest"/>
            <permission type="send" roles="guest"/>
         </security-setting>
      </security-settings>

   </configuration>
```

We need to detail everything because you could be messed up by all the configuration tags used.

Also, on this XML (or you can do the same thing on the `HornetQ-jms.xml` file), we now need to define the destination queue where we want to divert the messages with the alert in their body. The following is the configuration for that definition:

```
<queues>
<queue name="jms.queue.AlertQueue">
    <address>jms.queue.AlertQueue</address>
</queue>
</queues>
```

So, we have created all the connectors and the queue. Now, we need to declare how the divert should act. In this case we create an exclusive divert on the `ECGqueue` queue that will forward all the messages that contain the string alert to the `AlertQueue` queue in their body.

Now for the most interesting part; we create the divert as follows:

```
<diverts>
  <divert name="alert-divert">
    <address>jms.queue.ECGQueue</address>
    <forwarding-address>jms.queue.AlertQueue</forwarding-address>
    <filter string="alert"/>
    <exclusive>true</exclusive>
  </divert>
</diverts>
```

Apart from giving the divert a name, we define the two queues as source and destination. We define that our diverts will be based on a filter string that matches it with the message containing the alert. It is also possible to match the specific message property string and other attributes of the message such as the size or the priority. Note that we create the divert as exclusive so that no message with the alarm matching string will be stored in the `ECGQueue` queue.

Finally we need to create the bridge, the code for which is as follows:

```
<bridges>
  <bridge name="alert-forward-bridge">
    <queue-name>jms.queue.ECGQueue</queue-name>
    <forwarding-address>jms.topic.AlertQueue</forwarding-address>
    <reconnect-attempts>-1</reconnect-attempts>
    <static-connectors>
        <connector-ref>alert-connector</connector-ref>
    </static-connectors>
  </bridge>
</bridges>
```

Now we need to create the `ECGqueue` as usual for `serverA`, we hope you have learned how to do this without any reference code.

We now need to configure the XMLs of the second server. So, after moving to the `HORNETQ_ROOT\config\standalone\serverB` folder we need to open the `HornetQ-configuration.xml` file and simply create the connector for the port 5446 that we defined earlier, as shown in the following code snippet:

```
<connectors>
  <connector name="netty-connector">
    <factory-class>org.hornetq.core.remoting.impl.netty.
NettyConnectorFactory</factory-class>
    <param key="port" value="5446"/>
  </connector>
</connectors>

<acceptors>
  <acceptor name="netty-acceptor">
    <factory-class>org.hornetq.core.remoting.impl.netty.
NettyAcceptorFactory</factory-class>
    <param key="port" value="5446"/>
  </acceptor>
</acceptors>
```

As a last step, open the `HornetQ-jms.xml` file into the `HORNETQ_ROOT\config\standalone\serverB` folder and create the `AlertQueue` queue.

In this way, we end the configuration of the XML. We are now ready to move to the configuration of the starting scripts.

Changing the script

We here stress upon the concept that the configuration steps we used in this context are only needed because of the port binding by HornetQ. If you decide to have two different instances of HornetQ running on two different machines, you could still keep the configuration as the default one.

So when you use the default ports you could keep 5445 for the `NettyConnnector` and 1099 for the **JNDI**.

We also stressed the fact that using such default configurations when using two different machines (also on a WAM), requires careful configuration of the IP addresses, firewall, **NAT**, and port configurations. One of the main problems in such configurations is the firewall configuration, because then we need to allow access to port numbers that are not standard ones, like the port 80.

Moving on, in such cases we need to add some parameters to the JVM machine running the second instance of the HornetQ server, so move to the `HORNETQ_ROOT\bin` folder and open the `runB.sh` (`run.bat`) script file and replace the following line:

```
#export CLUSTER_PROPS="-Djnp.port=1099 -Djnp.rmiPort=1098 -Djnp.
host=localhost -Dhornetq.remoting.netty.host=localhost -Dhornetq.
remoting.netty.port=5445"
```

With the following one:

```
export CLUSTER_PROPS="-Djnp.port=2099 -Djnp.rmiPort=2098 -Djnp.
host=localhost -Dhornetq.remoting.netty.host=localhost -Dhornetq.
remoting.netty.port=5446"
```

Now save the file and from the command line in Ubuntu, launch the following command:

```
./run.sh
```

Coding the example

Now that we have both the HornetQ instances running, we are ready to code our example. First, to build the example, open your favorite IDE and create a new class named `DivertExample` as we did in *Chapter 6*, *Clustering with HornetQ* with a `main` method.

Now we are going to code for the following tasks:

- Creating a producer on the `ECGQueue` queue
- Sending two messages, one normal and one alert

- Creating two consumers; one for each `ECGQueue` queue and the other for the `AlertQueue` queue
- Checking that the message with the filter is diverted from the `ECGQueue` queue to the `AlertQueue` queue

Let us start by creating the producer, for the moment considering the entire example we have seen in the previous chapter. We leave the creation of the session and the connection to you, or to the book code. We will suppose that we have two **JMS** connections as the following code snippet suggests:

```
Queue ECGQueue = (Queue)initialContextA.lookup("/queue/ECQQueue");
Queue AlertQueue = (Queue)initialContextB.lookup("/queue/AlertQueue");

ConnectionFactory cfA = (ConnectionFactory)initialContextA.lookup("/
ConnectionFactory");
ConnectionFactory cfB = (ConnectionFactory)initialContextB.lookup("/
ConnectionFactory");


connectionA = cfA.createConnection();
connectionB = cfB.createConnection();

Session sessionA = connectionA.createSession(false, Session.AUTO_
ACKNOWLEDGE);
Session sessionB = connectionB.createSession(false, Session.AUTO_
ACKNOWLEDGE);
```

We leave the creation of `initialContextA` and `initialContextB` to you, reminding you that the Netty connection for the second one is on a different port.

Moving to the producer/consumer layer, we create and start both the connections, as shown in the following code snippet:

```
MessageProducer ECGProducer = sessionA.createProducer(ECGQueue);
MessageConsumer ECGConsumer = sessionB.createConsumer(ECGQueue);
MessageConsumer AlertConsumer = sessionB.createConsumer(AlertQueue);

connectionA.start();
connectionB.start();
```

Now it's time to produce and consume messages. First, we need to send the message using the `ECGProducer` producer and consume it in the queue. The following code snippet shows how this is to be done:

```
String ECG_TEXT = "1;31/01/2012 15:45:01.100;1021;1022;1023";
TextMessage ECGMessage = sessionA.createTextMessage(ECG_TEXT);

ECGProducer.send(ECGMessage);
```

```
System.out.println("Message " + ECGMessage.getText() + " send to
server A" );

TextMessage receivedMessageA = (TextMessage)ECGConsumer.receive(5000);

System.out.println("Receiving message " + receivedMessageA.getText() +
" from server A" );
```

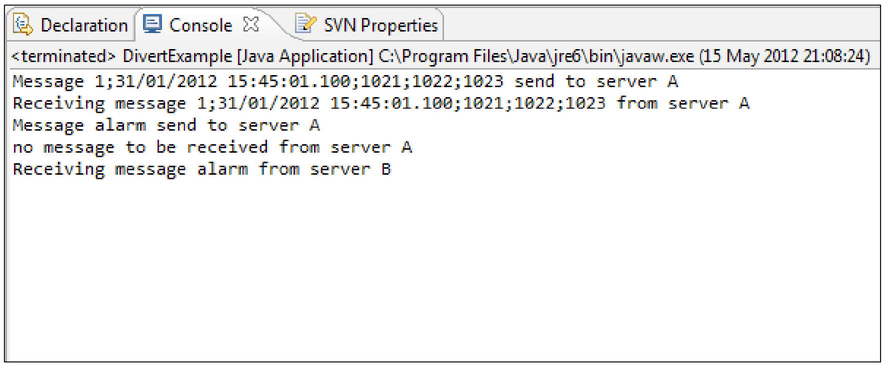
Now, we do the same thing on the `ECGQueue` queue with an alert message, only to demonstrate that in this case the consumer is not able to get the message because it is not in the queue. This is shown in the following code snippet:

```
String ALERT_TEXT = "this is an alert";
ECGMessage = sessionA.createTextMessage(ALERT_TEXT);
ECGProducer.send(ECGMessage);

System.out.println("Message " + ECGMessage.getText() + " send to
server A" );

TextMessage receivedMessageA = (TextMessage)ECGConsumer.receive(5000);

if (receivedMessageA == null)
 {
  System.out.println("no message to be received from server A" );
 }
```

Now, we can code the producer/consumer from the alert message that is diverted to the server into the `AlertQueue` queue. This is shown in the following code snippet:

```
String ALERT_TEXT = "alert";
ECGMessage = sessionA.createTextMessage(ECG_TEXT);

ECGProducer.send(ECGMessage);
System.out.println("Message " + ECGMessage.getText() + " send to
server A" );

TextMessage receivedMessageB = (TextMessage)AlertConsumer.
receive(5000);

System.out.println("Receiving message " + receivedMessageB.getText() +
" from server B" );
```

Executing the code on our Eclipse IDE will provide the following results:



```
<terminated> DivertExample [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (15 May 2012 21:08:24)
Message 1;31/01/2012 15:45:01.100;1021;1022;1023 send to server A
Receiving message 1;31/01/2012 15:45:01.100;1021;1022;1023 from server A
Message alarm send to server A
no message to be received from server A
Receiving message alarm from server B
```

As you can see from the code's point of view there's no need to make any changes. You only need to code the producer in the queue in which you diverted the message.

Some more advanced filtering possibilities

Filtering in HornetQ is not only a kind of filter for the body of the messages but as we said before, we also have the possibility to filter over the message properties.

HornetQ incorporates a functionality that allows us to do some kind of **ETL** (**extract, transform, and load**). Look at the HornetQ configuration's main file syntax, which is described in detail in the following URL:

`http://docs.jboss.org/hornetq/2.2.5.Final/user-manual/en/html_single/index.html#diverts`

The `<divert>` tag allows to specify another subtag known as `<transformer-class-name>`, where we can specify how the diverted message should be transformed.

For example, consider the example explained in this chapter. The diverted messages are the ones containing an alert that indicate that something went wrong. In this case, depending on the type of filter used, we need to alter the priority of the message before sending it to another queue.

To make this example scenario work in the right way, we need to do the following:

1. Stop both the server instances
2. Change the `divert` tag of the first `HornetQ-configuration.xml` file

3. Create the `transform` class by altering the message priority
4. Deploy everything
5. Check that everything works fine using the example code of the `DivertExample` class

So before continuing we need to stop both the server instances. Once done with this, re-open the `HornetQ-configuration.xml` file and modify the original code snippet that follows:

```
<diverts>
  <divert name="alert-divert">
    <address>jms.queue.ECGQueue</address>
    <forwarding-address>jms.topic.AlertQueue</forwarding-address>
    <filter string="alert"/>
    <exclusive>true</exclusive>
  </divert>
</diverts>
```

With the following one:

```
<diverts>
  <divert name="alert-divert">
    <address>jms.queue.ECGQueue</address>
    <forwarding-address>jms.topic.AlertQueue</forwarding-address>
    <filter string="alert"/>
    <transformer-class name>
      chapter06.AlterPriority
    </transformer-class-name>
 <exclusive>true</exclusive>
  </divert>
</diverts>
```

So we defined that the HornetQ server, before diverting the message that has an alert in the message body, should be transformed using the `AlterPriority` class contained in the code available with this book.

> You can create a JAR file containing every class that is needed for transforming a message before divert, but remember to put it on the classpath called by the `run.sh` script. The best way is to create a JAR file and put it into the `HORNETQ_ROOT\lib` folder, in this way they are automatically loaded at server startup.

Now open your IDE (**NetBeans** in our case) and add a new class called `AlterPriority` inside the `chapter07` package. Do not forget that the class should implement the `org.hornetq.core.server.cluster.Transformer` interface, which is contained in the `HornetQ-core.jar` file.

The generated code should look like the following code snippet:

```
public class AlterPriority implements Transformer
{
}
```

The method that we need to implement is the `transform` method. If you take a look at the `Transformer` interface in HornetQ's source code, you will notice that there's only a single method to implement, as shown in the following code snippet:

```
public interface Transformer
{
   ServerMessage transform(ServerMessage message);
}
```

As we can see, the `transform` method accepts a `ServerMessage` object and returns a new `ServerMessage` object.

In this case, by keeping in mind the example in *Chapter 5, Some More Advance Features of HornetQ*, we can alter the priority of the message with the following code:

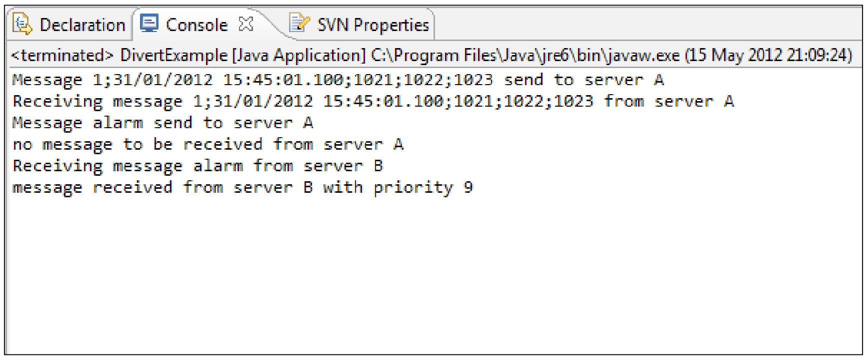
```
public class AlterPriority implements Transformer
{
   public ServerMessage transform(ServerMessage message)
   {
      return message.setPriority(9);
   }
}
```

In this case, the example was very easy because `ServerMessage` implements the method `setPriority` from the `org.hornetq.core.message.Message` interface.

Now reopen the `DivertExample` class and add the following lines at the end of the code:

```
System.out.println("Receiving message " + receivedMessageB.getText() +
" from server B" );

System.out.println("message received with priority " +
receivedMessageB.getPriority() + " from server B" );
```

At the end, launch both the servers using the `runA.sh` and `runB.sh` script files respectively and relaunch the `DivertExample`. You should see the following output:



```
<terminated> DivertExample [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (15 May 2012 21:09:24)
Message 1;31/01/2012 15:45:01.100;1021;1022;1023 send to server A
Receiving message 1;31/01/2012 15:45:01.100;1021;1022;1023 from server A
Message alarm send to server A
no message to be received from server A
Receiving message alarm from server B
message received from server B with priority 9
```

So as we have seen, the `transform` class intercepts the message. The message is caught before it has been diverted.

Again, we point out that this simple example serves only for demonstration purposes. We hope that you can grasp the great flexibility of the HornetQ server's `transform` class. Following the user needs, for example, it would be possible to store every diverted message or to trigger other server side functions.

Summary

In this chapter we learned the following:

- How to create a server-side divert between two different JMS queues
- We associated the divert to a filter on the message text
- Finally we have seen how to apply a simple transform to the diverted message so as to alter its priority

Now, we can move on to the next chapter where we will go into the details of the message flow.

8
Controlling Message Flow

In *Chapter 4*, *Monitoring HornetQ*, we learnt how to manage the messages in a queue so that we could control their state and behaviour using JMS and Core APIs. This approach is useless in cases where we would like to monitor a HornetQ instance that is under pressure because of too many messages.

This could happen due to various reasons; as we have seen in *Chapter 7*, *Divert and Filter Messages*, it is possible to deliver messages to another queue without changing the producer layer. Using the same scenario, when a device has some fault or it raises an alarm, the alarm is diverted to another queue on a different HornetQ instance.

In such cases, if we have multiple alarms it is possible that the queue that stores the alarms starts growing fast so as to cause an out-of-memory exception.

Using the many features that HornetQ provides, it is possible to control the flow of data in the following two ways:

- **Client to server side**: This means that it is possible to control the message flow of data from the consumer/producer layer to a HornetQ running instance
- **Server to server side**: In this case it is possible to control the message flow of data between two instances of HornetQ

This chapter will explain how it is possible to control the flow from both the JMS sides that use the Core API. In particular we will see the following:

- How to size the dimensions of the queues
- How to control the number of messages consumed by the producer layer

We will give some examples both for the JMS and the Core API cases.

What happens when you send or receive a message (for real)

High performance environments have managing issues that need different approaches with respect to client-server environments, which are not designed to handle a high, parallel number of tasks.

Even if HornetQ uses a transparent way to manage the messages from the client side using the send or receive methods for consumers, it uses a buffer to prefetch messages so as to avoid multiple client requests for a message every time there is a request to the queue where the message is stored.

This greatly improves performance, but a good way to tune these performances is to tune the size of the buffer. The default value for this buffer is 1MB (1024 bytes x 1024 bytes).

If you are using JMS, it is possible to alter the value of the buffer size. Let us see how to do this.

Changing the size of the buffer for JMS connections

This task is very simple. From the JMS point of view, changing the value of the buffer means only changing a tag in the `HornetQ-jms.xml` file.

The possible values for this tag are as follows:

- `-1`: This value tells HornetQ to allow the buffer size to grow without limit
- `0`: This value tells HornetQ that the size is 0 megabytes so that every message is not buffered
- **Any positive integer number**: This value fixes the maximum size of HornetQ's buffer consumer layer to the megabytes of this number

There are no general rules concerning the value to be set. We can underline the extremes, `0` and `-1`, so that you can grasp at least how to treat them.

In this case the size of the buffer depends also on the fact that the consumer in a high frequency environment has very short mean time to consume the messages. This means that if you need to consume billions of messages per second a simple calculation gives us the average time for a message to be consumed is in the order of 0.01 milliseconds.

According to the mean client consumer time we can have the following possibilities:

- **Fast consumer**: In this case the consumer processes the messages in a very fast way, so that they can be passed from the queue to the buffer, which in this case is unsized.
- **Slow consumers**: In this case the consumer needs some time to process every message. Processing a large message that is an XML could be an example of a slow consumer. On a queue we can have more connections and more consumers, so it is important to note that if one of the clients is slow in consuming the messages, then it affects the performance of the others. This is because while the consumer is consuming its message the other consumers will wait until the conclusion of the task.

For fast consumers, you can set the Windows buffer value to `-1`. We suggest you take this action with caution. This is because if for any reason the client starts to become slow in processing messages, then there is a queue of unprocessed messages in the client, because of which the client's memory consumption starts to grow. Thus, in the worst case scenario it can reach the limit size and raise an out-of-memory runtime exception.

For slow consumers you can set the windows buffer size to `0`, so that there is no buffer and all the messages remain on the server side and none of them go in the buffer.

We point out that 1MB as a default value will fit most of the previous scenarios.

Buffer size using Core API

The same concepts are reprised when dealing with the Core API. In this case the Windows buffer size parameter can be assigned using a method of the `ClientSessionFactory` class. We have given a brief example code. In this case, we will create an API queue at runtime to avoid boring the user with configuration issues. In this case, we create and produce the usual ECG text on an `ECGQueue` API as we did in *Chapter 2*, *Setting Up HornetQ*, but we set an unbound buffer size on the connection.

So fire up your preferred IDE and create a new class called `WindowBufferApiExample` with a `main` method in the `chapter08` package. Once done, in the `main` method add the following code:

```
HashMap map = new HashMap();
map.put("host", "localhost");
map.put("port", 5445);

TransportConfiguration configuration = new TransportConfiguration(Nett
yConnectorFactory.class.getName(), map);
```

```
ServerLocator serverLocator = null;
ClientSessionFactory factory = null;
ClientSession session = null;

try {

    serverLocator = HornetQClient.createServerLocatorWithoutHA(config
uration);

    factory = serverLocator.createSessionFactory();
    ((ServerLocator) factory).setConsumerWindowSize(0);
    session = factory.createSession(false, false, false);

    session.createQueue("ECGQueue", "ECGQueue", true);
    ClientProducer producer = session.createProducer("ECGQueue");
    System.out.println(producer.isClosed());
    ClientMessage message = session.createMessage(true);
    message.getBodyBuffer().writeString("Hello");
    message.setExpiration(60000);
    System.out.println("message = " + message.getBodyBuffer().
readString());
    producer.send(message);

    session.start();
    ClientConsumer consumer = session.createConsumer("ECGQueue");

    ClientMessage msgReceived = consumer.receive(100);
    System.out.println("message received "+ msgReceived.
getBodyBuffer().readString());
    session.close();

}catch(Exception ex){
  ex.printStackTrace();
} finally {
  if (session != null)
    session.close();
  if (factory != null)
    factory.close();

}
```

This code will simply produce and consume a message using the Core API. The difference with the previous example is that in this case we set from the code the consumer window size to `0` using the following line:

```
((ServerLocator) factory).setConsumerWindowSize(0);
```

So, in the case of a slow consumer, a buffer is not used for the messages produced within that session.

Controlling the consuming rate

As we have seen, it is possible to control the size of the buffer, but HornetQ also offers the possibility to control the rate at which a consumer should consume the messages.

The use of this setting is important because this setting allows us to control if a client consumes messages at a rate above the one specified. Again, as in the case of Windows buffer size, this parameter can be configured in XML for JMS and as a method for Core API.

The possible values are as follows:

- `-1`: This is the default value. This value means that the rate limited flow control is disabled so the consumer does not have a time limit for consuming messages.
- **A positive integer number**: This is the maximum message per seconds number that can be consumed by a client. Obviously this parameter can be configured coupled to the Windows buffer size, so that it is possible to tell a consumer not to consume more that 10 messages per second and without any time limit.

If you are using the Core API, this parameter can be set using a method of the `ClientSessionFactory` object. The following is a snippet of code that can help:

```
ClientSessionFactory.setConsumerMaxRate(int consumerMaxRate)
```

This method is for the entire connection factory, but you can also set the `consumerMax Rate` at a session level using the `CreateConsumer` method with the following parameters:

```
createConsumer(SimpleString queueName,
               SimpleString filter,
               int windowSize,
               int maxRate,
               boolean browseOnly)
```

Only for demonstration purposes, we gave a brief example using JMS for setting the `consumerMaxRate` parameter, to avoid more than ten messages per second being consumed from a client.

First the configuration; go to the `HORNETQ_ROOT\config\standalone\non-clustered` folder and open the `HornetQ-jms.xml` file. Now add the `<consumer-max-rate>10</consumer-max-rate>` tag between the `<connection-factory name="ConnectionFactory">` tag. Once the file is saved, we run the HornetQ server.

Now, we create the `ConsumerMaxRateExample` class using our preferred IDE inside the `chapter08` package with the `main` method.

Inside the `main` method add the following code:

```
String ECG_TEXT = "1;02/20/2012 14:01:59.010;1020,1021,1022";
java.util.Properties p = new java.util.Properties();

p.put(javax.naming.Context.INITIAL_CONTEXT_FACTORY,"org.jnp.
interfaces.NamingContextFactory");
p.put(javax.naming.Context.URL_PKG_PREFIXES, "org.jboss.naming:org.
jnp.interfaces");
p.put(javax.naming.Context.PROVIDER_URL, "jnp://localhost:1099");

initialContext = new javax.naming.InitialContext(p);

Queue queue = (Queue)initialContext.lookup("/queue/ECGQueue");
ConnectionFactory cf = (ConnectionFactory)initialContext.lookup("/
ConnectionFactory");
connection = cf.createConnection();

Session session = connection.createSession(false, Session.AUTO_
ACKNOWLEDGE);

MessageProducer producer = session.createProducer(queue);
MessageConsumer consumer = session.createConsumer(queue);

connection.start();

final int numMessages = 150;

for (int i = 0; i < numMessages; i++){
  TextMessage message = session.createTextMessage(ECG_TEXT);
  producer.send(message);
}

System.out.println("Sent messages");
System.out.println("Will now try and consume as many as we can in 10
seconds ...");

final long duration = 10000;
```

```
int i = 0;

long start = System.currentTimeMillis();

while (System.currentTimeMillis() - start <= duration){
  TextMessage message = (TextMessage)consumer.receive(2000);
  i++;
}

long end = System.currentTimeMillis();

double rate = 1000 * (double)i / (end - start);

System.out.println("We consumed " + i + " messages in " + (end -
start) + " milliseconds");
System.out.println("Actual consume rate was " + rate + " messages per
second");
```

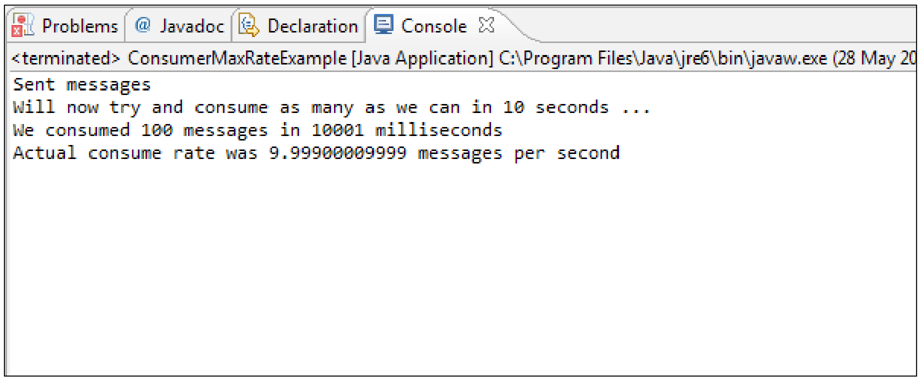
The previous code basically calculates the consumer rate of a producer on a queue where we previously sent 150 messages.

After sending these 150 messages to the `ECGQueue`, we receive all the messages for 10 seconds. Finally, we calculate the mean of messages received during the 10 seconds. The formula is the ratio between the number of messages consumed and the time in seconds during which the consuming task took place.

The portion of code is the one that follows:

```
long start = System.currentTimeMillis();

while (System.currentTimeMillis() - start <= duration)
{
  TextMessage message = (TextMessage)consumer.receive(2000);
  i++;
}

long end = System.currentTimeMillis();

double rate = 1000 * (double)i / (end - start);

System.out.println("We consumed " + i + " messages in " + (end -
start) + " milliseconds");
System.out.println("Actual consume rate was " + rate + " messages per
second");
```

One single execution of the code provides the following output in the Eclipse console:



```
<terminated> ConsumerMaxRateExample [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (28 May 20
Sent messages
Will now try and consume as many as we can in 10 seconds ...
We consumed 100 messages in 10001 milliseconds
Actual consume rate was 9.99900009999 messages per second
```

As you can see, we have the consumption limited to 10 messages per second. So the consuming rate was respected once set up in the configuration file.

Controlling the producer side

In the previous section we have seen how it is possible to limit the Windows buffer size and the maximum rate of the messages consumed within a queue, at least using the JMS specification.

But considering that the clients are the ones that send messages, it is possible that they can potentially block the consumer by sending too many messages. HornetQ provides some parameters from the producer's side so that it is possible to manage the number of messages sent to the queues.

HornetQ offers the same possibilities that we have seen from the consumer layer to the producer layer.

In our example, all the medical messages sent to the `ECGQueue` in general are not of a fixed size. HornetQ offers the possibility to set the maximum size of memory, so as to limit the maximum size of the messages, sent from any producer to the specific queue, to a fixed value.

In this case, when the threshold value is reached, the producer receives a block on their side waiting for the memory to return to the threshold value.

You should not forget that HornetQ is able to manage millions of messages in the same queue. However, a bunch of messages could overflow the queue at a point where the producer and consumers start becoming less and less performant. In a high frequency environment this could be very dangerous.

In the next example, we will create a JMS configuration such that the producer has a minimal amount of allowed memory. In this case, we will send lots of messages so as to force the server to block the producer from sending other messages.

As usual, before coding our example we will do the necessary steps for configuring the environment.

First open the `HornetQ-configuration.xml` file and add the following tags:

```
<address-settings>
  <address-setting match="jms.queue.ECGQueue">
    <max-size-bytes>100000</max-size-bytes>
    <address-full-policy>BLOCK</address-full-policy>
  </address-setting>
</address-settings>
```

As you can see, the tags used in the previous code block basically told HornetQ the following directives:

- It allows the `ECGQueue` a maximum of 100 KB as producer windows buffer
- In case the limit is reached, then the policy is to block producers from sending other messages until the consumer has freed the resource

The `<address-full-policy>` tag has the following possible values:

- `PAGE`: This is the default value, which means that when the `max-size-bytes` is reached, HornetQ starts paging. So the messages are temporarily stored in a disk, like a kind of swap files, as in the case of the OS system.
- `DROP`: This value tells HornetQ to simply drop all messages until the value of the buffer returns above the limit.
- `BLOCK`: This is probably the most interesting value. In this case the producer is not allowed to send messages.

Now that we have set the block feature for the tag, we need to create the `ECGqueue` in the `HornetQ-jms.xml` file.

Now we can code our example. First, we will produce messages until the thread threshold value is reached.

Let us start with the class creation and the connection settings. Open your IDE and create a new class called `JMSBlockExample` inside the `chapter08` package and with a `main` method.

Inside the `main` method add the following code for creating the connections, session, and producer:

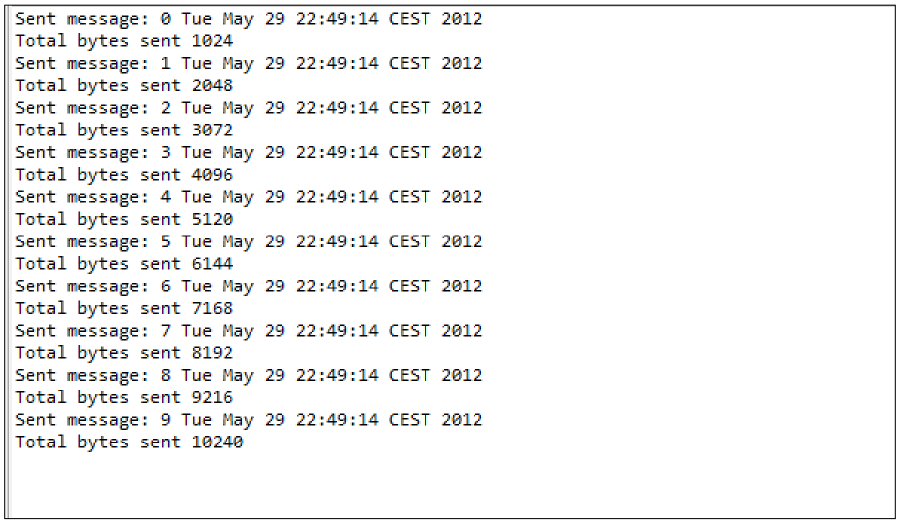
```
java.util.Properties p = new java.util.Properties();

p.put(javax.naming.Context.INITIAL_CONTEXT_FACTORY,
"org.jnp.interfaces.NamingContextFactory");

p.put(javax.naming.Context.URL_PKG_PREFIXES,
"org.jboss.naming:org.jnp.interfaces");

p.put(javax.naming.Context.PROVIDER_URL, "jnp://localhost:1099");

initialContext = new javax.naming.InitialContext(p);

Queue queue = (Queue)initialContext.lookup("/queue/ECGQueue");

QueueConnectionFactory cf = (QueueConnectionFactory)initialContext.
lookup("/ConnectionFactory");

connection = cf.createQueueConnection();
connection.start();

QueueSession session = connection.createQueueSession(false, Session.
AUTO_ACKNOWLEDGE);

MessageProducer producer = session.createProducer(queue);
```

To demonstrate what happens by setting the `<address-full-policy>` tag, we will send a bunch of messages to the `ECGQueue` and see from the output the effects of the configuration settings.

Now, we are ready to move to the interesting part of the code where we need to send 1000 messages. This action is pretty simple with the following code:

```
for (int i = 0; i<1000; i++) {
  BytesMessage message = session.createBytesMessage();
  message.writeBytes(new byte[10 * 1024]);
  producer.send(message);
  System.out.println("Sent message: " + i + " " + new Date());
}
```

Now, if we execute the program, the output will be the following:



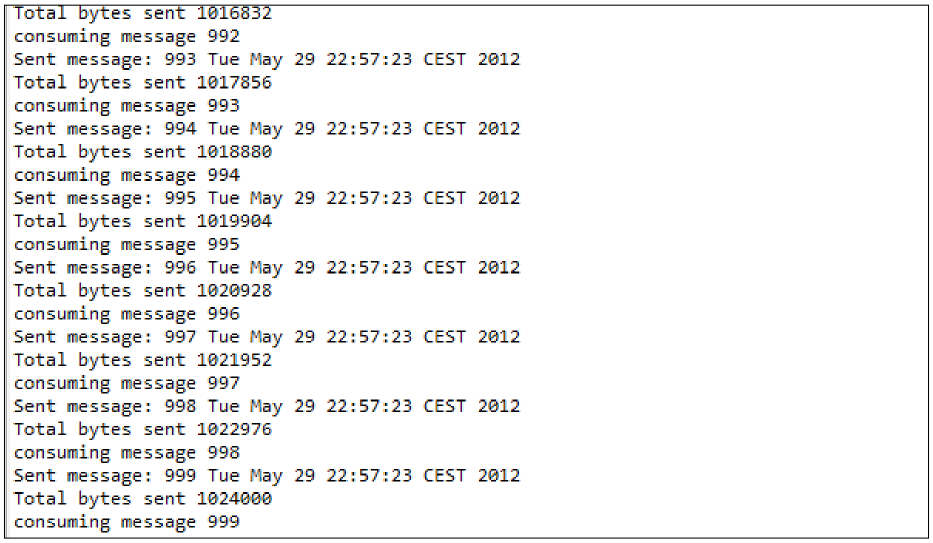
```
Sent message: 0 Tue May 29 22:49:14 CEST 2012
Total bytes sent 1024
Sent message: 1 Tue May 29 22:49:14 CEST 2012
Total bytes sent 2048
Sent message: 2 Tue May 29 22:49:14 CEST 2012
Total bytes sent 3072
Sent message: 3 Tue May 29 22:49:14 CEST 2012
Total bytes sent 4096
Sent message: 4 Tue May 29 22:49:14 CEST 2012
Total bytes sent 5120
Sent message: 5 Tue May 29 22:49:14 CEST 2012
Total bytes sent 6144
Sent message: 6 Tue May 29 22:49:14 CEST 2012
Total bytes sent 7168
Sent message: 7 Tue May 29 22:49:14 CEST 2012
Total bytes sent 8192
Sent message: 8 Tue May 29 22:49:14 CEST 2012
Total bytes sent 9216
Sent message: 9 Tue May 29 22:49:14 CEST 2012
Total bytes sent 10240
```

As you can see, this is the effect of the `BLOCK` setting. The producer is no longer able to send messages until a consumer frees the queue from some messages.

If we modify the code by adding the consumer inside using the following code:

```
for (int i = 0; i<1000; i++) {
  BytesMessage message = session.createBytesMessage();
  message.writeBytes(new byte[10 * 1024]);
  producer.send(message);
  System.out.println("Sent message: " + i + " " + new Date());
  message = (BytesMessage)messageConsumer.receive(3000);

  System.out.println("consuming message "+ i);

}
```

Then the output will be the following one:



```
Total bytes sent 1016832
consuming message 992
Sent message: 993 Tue May 29 22:57:23 CEST 2012
Total bytes sent 1017856
consuming message 993
Sent message: 994 Tue May 29 22:57:23 CEST 2012
Total bytes sent 1018880
consuming message 994
Sent message: 995 Tue May 29 22:57:23 CEST 2012
Total bytes sent 1019904
consuming message 995
Sent message: 996 Tue May 29 22:57:23 CEST 2012
Total bytes sent 1020928
consuming message 996
Sent message: 997 Tue May 29 22:57:23 CEST 2012
Total bytes sent 1021952
consuming message 997
Sent message: 998 Tue May 29 22:57:23 CEST 2012
Total bytes sent 1022976
consuming message 998
Sent message: 999 Tue May 29 22:57:23 CEST 2012
Total bytes sent 1024000
consuming message 999
```

In this case we have reached the limit of the `for` loop because the `BLOCK` option was never activated.

This example should have illustrated to you the fact that it is possible to control, using HornetQ, nearly every aspect of the memory management of the queues and the messages that can be found inside them.

If you are interested in fine tuning a high frequency message framework, we suggest trying to tune both the producer and the consumer in a queue to find the best configuration for your production environment.

This would not be difficult using the previous examples. In a testing scenario, we could send random messages, whose dimension in bytes is decided using the `random` Java class, so as to have as near to the real status as possible.

We also point out that these features can also be applied to a large messages queue. So if you are planning to use HornetQ for exchanging files of other large information messages, the control features can be applied in the same way.

From the producer layer's point of view there is also the possibility to use the `<producer-max-rate>` tag to force the queue to have only a predefined number of messages.

For a cluster environment, this tag can also be used to have some size-defined or number-defined queues, even if they are shared. This is done to be sure that at any moment the maximum number of messages in a queue could not be above the set limit.

Summary

HornetQ allows the developer and the system administrator to fine-tune the way in which the messages are stored/consumed by the consumer and the producer, both using JMS or Core API.

In this chapter, we have seen how to set the Windows buffer size so as to instruct HornetQ when to use the buffer and when not to.

We have also coded some examples to show how to control the rate of consumption of messages.

Finally, we have seen how to limit the size of the queue and instructed HornetQ on how to manage the over-threshold situation.

In the next chapter, we will look at the problem of security in HornetQ that is another important configuration task, and is often underestimated.

9 Ensuring Your HornetQ Security

Let us admit it, security is, in most cases, not the primary goal of a developer when coding software. But in the last few years as the web programming paradigm took place, security has become more and more relevant when coding.

Considering that the main focus of a Message Oriented Framework is to exchange messages and so information, it is very important to ensure that this process takes place in a secure context.

In this chapter, we will see how to secure HornetQ, both at the configuration layer and at a software layer, using the various instruments the framework offers to gain this objective.

We will learn how to:

- Configure the HornetQ instance to allow only some operation to the consumer/producer layer
- Configure the transport to add to the instance a more secure way to exchange data
- Code a secure message exchange

At the end the user will be able to re-think the example code in a new way. We will use the example scenario that drives us till now, because medical data is in most of the countries subject to a strict law. Hence particular attention is required when treating it in an electronic way.

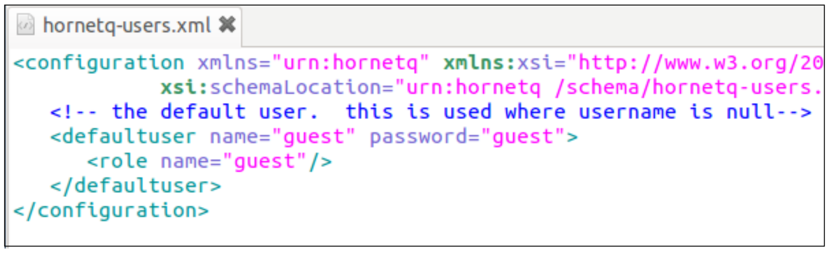
Configuring the HornetQ security

If you remember in *Chapter 4*, *Monitoring HornetQ*, where we dealt with the management API, we were required to add a tag in the `HornetQ-configuration` file to completely disable all the security checks that HornetQ uses by default when dealing with queues.

We set the `<security-enabled>` tag to `false` (the default is `true`). This was done because otherwise at runtime we would have a security exception when trying to use the management API. Obviously, it is strongly suggested not to use this tag outside of a developing environment. So a first check you can perform, when going in production, is to remove any `<security-enable>` tag from your `HornetQ-configuration.xml` files.

When starting a single instance, HornetQ uses an internal mechanism based on users and roles to decide which action is allowed and which one is not.

Let us start by identifying the users in a HornetQ environment. If you remember, HornetQ uses a configuration file, that is `HornetQ-users.xml`, for defining the users. The file is located in the `HORNETQ_ROOT\config\standalone\clustered` or into the `HORNETQ_ROOT\config\standalone\non-clustered` folders. If we take a look at a fresh installed version for both, you should see the following configuration:



```
<configuration xmlns="urn:hornetq" xmlns:xsi="http://www.w3.org/20
            xsi:schemaLocation="urn:hornetq /schema/hornetq-users.
   <!-- the default user.  this is used where username is null-->
   <defaultuser name="guest" password="guest">
      <role name="guest"/>
   </defaultuser>
</configuration>
```

As you can see, by default when a producer, consumer, or management process tries to access the HornetQ instance, and it does not provide any login information, then the default user is used.

The `role` tag accepts user defined tags so as to leave the user free to define a personalized `role` tag, for a single queue/topic, so roles are not predefined but user defined. One file example of user defined roles is as follows:

```
<configuration xmlns="urn:hornetq"  xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance" xsi:schemaLocation="urn:hornetq ../schemas/
hornetq-users.xsd ">

<defaultuser name="guest" password="guest">
```

```
<role name="guest"/>
</defaultuser>

<user name="ecg" password="ecg001">
<role name="admin"/>
</user>

<user name="management" password="management001">
<role name="admin"/>
<role name="guest"/>
</user>

<user name="egc1" password="ecg101">
<role name="ecg-users"/>
<role name="guest"/>
</user>

</configuration>
```

In this case, apart from the default user, we have three other users:

- `ecg`: This has administration roles
- `management`: This has both guest and administration roles
- `ecg1`: This has guest and ecg-users roles

Coupled with this, we can have the following security setting on the associated `HornetQ-configuration.xml` file:

```
<security-setting match="#">
<permission type="createDurableQueue" roles="admin"/>
<permission type="deleteDurableQueue" roles="admin"/>
<permission type="createNonDurableQueue" roles="admin, ecg-users"/>
<permission type="deleteNonDurableQueue" roles="admin, ecg-users"/>
<permission type="send" roles="guest,admin, ecg-users"/>
<permission type="consume" roles="admin, ecg-users"/>
</security-setting>
```

Going through the configuration we will find the following case:

- A default user can only send messages on any queue
- An `ecg` user can do anything but he cannot consume messages (does not have `consume` as a permission)
- A management user can do anything on any queue
- An `ecg1` user can only send/consume messages because `ecg` users are bound only to send/consume permissions

Notice that in this case we have used the wildcard `#` that defines these permissions for every queue and topic. The permissions are applied both to JMS and core queues/topics inside the instance.

But it is possible to bind permissions only to some queues identified by wildcards. So for example, we could have multiple `security-setting match` tags that define some common permissions for a queue but can give specific permissions to an other specific queue.

As an example, let us analyze the following configuration:

```
<security-setting match="jms.queue.#">
  <permission type="createDurableQueue" roles="user"/>
  <permission type="deleteDurableQueue" roles="user"/>
  <permission type="createNonDurableQueue" roles="user"/>
  <permission type="deleteNonDurableQueue" roles="user"/>
  <permission type="send" roles="user"/>
  <permission type="consume" roles="user"/>
</security-setting>

<security-setting match="jms.ecg.queue.#">
  <permission type="createDurableQueue" roles="user"/>
  <permission type="deleteDurableQueue" roles="user"/>
  <permission type="createNonDurableQueue" roles="user"/>
  <permission type="deleteNonDurableQueue" roles="user"/>
  <permission type="send" roles="ecg-user"/>
  <permission type="consume" roles="ecg-user"/>
</security-setting>
```

In this case, for the queues that start with the word `queue`, the user without a defined role can do nothing. But for queues that start with `ecg.queue`, only the user that has the `ecg-user` role can send/consume messages.

The security mechanism does not have an inherited logic. This allows maximum flexibility on configuring roles/permissions/users on single queues/topics.

In a production environment, we strongly suggest to define at least default permission settings to avoid untrusted queue generations. In general, users that send messages should not have the permission to modify the behavior of the queues/topics. These kind of permissions should be done only by the user with administrative permission. Similarly, admin users should not have the permission to send/consume messages on queues apart from the management ones that we have seen in *Chapter 4*, *Monitoring HornetQ*.

Only to resume, the types of permissions allowed are as follows:

Permission type	Permission granted
`createDurableQueue`	This permission allows the user to create a durable queue.
`deleteDurableQueue`	This permission allows the user to delete a durable queue.
`createNonDurableQueue`	This permission allows the user to create a non-durable queue.
`deleteNonDurableQueue`	This permission allows the user to delete a non-durable queue.
`send`	This permission allows the user to send a message to matching addresses.
`consume`	This permission allows the user to consume a message from a queue bound to matching addresses.
`manage`	This permission allows the user to invoke management operations by sending management messages to the management address.

We can manage, in a very detailed way, how the queue should be used by HornetQ users. Remember that as every JMS queue is bounded internally to a HornetQ core queue, such permissions also become permissions on the associated core queues.

Resuming the example on security and permissions

To resume what we have seen in the previous section, we gave an example by securing our example scenario. Our goals are as follows:

- Avoiding any connection without login/password mechanism
- Creating one user that is only able to send messages on queues
- Creating one user that is only able to consume messages on queues
- Creating one user that is able to create queues but cannot produce or consume messages on those queues

Using such permissions, we distinctly separate the consumer permissions from the producer's permissions. In this way a producer could connect and send message, but it does not have the possibility to consume messages on the same queue.

To add another level, such that all this permission should be applied to the `ecg` queue, we also set this permission only for the `EcgExample` queue.

To achieve our goal we need to modify three files:

- `HornetQ-users.xml`: For creating the users needed
- `HornetQ-jms.xml`: To define the `EcgExample` queue
- `HornetQ-configuration.xml`: To create the security setting for the users for the final queue

As usual, all the files are stored in the `HORNETQ_ROOT\config\standalone\non-clustered` folder. Let us start by opening the `HornetQ-users.xml` file with an editor and add the following users:

```
<configuration xmlns="urn:hornetq" xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance"
            xsi:schemaLocation="urn:hornetq /schema/hornetq-users.
xsd">

<user name="admin" password="admin01">
<role name="manager"/>
</user>

<user name="ecgconsumer" password="ecgconsumer01">
<role name="consumer"/>
</user>

<user name="ecgproducer" password="ecgproducer01">
<role name="producer"/>
</user>

</configuration>
```

As the reader can see, there is no default user. So any connection without providing credential, as we did in all previous chapters, will be refused.

We now need to define the queue by opening the `hornetq-jms.xml` file and by adding the following tag:

```
<queue name="ECGQueue">
  <entry name="queues/ECGQueue"/>
</queue>
```

Now that we have created the users and the queue, we need to bind the permission from users to the queue. To do this, we need to open the `hornetq-configuration.xml` file and add the following tag:

```
<security-settings>

  <security-setting match="jms.queue.ECGQueue">
  <permission type="createDurableQueue" roles="manager"/>
  <permission type="deleteDurableQueue" roles="manager"/>
  <permission type="createNonDurableQueue" roles="manager"/>
  <permission type="deleteNonDurableQueue" roles="manager"/>
  <permission type="send" roles="producer"/>
  <permission type="consume" roles="consumer"/>
  </security-setting>

  <security-setting match="#">
  <permission type="createDurableQueue" roles="manager"/>
  <permission type="deleteDurableQueue" roles="manager"/>
  <permission type="createNonDurableQueue" roles="manager"/>
  <permission type="deleteNonDurableQueue" roles="manager"/>
  <permission type="send" roles="manager"/>
  <permission type="consume" roles="manager"/>
  </security-setting>

</security-settings>
```

As the reader can see, if we go through the permissions we can identify the following:

- On the `ECGQueue` queue the user admin can create and delete durable or non-durable queues on the server, but it cannot send or consume messages on that queue
- The user `ecgconsumer` can consume messages on the queue `ECGQueue`
- The user `ecgproducer` can send messages on the queue `ECGQueue`
- In general, we configure any other queue, apart from the `ECGQueue` queue, to be managed by the user admin

The advantages of such granularity in permission allows to fine-tune what action the client can perform on a specific queue/topic. So even if at runtime we connect to a queue without authorization, we are sure that the messages will not be inserted.

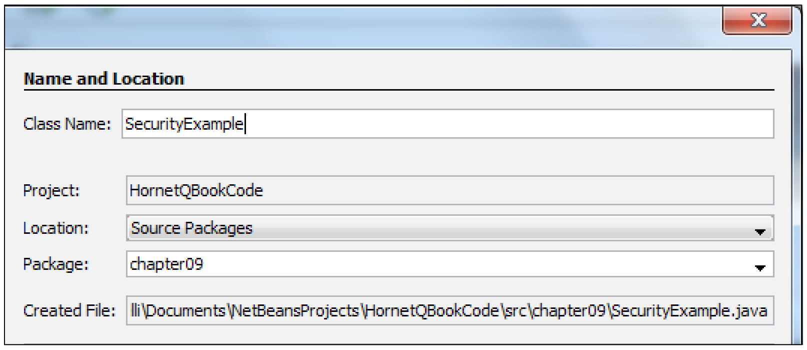
Testing our permissions

Now that we have set up our permissions, it is time to move to the code to understand what happens both from the consumer and producer layer when we try to use the resources involved in our previous settings.

Only to detail what we are going to do, we will force our code to perform the following actions:

- Connecting to a generic queue in an anonymous way to see the generated unauthorized exception
- Using the `ecgproducer` user to connect to `EGCQueue` and see that a message can be produced on that queue with that user
- Trying to consume from the `ECGQueue` queue with the `ecgproducer` user and see an error message
- Consuming correctly from the `ECGQueue` queue using the `ecgconsumer` user

To code our example, open your favorite IDE and create a new Java class with a main method called `SecurityExample` within the `chapter09` package, as shown in the following screenshot:



Name and Location

Class Name: SecurityExample

Project: HornetQBookCode

Location: Source Packages

Package: chapter09

Created File: lli\Documents\NetBeansProjects\HornetQBookCode\src\chapter09\SecurityExample.java

Now, as usual, let us proceed with the main connection that will share three sessions for three different users:

```
java.util.Properties p = new java.util.Properties();

p.put(javax.naming.Context.INITIAL_CONTEXT_FACTORY,
"org.jnp.interfaces.NamingContextFactory");

p.put(javax.naming.Context.URL_PKG_PREFIXES,
"org.jboss.naming:org.jnp.interfaces");
```

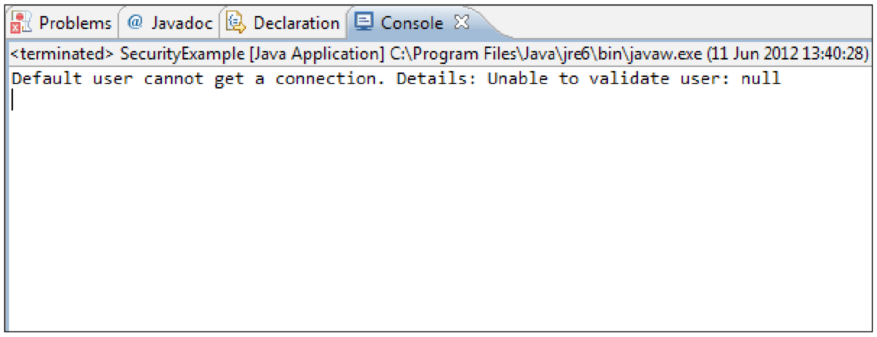
```
p.put(javax.naming.Context.PROVIDER_URL, "jnp://localhost:1099");

initialContext = new javax.naming.InitialContext(p);

Queue queue = (Queue)initialContext.lookup("/queue/ECGQueue");
```

Now we will try to connect to the `ECGQueue` queue without providing any username or password couple, so as to force the HornetQ session to be connected with the default user that is not allowed to do such an action.

We start our code with the following:

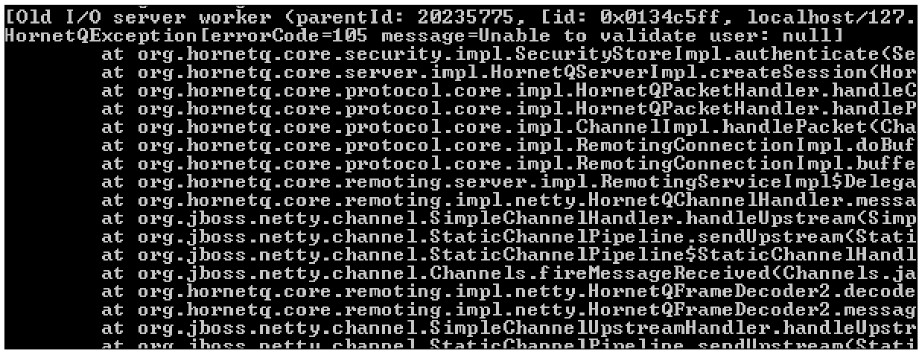
```
ConnectionFactory cf = (ConnectionFactory)
  initialContext.lookup("/ConnectionFactory");

try
{
  cf.createConnection();
}
catch (JMSSecurityException e)
{
  System.out.println("Default user cannot get a connection.
    Details: " + e.getMessage());
}
```

In this case we try to create a connection using the `ConnectionFactory` object (highlighted in the previous code snippet), but in this case a specific user is not provided, so a `JMSSecurityException` is raised, as shown in the following screenshot:



```
<terminated> SecurityExample [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (11 Jun 2012 13:40:28)
Default user cannot get a connection. Details: Unable to validate user: null
```

As expected, we cannot establish a connection without providing a user. The same error is also logged at a system level, as you can see from the console output (Windows in our case):

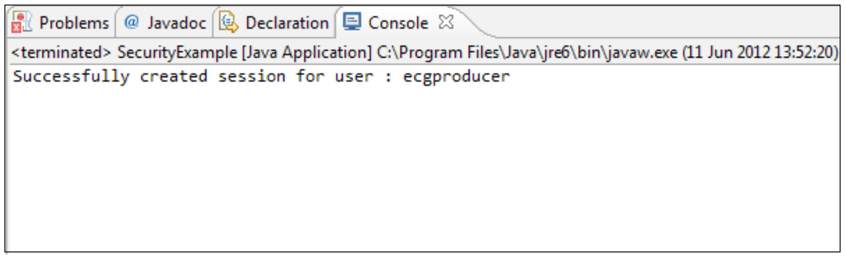


```
[Old I/O server worker (parentId: 20235775, [id: 0x0134c5ff, localhost/127.
HornetQException[errorCode=105 message=Unable to validate user: null]
        at org.hornetq.core.security.impl.SecurityStoreImpl.authenticate(Se
        at org.hornetq.core.server.impl.HornetQServerImpl.createSession(Hor
        at org.hornetq.core.protocol.core.impl.HornetQPacketHandler.handleC
        at org.hornetq.core.protocol.core.impl.HornetQPacketHandler.handleP
        at org.hornetq.core.protocol.core.impl.ChannelImpl.handlePacket(Cha
        at org.hornetq.core.protocol.core.impl.RemotingConnectionImpl.doBuf
        at org.hornetq.core.protocol.core.impl.RemotingConnectionImpl.buffe
        at org.hornetq.core.remoting.server.impl.RemotingServiceImpl$Delega
        at org.hornetq.core.remoting.impl.netty.HornetQChannelHandler.messa
        at org.jboss.netty.channel.SimpleChannelHandler.handleUpstream(Simp
        at org.jboss.netty.channel.StaticChannelPipeline.sendUpstream(Stati
        at org.jboss.netty.channel.StaticChannelPipeline$StaticChannelHandl
        at org.jboss.netty.channel.Channels.fireMessageReceived(Channels.ja
        at org.hornetq.core.remoting.impl.netty.HornetQFrameDecoder2.decode
        at org.hornetq.core.remoting.impl.netty.HornetQFrameDecoder2.messag
        at org.jboss.netty.channel.SimpleChannelUpstreamHandler.handleUpstr
        at org.jboss.netty.channel.StaticChannelPipeline.sendUpstream(Stati
```

As far as we know the role producer can connect and send a message to the `ECGQueue` queue, we try to send a message on that queue. The code is pretty much the same as the previous one, but notice that in this case we provide the credential to access.

```
try
{
  cf.createConnection("ecgproducer", "ecgproducer01");
  System.out.println("Successfully created session for user :
    ecgproducer");
}
catch (JMSSecurityException e)
{
  System.out.println("Default user cannot get a connection.
    Details: " + e.getMessage());
}
```

In this case, as highlighted in the code, we provide the login information of the `ecgproducer` user. So in this case we are able to successfully connect to the queue, as the console message suggests:



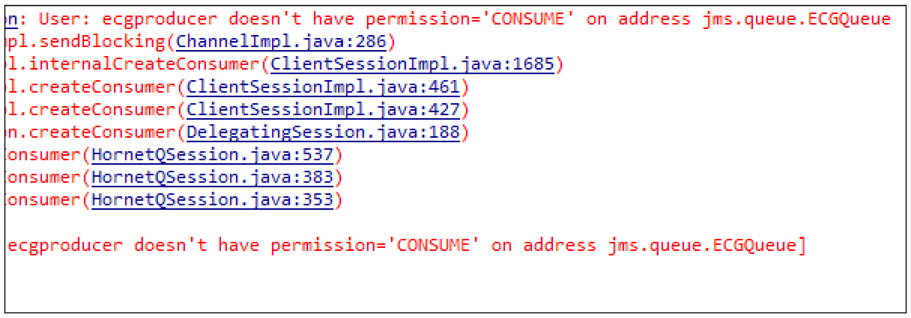
```
<terminated> SecurityExample [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (11 Jun 2012 13:52:20)
Successfully created session for user : ecgproducer
```

These were simply checks. The interesting part comes now while we try to send and consume a message on the connection we have just created.

The code is pretty much the same as we used in the previous chapters.

```
Session session = connection.createSession(false,
  Session.AUTO_ACKNOWLEDGE);
MessageProducer producer = session.createProducer(queue);
MessageConsumer consumer = session.createConsumer(queue);

TextMessage msg = session.createTextMessage("ecg_message");
producer.send(msg);
System.out.println("message sent on "  + (new java.util.Date()) );

TextMessage receivedMsg = (TextMessage)consumer.receive(2000);
```

But now, if we run the example, we will have a very different output like the one shown in the following screenshot:



```
n: User: ecgproducer doesn't have permission='CONSUME' on address jms.queue.ECGQueue
pl.sendBlocking(ChannelImpl.java:286)
l.internalCreateConsumer(ClientSessionImpl.java:1685)
l.createConsumer(ClientSessionImpl.java:461)
l.createConsumer(ClientSessionImpl.java:427)
n.createConsumer(DelegatingSession.java:188)
onsumer(HornetQSession.java:537)
onsumer(HornetQSession.java:383)
onsumer(HornetQSession.java:353)

ecgproducer doesn't have permission='CONSUME' on address jms.queue.ECGQueue]
```

The output is pretty self-explanatory, but we will comment on only one thing. As expected using the same connection that was created, but in this case, providing the credentials for the user with role producer, we have another exception raised. The exception is raised in the following line of code:

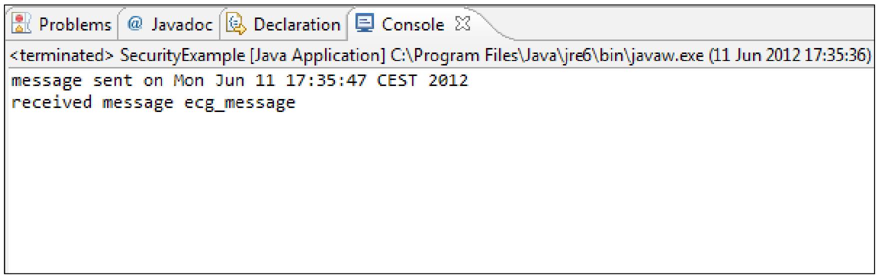
```
MessageConsumer consumer = session.createConsumer(queue);
```

To correctly consume the messages on the `ECGQueue` queue, we need to create another connection providing the login information of the `ecgconsumer` user. To do this we need the following block of code:

```
Connection producerconnection = cf.createConnection("ecgconsumer",
  "ecgconsumer01");
producerconnection.start();

Session producersession = producerconnection.createSession(false,
  Session.AUTO_ACKNOWLEDGE);
```

```
producer = producersession.createProducer(queue);

MessageConsumer consumer = producersession.createConsumer(queue);
TextMessage receivedMsg = (TextMessage)consumer.receive(2000);

System.out.println("received message " + receivedMsg.getText());
```

So on the same `ConnectionFactory cf` object we create another connection providing a couple of different credentials, the one of the `ecgconsumer` user. In this case the permissions are granted for the user, so the output will be the following screenshot:



Problems | Javadoc | Declaration | Console

```
<terminated> SecurityExample [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (11 Jun 2012 17:35:36)
message sent on Mon Jun 11 17:35:47 CEST 2012
received message ecg_message
```

This certifies that now with the connection and session instantiated, the `ecgconsumer` user is able to consume the message on the `ECGQueue` queue.

> We invite you to test different scenarios and also to try to do some management operations on the queues we provide.

We also remind you that everything we saw in this example could be easily moved to the `topic` object instead of queues, by remembering that the wildcard of the `security-setting` tag works both from queues and topics. The difference is that when you need to set a particular permission on a topic, the name should start with the prefix `jms.topic` for topics and the topic tag in the `hornetq-jms.xml` file should be written accordingly. So if you need to set a security on the topic named `example`, in the `HornetQ-configuration.xml` file you should have something like the following:

```
<security-setting match="jms.topic.example">
```

While in the `hornetq-jms.xml` file you should have a tag like the following:

```
<topic name="example">
<entry name="/topic/example"/>
</topic>
```

Cluster security

As we saw in *Chapter 6, Clustering with HornetQ*, when we configure a cluster, every node needs to communicate with the other ones using a special user that is configured in the `HornetQ-configuration.xml` file. When creating a cluster, every node should have written in the configuration file the value changed for this special user. The tags to do this are:

```
<management-cluster-
  user>HORNETQ.MANAGEMENT.ADMIN.USER</management-cluster-user>
<management-cluster-password>password</management-cluster-
  password>
```

In fact, every time you start a single node, both in clustered and non-clustered environments, you should see that even HornetQ warns you to change the password values as you can see from the extracted text in the following log:

```
[main] 23:36:00,036 WARNING
  [org.hornetq.core.server.impl.HornetQServerImpl]  Security risk!
  It has been detected that the cluster admin user and password
  have not been changed from the installation default. Please see
  the HornetQ user guide, cluster chapter, for instructions on how
  to do this.
```

So before creating a cluster in a production environment, change every node's default password.

Now that we have seen how to set the permission on resources, we are ready to move to the next step that is securing the messages themselves using SSL.

Configuring the SSL connection

Apart from the login/password mechanism, an additional way to strengthen the security on the producer/consumer layer is to communicate messages using the **Secure Socket Layer protocol** (**SSL**), which is a way to exchange data in encrypted format.

Generally speaking, when a connection happens inside an untrusted network it is always good practice to encrypt the content. HornetQ provides Netty to communicate with the producers and the consumers. However, all the data that we send to a Netty acceptor are unencrypted, so we need to move to a more secure approach to assure security in communications. This approach could increase the difficulty for a **Man In the Middle** (**MIM**) attack to succeed. Just to explain briefly, a Man In the Middle attack consists of intercepting all the traffic between the client consumer and the HornetQ server by inserting a PC in the middle where all the traffic can be dumped. If we do not use an SSL connection, all the messages are exchanged in plain text so they are visible in the dump.

If we are using JMS to exchange messages, we do not have to do anything. The code that we create does not need any change, but we need to properly configure the HornetQ instance for it to run correctly. Being that the client is the one who opens the connection and sends messages, we need to store a key for SSL to work properly.

Going through the previous example, we will create an SSL acceptor for our producer to send messages to our HornetQ.

In this case we do not need to change the code we used, because the SSL acceptor for JMS works in a transparent way from the code's perspective.

Setting up an SSL connection

To create an SSL connection we need to use a configuration file and also create two keys, one for the client and one for the server, based on a certificate.

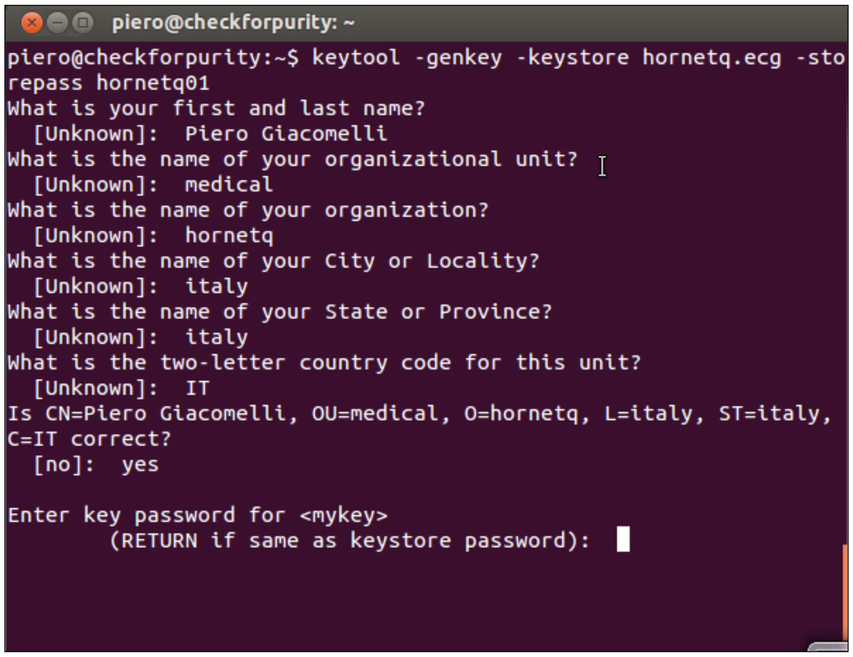
To do this, we need to use the keytool utility. Keytool is a Java utility present in the JDK, that is able to manage from the command line the Java Key Store, which is the repository where all the keys are stored. The JDK is nearly OS agnostic, so we will provide the example using our Ubuntu test environment, but the same set of commands can be used on a Windows prompt. For further information we suggest you to take a look at the official documentation (`http://docs.oracle.com/javase/6/docs/api/java/security/KeyStore.html`).

For configuring HornetQ to work with the keystore of Java, you only need the files, so it is not necessary to create everything on the server. But to avoid any issues we suggest using the JDK inside the machine where HornetQ is running.

To start with our configuration the first step is to create the keystore, and the key to do this is to open a terminal prompt and type in the following command:

```
keytool -genkey -keystore hornetq.ecg -storepass hornetq01
```

So in this case we create a key using the keystore `Hornetq.keystore` and password `hornetq01`. The terminal will prompt some questions as shown in the following screenshot:



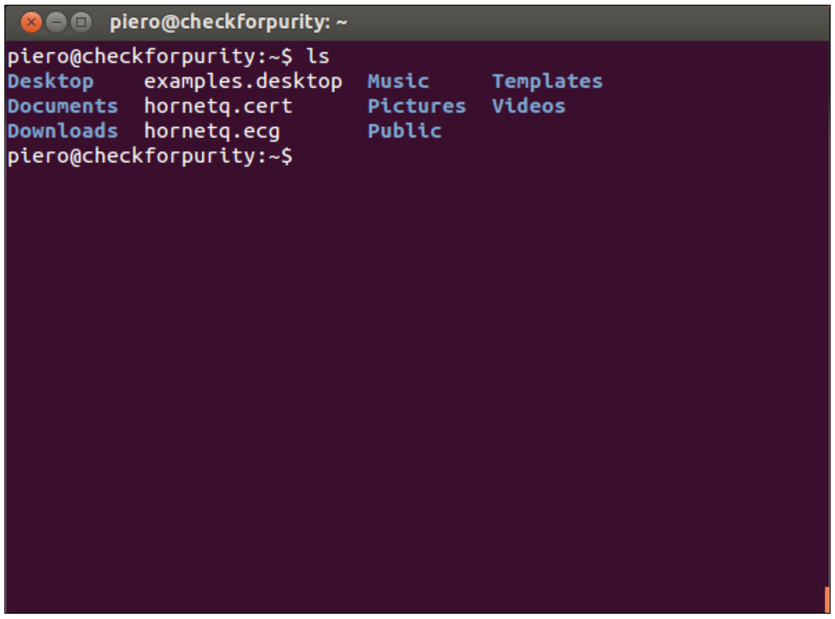
```
piero@checkforpurity:~$ keytool -genkey -keystore hornetq.ecg -sto
repass hornetq01
What is your first and last name?
  [Unknown]:  Piero Giacomelli
What is the name of your organizational unit?
  [Unknown]:  medical
What is the name of your organization?
  [Unknown]:  hornetq
What is the name of your City or Locality?
  [Unknown]:  italy
What is the name of your State or Province?
  [Unknown]:  italy
What is the two-letter country code for this unit?
  [Unknown]:  IT
Is CN=Piero Giacomelli, OU=medical, O=hornetq, L=italy, ST=italy,
C=IT correct?
  [no]:  yes

Enter key password for <mykey>
        (RETURN if same as keystore password):
```

Remember to answer `yes` when you have checked the information, provided it is correct, and to enter `RETURN` to use the same password. Now to test that everything was correct, you should see a new file called `hornetq.ecg` that will be one of the two files that we need to move to the `hornetq config` folder and that will create the server's trusted key. The file is directly generated on the same folder where the command has been launched.

Now that we have created our key, we need to export the certificate to a file using the following command:

```
keytool -export -keystore hornetq.ecg -file hornetq.cer
```

The prompt will ask for your previous password and will create a file `hornet.cert`, as you can see in the following screenshot:

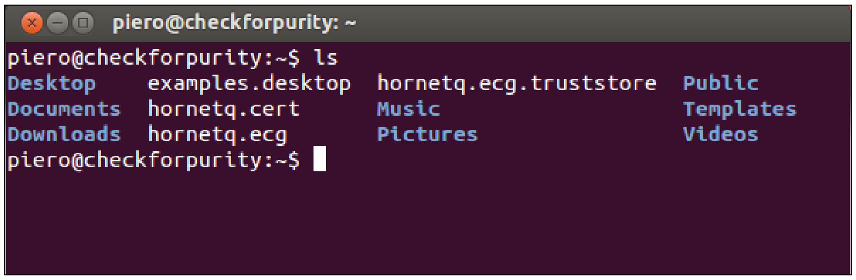


```
piero@checkforpurity:~$ ls
Desktop    examples.desktop  Music     Templates
Documents  hornetq.cert      Pictures  Videos
Downloads  hornetq.ecg       Public
piero@checkforpurity:~$
```

Now as a final step, we need to create a second file that will be our client's key. The command to do this is as follows:

```
keytool -import -file hornetq.cert -keystore hornetq.ecg.truststore -
  storepass hornetq01
```

The prompt will ask if the certificate should be trusted, and if we reply yes, we should now have the following screenshot and another file created:



```
piero@checkforpurity:~$ ls
Desktop    examples.desktop  hornetq.ecg.truststore  Public
Documents  hornetq.cert      Music                   Templates
Downloads  hornetq.ecg       Pictures                Videos
piero@checkforpurity:~$
```

Now that we have created the key, we are ready to configure HornetQ. To do this, first grab the two files created; `HornetQ.ecg` and `HornetQ.ecg.truststore`, and move them to the `HORNETQ_ROOT\config\standalone\non-clustered` folder.

Now to configure HornetQ so that we use an SSL connection, we need to configure acceptors and connectors to force using SSL and the key we have created. To do this go into the `HORNETQ_ROOT\config\standalone\non-clustered` folder and open the `HornetQ-configuration.xml file`. You should see the following tags:

```
<connectors>
<connector name="netty">
  <factory-  class>org.hornetq.core.remoting.impl.netty.
NettyConnectorFactory
</factory-class>
  <param key="host"  value="${hornetq.remoting.netty.
host:localhost}"/>
  <param key="port"  value="${hornetq.remoting.netty.port:5445}"/>
</connector>

<connector name="netty-throughput">
  <factory-class>org.hornetq.core.remoting.impl.netty.
NettyConnectorFactory</factory-class>
  <param key="host"  value="${hornetq.remoting.netty.
host:localhost}"/>
  <param key="port"  value="${hornetq.remoting.netty.batch.
port:5455}"/>
  <param key="batch-delay" value="50"/>
</connector>
</connectors>

<acceptors>
<acceptor name="netty">
  <factory-class>org.hornetq.core.remoting.impl.netty.
NettyAcceptorFactory</factory-class>
  <param key="host"  value="${hornetq.remoting.netty.
host:localhost}"/>
  <param key="port"  value="${hornetq.remoting.netty.port:5445}"/>
</acceptor>

<acceptor name="netty-throughput">
  <factory-class>org.hornetq.core.remoting.impl.netty.
NettyAcceptorFactory</factory-class>
  <param key="host"  value="${hornetq.remoting.netty.
host:localhost}"/>
  <param key="port"  value="${hornetq.remoting.netty.batch.
port:5455}"/>
  <param key="batch-delay" value="50"/>
  <param key="direct-deliver" value="false"/>
</acceptor>
</acceptors>
```

To force HornetQ using the SSL connector and acceptor, we need to remove them and substitute them with the following ones:

```
<connectors>
<connector name="netty-ssl-connector">
  <factory-class>org.hornetq.core.remoting.impl.netty.
NettyConnectorFactory</factory-class>
  <param key="host" value="localhost"/>
  <param key="port" value="5500"/>
  <param key="ssl-enabled" value="true"/>
  <param key="key-store-path" value="/hornetq/config/standalone/non-
clustered/"/>
  <param key="key-store-password" value="hornetq01"/>
</connector>
</connectors>

<acceptors>
<acceptor name="netty-ssl-acceptor">
  <factory-class>org.hornetq.core.remoting.impl.netty.
NettyAcceptorFactory</factory-class>
  <param key="host" value="localhost"/>
  <param key="port" value="5500"/>
  <param key="ssl-enabled" value="true"/>
  <param key="key-store-path" value="hornetq.ecg"/>
  <param key="key-store-password" value="hornetq01"/>
  <param key="trust-store-path" value="hornetq.ecg.truststore"/>
  <param key="trust-store-password" value="hornetq01"/>
</acceptor>
</acceptors>
```

The highlighted tags in the previous code snippet configure the following facts in HornetQ:

- Using SSL: `<param key="ssl-enabled" value="true"/>`
- The path where the key files are stored (absolute): `<param key="key-store-path" value="/hornetq/config/standalone/non-clustered/"/>`
- The password to be used: `<param key="key-store-password" value="hornetq01"/>`
- The files to be used: `<param key="key-store-path" value="hornetq.ecg"/>` and `<param key="trust-store-path" value="hornetq.ecg.truststore"/>`

This was our last step. If we now run the server, we get the connections from the producer consumer layer, which is done using SSL and the certificate keys we provided.

From the coder's point of view nothing changed because all the trusted communication and the encryption layer is done by the Netty connector without explicit programming.

This way of operating lifts up the level of the security, both from the producer and the consumer layer.

If we are in a cluster environment, the same configurations could be reported to every node of the cluster itself so as to allow every node to communicate between the others in a more secure way.

For the willing reader, we would also suggest an additional security focus setting that we can provide that is not so difficult to be coded, but that requires some object-oriented thoughts.

The idea is also to encrypt messages so that only encrypted messages are stored on the queue. To do this we suggest you extend the `TextMessage` object provided by HornetQ so that we can add two new methods `encrypt` and `decrypt`. The main idea is to encrypt the body of the message before sending it to a queue in the usual way.

One code idea would be to create a class like the following one:

```
public class myTextMessage implements javax.jms.TextMessage {
  public void Encrypt(){
    //stuff here
  }
  protected String Decrypt(){
    //stuff here
  }
}
```

Now, before sending the message you could use the `encrypt` method to encrypt it and when consuming it you could decrypt it.

This final suggestion is only optional to this chapter where we learned:

- How to create users and roles in HornetQ
- How to assign permissions to queues/topics to those users
- How to connect clients using such users
- How to create an SSL connection between the clients and the HornetQ server

HornetQ can be configured to be used with other security managers besides the one provided by default. For example, when HornetQ runs inside a JBoss environment, the security manager used is the one provided by JBoss.

We won't cover this topic here. As for other settings, the different security managers can be set up changing the `hornet-beans.xml` file and declaring what class implementing the `HornetQSecurityManager` should be used. We invite the reader to refer to the HornetQ documentation (`http://docs.jboss.org/hornetq/2.2.5.Final/user-manual/en/html_single/index.html#change-security-manager`) for some examples.

Summary

In this chapter we have seen how to create roles and grant permission on queues/topics to defined users. We have also seen how to configure HornetQ to deal with an SSL connection. We are now ready to move to the next chapter where we will see how HornetQ talks to its big brother JBOSS.

10
HornetQ in JBoss Environment

So far we have seen the many features of HornetQ; we have touched upon standalone, cluster configuration, management, security, and so on. But all the example codes that we had were somehow related to an enterprise environment. Because HornetQ is basically a set of integrated libraries, it is possible to use it in a web environment. As we have seen in *Chapter 2*, *Setting Up HornetQ*, HornetQ comes as the default JMS message middleware within the JBoss environment. This chapter will be dedicated to describing how to use HornetQ within JBoss. We will move from the basic configurations, to the web interface, to the coding of an example. The main topics that will be covered are as follows:

- Configuring HornetQ inside JBoss
- Coding an HornetQ **Message Driven Bean** (**MDB**) to be used in a servlet

Configuring and installing JBoss

In this section we will give you a quick introduction on how to run a JBoss 7 correctly. This is the latest version with HornetQ as the default messaging framework.

We will not give a complete installation procedure, because, like HornetQ, JBoss is also very rich in configuration possibilities. For a very detailed introduction to this subject, we refer you to the book *JBoss AS 7 Configuration, Deployment and Administration*, *Francesco Marchioni*, *Packt Publishing*. Anyway, we will detail the installation procedure on an Ubuntu server, so that you can get familiar with the standard procedures.

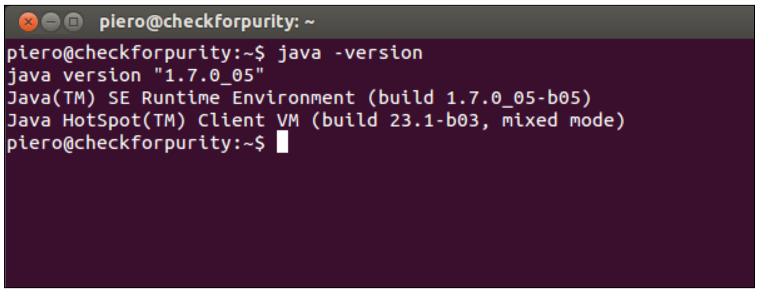
Before starting we need to remember, as we did in *Chapter 2*, *Setting Up HornetQ*, that Ubuntu comes without the default JDK installed, so you need to redo the procedure we adopted in *Chapter 2*, *Setting Up HornetQ* (installing the JVM/JDK) before proceeding.

To check if everything is OK, use the following steps:

1. Open a terminal.
2. Enter the following command:

   ```
   /java -version
   ```

You should get the following output:



```
piero@checkforpurity: ~
piero@checkforpurity:~$ java -version
java version "1.7.0_05"
Java(TM) SE Runtime Environment (build 1.7.0_05-b05)
Java HotSpot(TM) Client VM (build 23.1-b03, mixed mode)
piero@checkforpurity:~$
```

Now that you are ready, we need to download the latest version of the JBoss binaries, so we need to go to the default download website located at `http://www.jboss.org/jbossas/downloads/` and download the 7.1.1.Final release version using the `tar.gz` archive (or the ZIP archive if you are in a Windows environment).

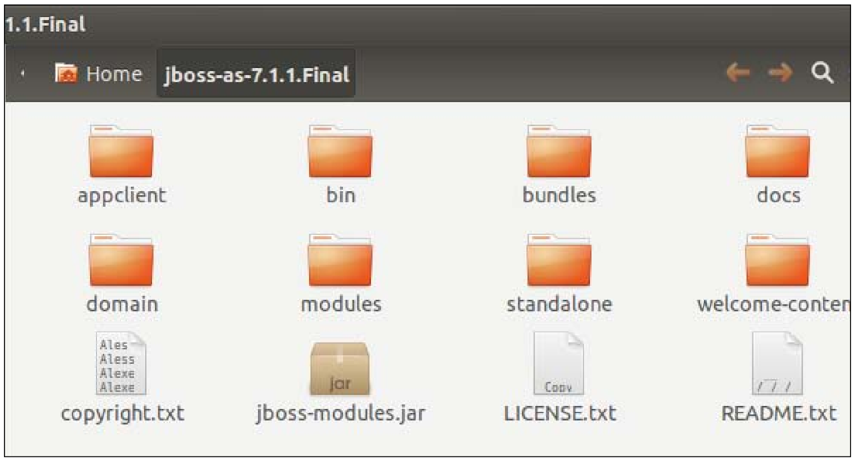
If you are using a text-only environment, you could download directly using the following command:

```
wget http://download.jboss.org/jbossas/7.1/jboss-as-7.1.1.Final/jboss-as-
7.1.1.Final.tar.gz
```

While waiting for the download to complete, remember that as the version changes, the release could change. Once the download is finished, unpack the archive by using the following command:

```
tar -xvzf jboss-as-7.1.1.Final.tar.gz
```

So to have a folder with all the JBoss server, we have a `JBOSS_ROOT` folder as the one shown in the following screenshot (in our case **/home/~ /jboss-as-7.1.1.Final**):



JBoss mimics the same folder structure as HornetQ, so all the scripts for running the server and other additional resources are in the `JBOSS_ROOT\bin` folder, while the XML for the configuration is located inside the `JBOSS_ROOT\standalone\configuration` folder.

JBoss uses a single file for storing everything and by default this file is the `standalone.xml` file. This is located in the `JBOSS_ROOT\standalone\configuration` folder. However, this file does not have the extension for the messaging system, that is HornetQ.

So as to allow JBoss starting with an HornetQ extension to be enabled, you should use the `standalone-full.xml` configuration file that is contained in the same folder. To use this file for starting JBoss, we should explicitly tell JBoss to use this file.

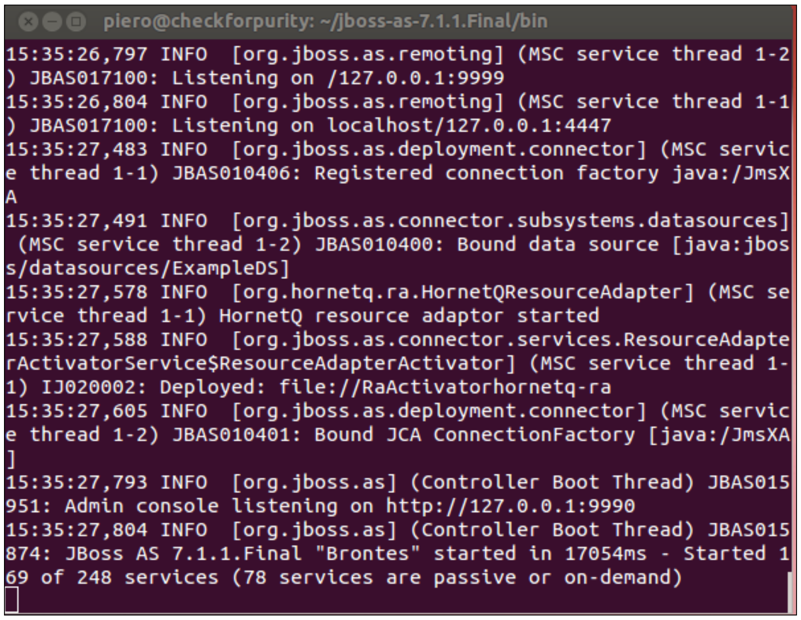
The command to be executed is as follows:

```
./standalone.sh --server-config=standalone-full.xml
```

This command should be launched within the `JBOSS_ROOT\bin` folder. The same procedure should be used in a Windows environment, except that the command will be as follows:

```
./standalone.bat -server-config=standalone-full.xml
```

By following these simple steps, the result should be the following terminal window:



```
piero@checkforpurity: ~/jboss-as-7.1.1.Final/bin
15:35:26,797 INFO  [org.jboss.as.remoting] (MSC service thread 1-2
) JBAS017100: Listening on /127.0.0.1:9999
15:35:26,804 INFO  [org.jboss.as.remoting] (MSC service thread 1-1
) JBAS017100: Listening on localhost/127.0.0.1:4447
15:35:27,483 INFO  [org.jboss.as.deployment.connector] (MSC servic
e thread 1-1) JBAS010406: Registered connection factory java:/JmsX
A
15:35:27,491 INFO  [org.jboss.as.connector.subsystems.datasources]
 (MSC service thread 1-2) JBAS010400: Bound data source [java:jbos
s/datasources/ExampleDS]
15:35:27,578 INFO  [org.hornetq.ra.HornetQResourceAdapter] (MSC se
rvice thread 1-1) HornetQ resource adaptor started
15:35:27,588 INFO  [org.jboss.as.connector.services.ResourceAdapte
rActivatorService$ResourceAdapterActivator] (MSC service thread 1-
1) IJ020002: Deployed: file://RaActivatorhornetq-ra
15:35:27,605 INFO  [org.jboss.as.deployment.connector] (MSC servic
e thread 1-2) JBAS010401: Bound JCA ConnectionFactory [java:/JmsXA
]
15:35:27,793 INFO  [org.jboss.as] (Controller Boot Thread) JBAS015
951: Admin console listening on http://127.0.0.1:9990
15:35:27,804 INFO  [org.jboss.as] (Controller Boot Thread) JBAS015
874: JBoss AS 7.1.1.Final "Brontes" started in 17054ms - Started 1
69 of 248 services (78 services are passive or on-demand)
```

If JBoss uses the full XML configuration, there are a lot of services that are started. To see that HornetQ was started correctly and that some queues are deployed, you could see the log files that are located in the `JBOSS_ROOT\standalone\log` folder, in our case the `server.log` file. If we open it with a text editor we should find the following line:

```
15:33:19,318 INFO  [org.hornetq.core.server.impl.HornetQServerImpl]
(MSC service thread 1-1) live server is starting with configuration
HornetQ Configuration
```

And below this line the following ones:

```
15:35:26,686 INFO  [org.jboss.as.messaging] (MSC service thread 1-1)
JBAS011601: Bound messaging object to jndi name java:/queue/test
15:35:26,698 INFO  [org.jboss.as.messaging] (MSC service thread 1-1)
JBAS011601: Bound messaging object to jndi name java:jboss/exported/
jms/queue/test
15:35:26,718 INFO  [org.jboss.as.messaging] (MSC service ths
```

The previous code snippet is telling us that HornetQ has been started as the default messaging extension for JBoss and that the JMS test queue was deployed successfully.

Configuring HornetQ within JBoss

As we have seen, we were able to run JBoss within HornetQ in a standalone configuration without great effort. However, we need to understand what happens, so first we need to move to the configuration folder we used. In our case it was the `standalone-full.xml` file. We will now open the file and analyze the main sections.

JBoss uses an extension mechanism so various modules can be enabled or disabled according to the configuration needed. In our case, by opening the file we could find the following section:

```
<extensions>
  <extension module="org.jboss.as.messaging"/>
```

The previous code snippet tells JBoss that the messaging module should be enabled and started. The `org.jboss.as.messaging` attribute value corresponds to another tag section below in the same file, precisely the following code snippet:

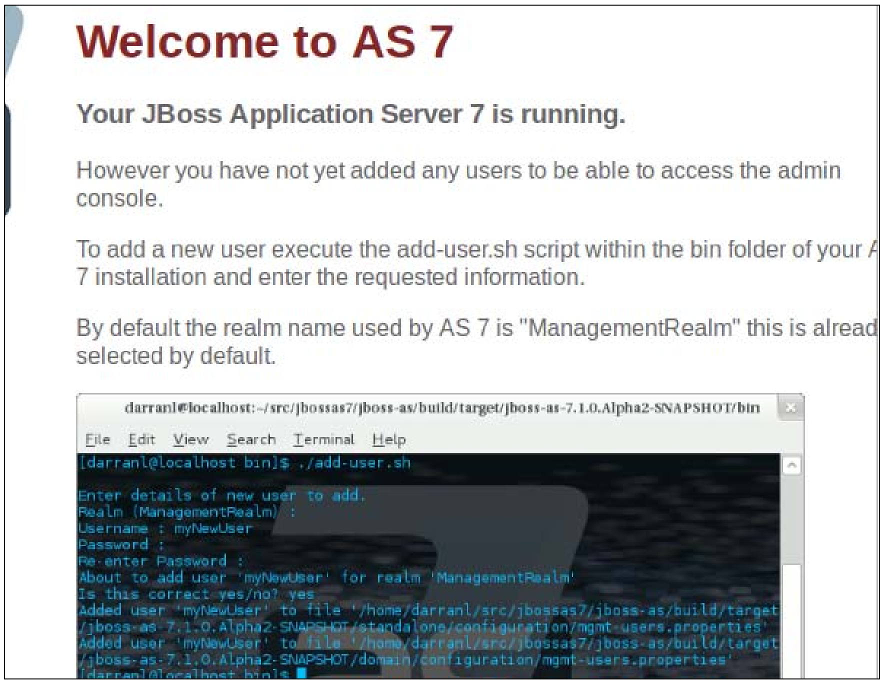
```
<subsystem xmlns="urn:jboss:domain:messaging:1.1">
  <hornetq-server>
    <persistence-enabled>true</persistence-enabled>
    <journal-file-size>102400</journal-file-size>
    <journal-min-files>2</journal-min-files>
    <connectors>

    </connectors>
    <acceptors>
```

We do not list all the tags but only some of them to let you understand that they are the same as the standard HornetQ configurations.

As you can see, the file contains some of the tags of the usual `HornetQ-configuration.xml` file of the HornetQ server, and some of the tags of the `HornetQ-jms.xml` file that contains the queue definition. You should recognize the main tags such as the `jms-queue` tag and the others.

Obviously one way to use HornetQ within JBoss is to configure the JBoss configuration file in order to enable HornetQ. Another interesting way is to use the JBoss admin console that, if enabled, would respond at the URL `http://127.0.0.1:9990`.

Before trying to connect, we need to add a user with the permission to use the management console, otherwise the first time connection will give the following page:



Welcome to AS 7

Your JBoss Application Server 7 is running.

However you have not yet added any users to be able to access the admin console.

To add a new user execute the add-user.sh script within the bin folder of your A
7 installation and enter the requested information.

By default the realm name used by AS 7 is "ManagementRealm" this is alread
selected by default.

darranl@localhost:~/src/jbossas7/jboss-as/build/target/jboss-as-7.1.0.Alpha2-SNAPSHOT/bin
File Edit View Search Terminal Help

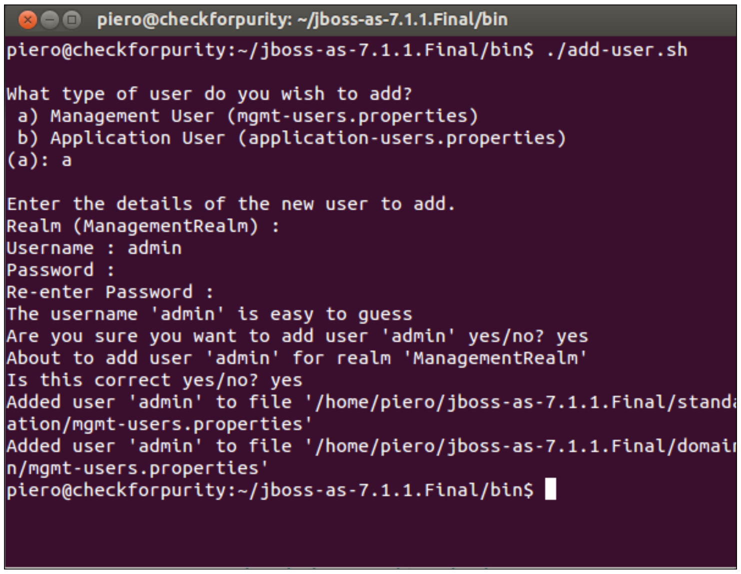
```
[darranl@localhost bin]$ ./add-user.sh

Enter details of new user to add.
Realm (ManagementRealm) :
Username : myNewUser
Password :
Re-enter Password :
About to add user 'myNewUser' for realm 'ManagementRealm'
Is this correct yes/no? yes
Added user 'myNewUser' to file '/home/darranl/src/jbossas7/jboss-as/build/target
/jboss-as-7.1.0.Alpha2-SNAPSHOT/standalone/configuration/mgmt-users.properties'
Added user 'myNewUser' to file '/home/darranl/src/jbossas7/jboss-as/build/target
/jboss-as-7.1.0.Alpha2-SNAPSHOT/domain/configuration/mgmt-users.properties'
[darranl@localhost bin]$
```

So as shown in the previous screenshot, to add a user you need to open a terminal and run a script that will add to the user allowed, the necessary permissions to use the management console.

So before connecting with the web admin console, open a terminal on the `JBOSS_ROOT\bin` folder and enter the following command:

```
./add-user.sh (under Linux)
add-user.bat (under Windows)
```

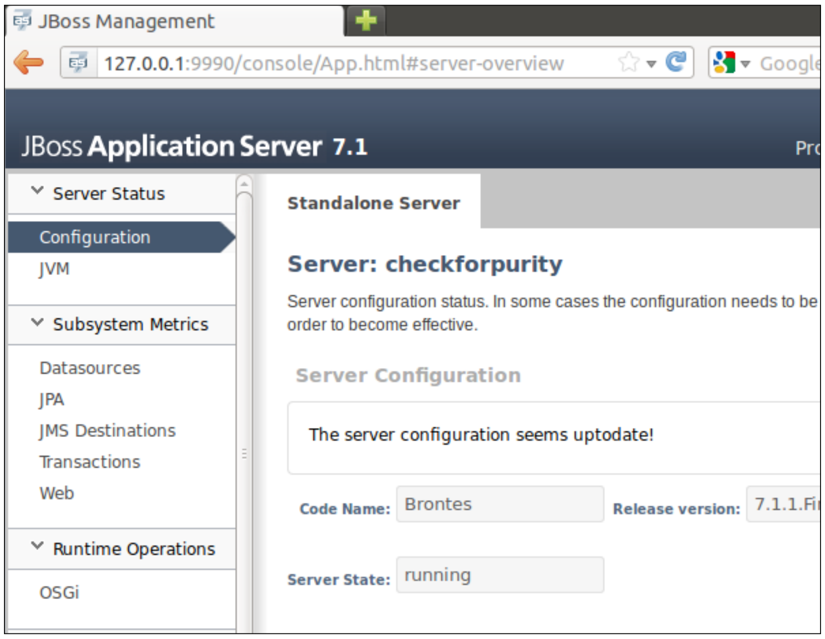
The system will reply by asking the grants and some questions about the user, such as the username and the password. The result should be the following screenshot:



```
piero@checkforpurity: ~/jboss-as-7.1.1.Final/bin
piero@checkforpurity:~/jboss-as-7.1.1.Final/bin$ ./add-user.sh

What type of user do you wish to add?
 a) Management User (mgmt-users.properties)
 b) Application User (application-users.properties)
(a): a

Enter the details of the new user to add.
Realm (ManagementRealm) :
Username : admin
Password :
Re-enter Password :
The username 'admin' is easy to guess
Are you sure you want to add user 'admin' yes/no? yes
About to add user 'admin' for realm 'ManagementRealm'
Is this correct yes/no? yes
Added user 'admin' to file '/home/piero/jboss-as-7.1.1.Final/standa
ation/mgmt-users.properties'
Added user 'admin' to file '/home/piero/jboss-as-7.1.1.Final/domai
n/mgmt-users.properties'
piero@checkforpurity:~/jboss-as-7.1.1.Final/bin$
```

Do not forget to reply to the first question that you want a **Management User.**

Now by connecting to `http://127.0.0.1:9990`, JBoss should ask you the username/password combination to access the page. Once entered we will see the following page:



Server: checkforpurity

Server configuration status. In some cases the configuration needs to be
order to become effective.

Server Configuration

The server configuration seems uptodate!

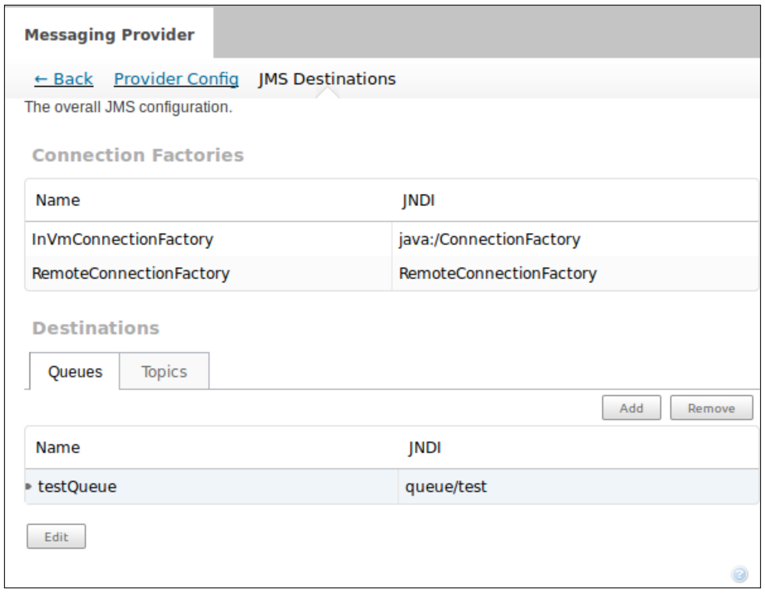
Code Name: Brontes
Release version: 7.1.1.Fi
Server State: running

Creating and managing queues/topics

Now to create and manage the queues/topics, we need to use the following steps:

1. On the upper-right corner, click on the **Profile** link.
2. On the left menu expand the **Messaging** menu item.
3. Click on the **Message Provider** link.
4. Click on the center text **JMS Destinations**.

Now we are on the page that provides a set of tools for creating, managing, and assigning roles and binds to queues. This page is shown in the following screenshot:



The overall JMS configuration.

Connection Factories

Name	JNDI
InVmConnectionFactory	java:/ConnectionFactory
RemoteConnectionFactory	RemoteConnectionFactory

Destinations

Name	JNDI
testQueue	queue/test

As for HornetQ, the queues are deployed at runtime without any server restart. For the examples that will follow, we invite you to create the `ECGQueue` queue, by using the **Add** button, and see what happens in the configuration files.

HornetQ and J2EE environment

HornetQ provides a **Java Connector Architecture** (**JCA**) for connecting itself with **J2EE** environments, that support not only the JBoss environment, but this architecture as well.

The JCA adapter controls the inflow of messages to Message Driven Beans (MDB) and the outflow of messages sent from servlets and JSP.

A Message Driven Bean is a logic component that is attached to a queue or a topic and is automatically activated when new messages come in the queue; so basically it acts like a consumer that is integrated in a J2EE environment.

In our example we will create a Message Driven Bean that will print in the standard output the text of a message that will be sent by invoking a servlet.

We will manually create and deploy the Message Driven Beans and the servlet in JBoss, because even if both Eclipse and NetBeans offer tools for deploying Message Driven Beans to JBoss, their configuration is out of the scope of this chapter. You can check out the online documentation for configuring JBoss. The links for the respective IDEs are as follows:

- `https://docs.jboss.org/author/display/AS7/Starting+JBoss+AS+from+Eclipse+with+JBoss+Tools` for Eclipse
- `https://community.jboss.org/thread/168237` for NetBeans

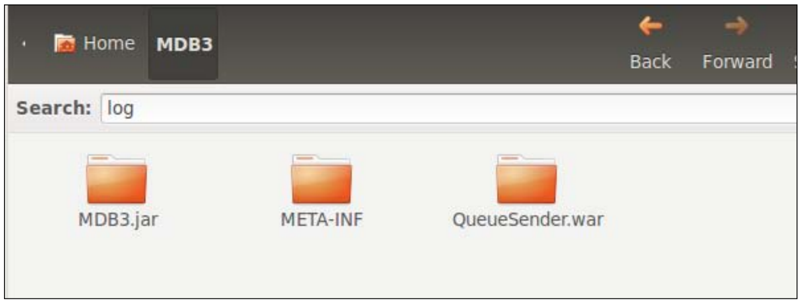
To execute everything, the steps we will cover are as follows:

1. Create a dedicated folder.
2. Create the MDB and compile it.
3. Create the configuration for deploying.
4. Create the servlet.
5. Deploy everything.
6. Test online.

So let us start with the folder creation on Linux. Enter the following commands:

```
mkdir MDB3
cd MDB3
mkdir MDB3.jar, META-INF, QueueSender.war
```

This will result in the following folder structure:



In the `MDB3.jar` folder, create the `MyMDB.java` file and copy the following code:

```
import javax.jms.Message;
import javax.jms.TextMessage;
import javax.jms.JMSException;
import javax.jms.MessageListener;
import javax.ejb.MessageDriven;
import javax.ejb.ActivationConfigProperty;

@MessageDriven(activationConfig =
        {
        @ActivationConfigProperty(propertyName="destinationType",
propertyValue="javax.jms.Queue"),
        @ActivationConfigProperty(propertyName="destination",
propertyValue="queue/test")
        })

public class MyMDB implements MessageListener
{
    public void onMessage(Message message)
    {
        TextMessage textMessage = (TextMessage) message;
        try
       {
            System.out.println("===> MyMDB Received: "+ textMessage.
getText());
        }
        catch (JMSException e)
        {
            e.printStackTrace();
        }
    }
}
```

The Message Driven Bean implements only the `onMessage` method. In our case, we will only print the message body as we consume it. As you can see, the queue destination value is `queue/test`, which is deployed by default using the `standalone-full.xml` configuration file.

You could however refer to any queue coded into the configuration file.

To correctly import the Message Driven Beans, add the `javaee-api-6.0.jar` JAR file, which is downloadable from the `http://download.java.net/maven/2/javax/javaee-api/6.0/javaee-api-6.0.jar` URL, to your classpath.

Once you have created the Message Driven Bean class file, we need to set up the connector within the servlet. To do this, we need to move from the `MDB3.jar` folder to the `META-INF` folder and create a file called `application.xml` within the following content:

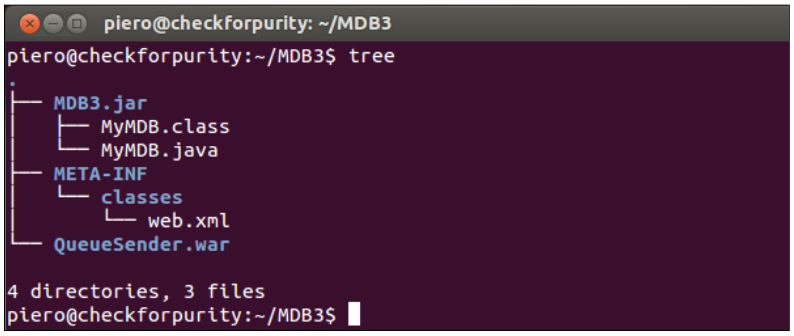
```
<?xml version="1.0" encoding="UTF-8"?>
<application version="5" xmlns="http://java.sun.com/xml/ns/javaee"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xsi:schemaLocation="http://java.sun.com/xml/ns/javaee

http://java.sun.com/xml/ns/javaee/application_5.xsd">

   <module>
       <ejb>MDB3.jar</ejb>
   </module>
   <module>
  <web>
     <web-uri>QueueSender.war</web-uri>
     <context-root>QueueSender</context-root>
  </web>
   </module>
</application>
```

So in the `MDB3.jar`, we have the basic definition files that will be needed for the WAR file. Do not forget that the name of the final folder should respect the one in the previous code.

Now move to `QueueSender.war` and create the following:

- A folder called `WEB-INF`
- Inside the `WEB-INF` folder, another folder called `classes`
- Inside the `classes` folder, a file called `web.xml`

As a result, you should have the following folder structure:



```
piero@checkforpurity:~/MDB3$ tree
.
├── MDB3.jar
│   ├── MyMDB.class
│   └── MyMDB.java
├── META-INF
│   └── classes
│       └── web.xml
└── QueueSender.war

4 directories, 3 files
piero@checkforpurity:~/MDB3$
```

If you have some experience in web development, you will recognize **web.xml** as the file to configure for the servlet deployment. The **web.xml** file should have the following tags inside it:

```
<web-app>
<servlet>
  <servlet-name>QueueSenderServlet</servlet-name>
  <servlet-class>QueueSenderServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>QueueSenderServlet</servlet-name>
  <url-pattern>/QueueSenderServlet</url-pattern>
</servlet-mapping>
</web-app>
```

Basically, the file defines the mapping between the logical name and the URL and class of a servlet. We also add the additional parameter `load-on-startup` that calls the `init` method of the servlet at application startup.

Now, we can go through the interesting activity, that is, coding a servlet that uses our MDB to produce messages in the test queue. The final result will be the message printed in the console when the `OnMessage` method is fired.

Remember that the servlet must extend the `HttpServlet` class. To code the servlet correctly, we need to add the `javaee-api` JAR to the classpath.

Now to code the servlet in the right position, we need to move to the `classes` folder where we create the `QueueSenderServlet.java` file.

The code for the servlet is as follows:

```
import java.io.*;
import javax.jms.*;
import javax.naming.*;
import javax.servlet.http.*;
import javax.servlet.ServletException;

public class QueueSenderServlet extends HttpServlet
{
  PrintWriter out;

  final String CNN_FACTORY="/ConnectionFactory";

  String QUEUE_NAME="queue/Test";

  QueueConnectionFactory qconFactory;
  QueueConnection qcon;
  QueueSession qsession;
  QueueSender qsender;
  Queue queue;
  TextMessage msg;

  public void service(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException
    {
    try
    {
      out=response.getWriter();
      InitialContext ic = new InitialContext();
      qconFactory = (QueueConnectionFactory) ic.lookup(CNN_FACTORY);
      qcon = qconFactory.createQueueConnection();
      qsession = qcon.createQueueSession(false, Session.AUTO_
ACKNOWLEDGE);
      queue = (Queue) ctx.lookup(queueName);
      qsender = qsession.createSender(queue);
      msg = qsession.createTextMessage();
      qcon.start();

        out.println("");
        out.println("");
        out.println("");
        out.println("<h1>Queue Sender Servlet</h1>");
        out.println("Following Messages has been sent !!!");
        out.println("====================================");
            msg.setText("this is servlet text message");        //
Messages
            qsender.send(msg);       // Messages sent
            out.println("Message Sent");

        out.println("====================================");
        out.println("");
        out.println("");
        out.println("");

    }
    catch(Exception e)
    {
      e.printStackTrace();
    } finally{
  if (qsession != null)
```

```
        qsession.close();
      if (qconn != null)
        qconn.close();
    }
      }

    }
```

As you can see, while connecting to a JMS resource we always use JNDI lookup to the queue that was configured to the JBoss' main configuration file.

The port and so on are coded as default. As usual, we add the out `PrintWriter` object for the servlet that will contain the output of the page, so that you can have feedback from the page.

Now we need to compile the servlet so as to have the `.class` files inside the classes folder.

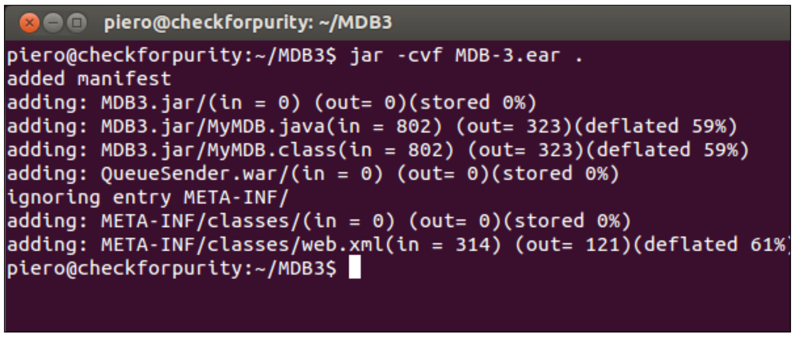
The next step is to pack everything into an EAR file, which will be deployed to the JBoss container.

To do this you could use the JAR command provided with the JDK. So move to the `MDB3` main folder and enter the following command:

```
jar -cvf MDB-3.ear.
```

Do not forget to add the final period, otherwise the example will not work.

As a result, we now have the `MDB-3.ear` file packed in the `MDB3` folder as shown in the following screenshot:



```
piero@checkforpurity:~/MDB3$ jar -cvf MDB-3.ear .
added manifest
adding: MDB3.jar/(in = 0) (out= 0)(stored 0%)
adding: MDB3.jar/MyMDB.java(in = 802) (out= 323)(deflated 59%)
adding: MDB3.jar/MyMDB.class(in = 802) (out= 323)(deflated 59%)
adding: QueueSender.war/(in = 0) (out= 0)(stored 0%)
ignoring entry META-INF/
adding: META-INF/classes/(in = 0) (out= 0)(stored 0%)
adding: META-INF/classes/web.xml(in = 314) (out= 121)(deflated 61%
piero@checkforpurity:~/MDB3$
```

As a last step, we need to deploy everything. JBoss offers a command-line utility to do this, but we need to configure every step into a file that will be passed as a parameter to the utility.

So now move to the `JBOSS_ROOT\bin` folder and create a file called `queue-setup.cli`, which will have the following instruction inside it:

```
connect
deploy /home/piero/MDB3/MDB-3.ear
```

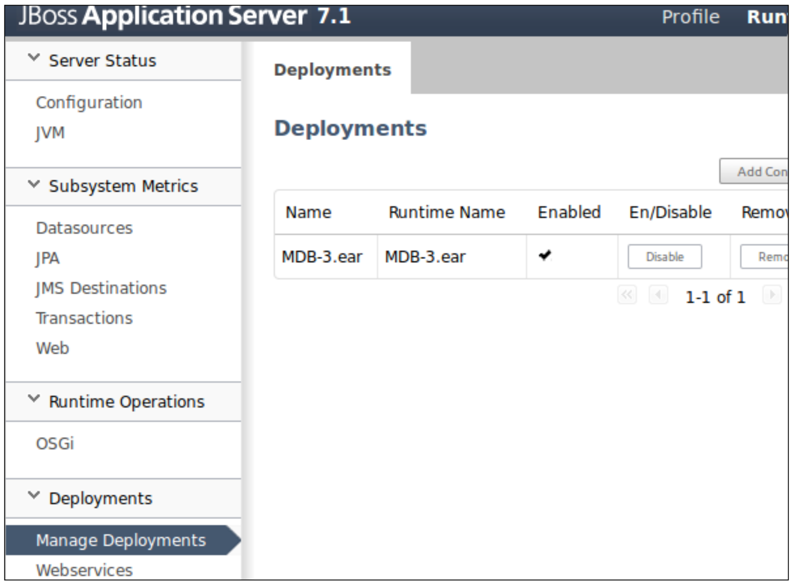
The `deploy` command should have the absolute path to the EAR file previously generated, so change the second command accordingly.

From these commands that were used by JBoss, it is also possible to generate the queue and do other interesting operations on the queues. We suggest you take a look at the documentation from JBoss to understand how to use these commands. The URL of the documentation is `https://docs.jboss.org/author/display/AS71/CLI+Recipes`.

Now to launch the `queue-setup.cli` file, simply enter the following command within the `JBOSS_ROOT\bin` folder.

```
./jboss-cli.sh --file=queue-setup.cli
```

Alternatively, it is also possible to use the web admin console to deploy the EAR file, as shown in the following screenshot:



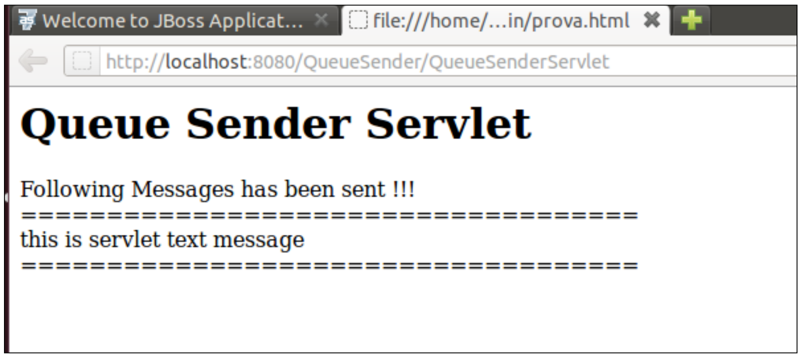
Deployments

Name	Runtime Name	Enabled	En/Disable	Remo
MDB-3.ear	MDB-3.ear	✓	Disable	Remo

Once the deployment is finished, you could simply point your browser to the URL:

`http://localhost:8080/QueueSender/QueueSenderServlet`

You should see the following page:


Welcome to JBoss Applicat...
file:///home/...in/prova.html
http://localhost:8080/QueueSender/QueueSenderServlet

Queue Sender Servlet

```
Following Messages has been sent !!!
=======================================
this is servlet text message
=======================================
```

This time we can truly say "eureka!".

As you can see, once the Message Driven Bean has been correctly configured, all the techniques for sending messages could be re-used in a servlet without any major issues.

We gave this example only for demonstration purposes. However, there are a lot of possible configurations that can be done using JBoss and HornetQ. We invite you to take a look at the documentation and at the forums that could be a rich resource for managing all this configuration.

Now that we have seen how to use a servlet, we invite you to create a JSP page that does the same thing as the servlet.

If you are willing, you would like to lose some sleep thinking about another interesting problem.

> Considering that in our book's example, the devices store ECG data using HornetQ, we invite you to think of creating a web service—we suggest using **Apache Axis**—that receives the ECG data or an `HL7` file and then use this information to store a message in a JBoss-deployed JMS queue.

Summary

As last revision for all have done, we remember that in this chapter, we have seen how to install and set up JBOSS 7.1.1 to work with HornetQ. We have also seen how to create, deploy, and test a servlet, which uses a Message Driven Bean to store a message in a queue.

We are now ready to move to our last chapter where we will see how to run HornetQ in an embedded code. This will demonstrate how we can provide our Java application with a powerful message-oriented framework without having to configure an environment from scratch, but using only JAR files and runtime settings.

11
More on HornetQ Embedding

In the previous chapter we saw how to integrate HornetQ in JBoss, so we saw how a big application server can interface with HornetQ. We also saw how to launch HornetQ in a standalone or clustered way to parse messages.

This last chapter is devoted to seeing how to use HornetQ in different scenarios and we will describe how to correctly integrate HornetQ. We will cover the following topics:

- Using HornetQ server functionalities in a Java application, as a standalone server or in a clustered way
- Using the STOMP protocol we will see how to allow different STOMP clients talking to HornetQ
- Integrating HornetQ within the latest framework for J2EE programming like Spring

Embedding the HornetQ server in your application

The reader should be familiar with the client server programming of HornetQ so that he/she should be able to use the example code to equip any Java application to send messages to a HornetQ server.

But HornetQ is nothing more than a set of JAR files equipped with a launch script and some parameters to read the `config` files so that it is possible to run the HornetQ server.

If you open the usual `run.sh` file, the line that does the magic is the following one:

```
java $JVM_ARGS -classpath $CLASSPATH -Dcom.sun.management.jmxremote
org.hornetq.integration.bootstrap.HornetQBootstrapServer $FILENAME
```

The experienced Java programmer should see that the main class that launches the HornetQ instance is the `HornetQBootstrapServer` class within the `config` file.

In the following section, we will cover how to include a running instance of the HornetQ server so as to embed a powerful asynchronous messaging server in your Java application, both in a desktop or web environment.

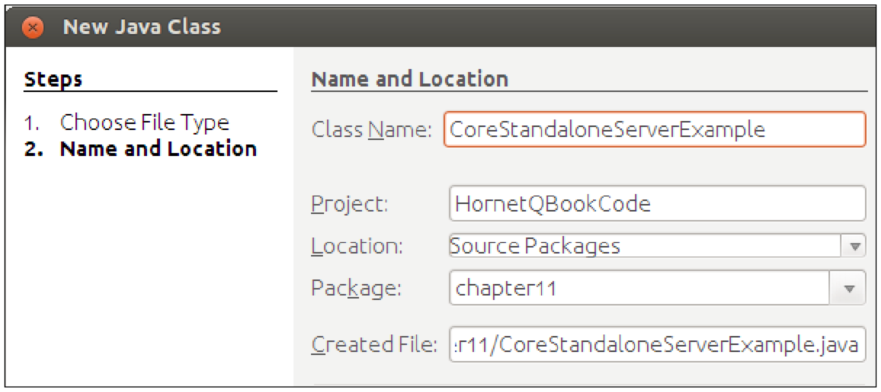
Embedding HornetQ in a simple way

In this section, we will learn how to embed a standalone HornetQ running instance into a Java application using the `hornetq-configuration.xml` main configuration file. Our server instance will be a core API server that is the simplest one and it is the native one for messaging functionalities using HornetQ.

In our example we will start a standalone non-clustered server using the configuration file that is present in the `HORNETQ_ROOT\config\standalone\non-clustered\` folder. This file should be placed inside the classpath of the Java application to be retrieved when the server reads the configuration.

Our test environment will be Ubuntu with NetBeans. As we have seen in *Chapter 2, Setting Up HornetQ*, the `HORNETQ_ROOT` folder corresponds in our test environment with the absolute path `/hornetq`.

Now open NetBeans and create a new class called `CoreStandaloneServerExample`, as shown in the following screenshot:

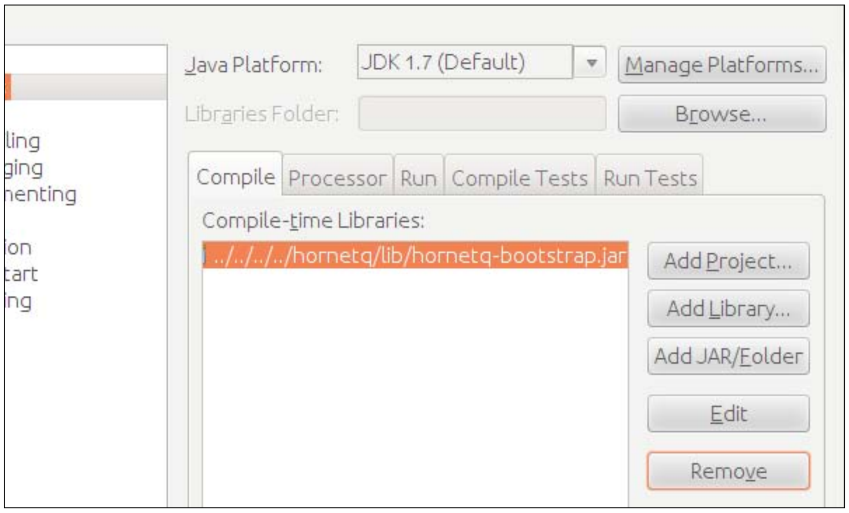


Class Name: CoreStandaloneServerExample
Project: HornetQBookCode
Location: Source Packages
Package: chapter11
Created File: :r11/CoreStandaloneServerExample.java

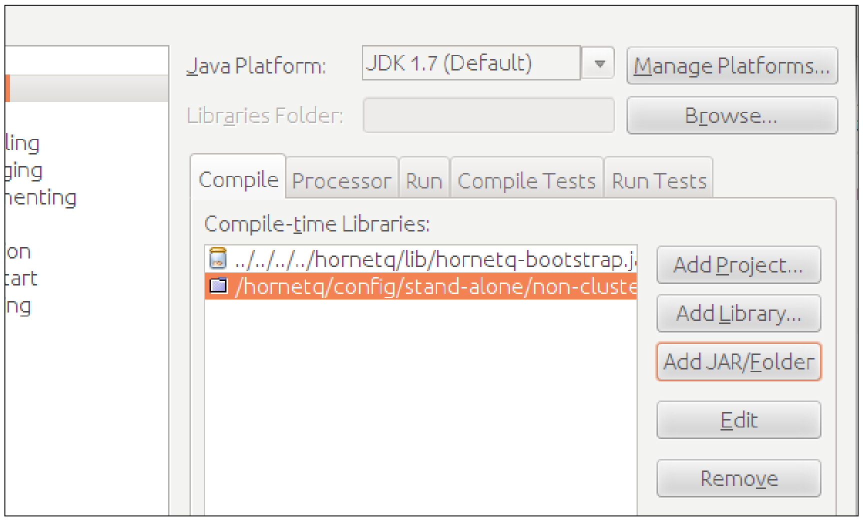
For creating a running server instance you need two JAR files: `hornetq-core.jar` and `netty.jar`. You require the first one because it contains the classes that are needed to manage a server instance. The second one is needed because Netty is the default HTTP server used for accepting connections.

You should also add the `hornetq-configuration.xml` file to the build classpath located in the `HORNETQ_ROOT\config\standalone\non-clustered\` folder. To do this in NetBeans you should perform the following steps:

1. Right-click on **Project properties**. On the left panel click on **Libraries** so as to arrive at the following screen:



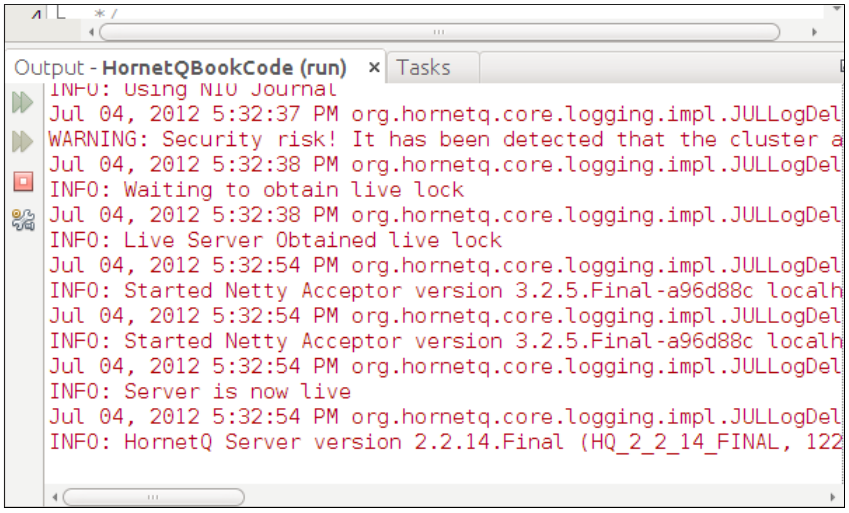
2. Now click on the **Add Jar/Folder** button and add the `HORNETQ_ROOT\config\standalone\non-clustered\` folder, so the screen should look like the following screenshot:



3. Now click on **OK** to confirm the choices. To run the server, the next step is simply to create an instance of the server and to launch it.
4. To do this on your main method add the following lines of code:

   ```
   EmbeddedHornetQ embedded = new EmbeddedHornetQ();
   embedded.start();
   ```

5. Now simply run the Java class and you should see the following output on the standard output console:



```
INFO: Using NIO Journal
Jul 04, 2012 5:32:37 PM org.hornetq.core.logging.impl.JULLogDel
WARNING: Security risk! It has been detected that the cluster a
Jul 04, 2012 5:32:38 PM org.hornetq.core.logging.impl.JULLogDel
INFO: Waiting to obtain live lock
Jul 04, 2012 5:32:38 PM org.hornetq.core.logging.impl.JULLogDel
INFO: Live Server Obtained live lock
Jul 04, 2012 5:32:54 PM org.hornetq.core.logging.impl.JULLogDel
INFO: Started Netty Acceptor version 3.2.5.Final-a96d88c localh
Jul 04, 2012 5:32:54 PM org.hornetq.core.logging.impl.JULLogDel
INFO: Started Netty Acceptor version 3.2.5.Final-a96d88c localh
Jul 04, 2012 5:32:54 PM org.hornetq.core.logging.impl.JULLogDel
INFO: Server is now live
Jul 04, 2012 5:32:54 PM org.hornetq.core.logging.impl.JULLogDel
INFO: HornetQ Server version 2.2.14.Final (HQ_2_2_14_FINAL, 122
```

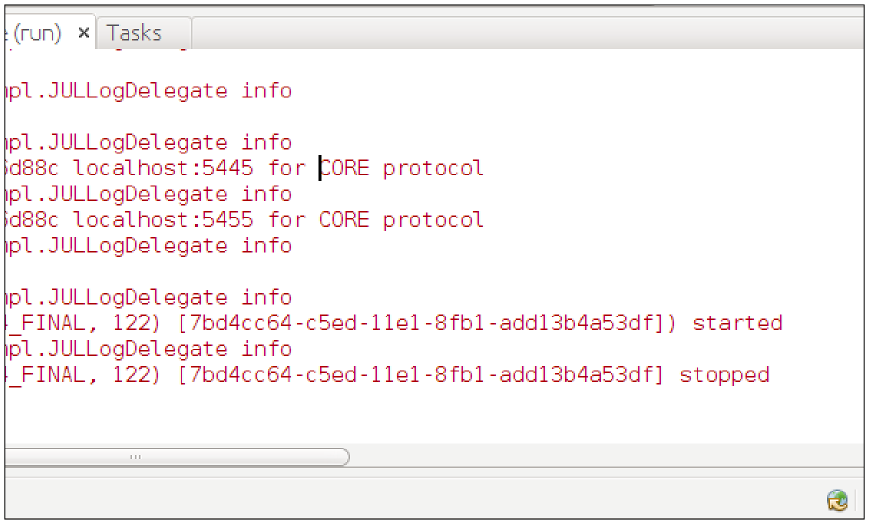
That's it! With two JAR files, two lines of code, and one configuration file we were able to launch a running instance of a core HornetQ standalone server from our application.

We can also extend our very simple example by adding the call to the `stop` method to show the reader how to stop a HornetQ core server from a simple application.

Again the code to do this is very simple and it is as follows:

```
EmbeddedHornetQ embedded = new EmbeddedHornetQ();
embedded.start();
Thread.sleep(10000);
embedded.stop();
```

Re-launching the `CoreStandaloneServerExample` class will produce the following output:



```
pl.JULLogDelegate info

pl.JULLogDelegate info
d88c localhost:5445 for CORE protocol
pl.JULLogDelegate info
d88c localhost:5455 for CORE protocol
pl.JULLogDelegate info

pl.JULLogDelegate info
_FINAL, 122) [7bd4cc64-c5ed-11e1-8fb1-add13b4a53df]) started
pl.JULLogDelegate info
_FINAL, 122) [7bd4cc64-c5ed-11e1-8fb1-add13b4a53df] stopped
```

We only add a sleeping period to allow the server to start correctly and avoid having the shutdown signal before completing the starting procedure. If you remove this line, the program will close before the server starts up.

We remind the readers to use this implementation to start the server inside the same JVM that was launched with the class where the server was started.

In our example we use the standard configuration files located in the standard `config` folder. But it is also possible to start a server at runtime by configuring everything at runtime without using the configuration files.

We will use the same example but in this case we will create a configuration at runtime so as to allow everything to be controlled by code.

Let us take a look at the following code:

```
Configuration config = new ConfigurationImpl();
HashSet<TransportConfiguration> transports = new HashSet<TransportCon
figuration>();

transports.add(new TransportConfiguration(NettyAcceptorFactory.class.
getName()));
transports.add(new TransportConfiguration(InVMAcceptorFactory.class.
getName()));
```

```
config.setAcceptorConfigurations(transports);

EmbeddedHornetQ server = new EmbeddedHornetQ();
server.setConfiguration(config);

server.start();
```

Using the `org.hornetq.core.config.Configuration` and `org.hornetq.core.config.impl.ConfigurationImpl` objects we can define the transports for the HornetQ core API (that is the minimalistic configuration), and run the server using this configuration.

As we have seen in *Chapter 5*, *Some More Advanced Features of HornetQ*, it is also possible to define many parameters within the `config` object so as to allocate the message size, the large message properties and so on.

For example, as we saw, we can completely disable any security controls using the following line:

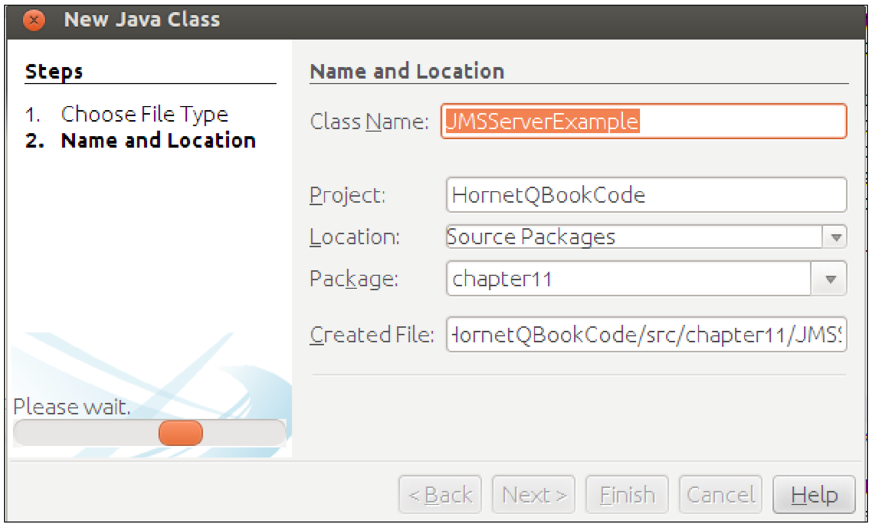
```
config.setSecurityEnabled(false)
```

Just to give a complete code-based example we will create a fully embedded server that deploys a core queue named `ECGQueue`.

Embedding JMS HornetQ server

We have seen how to deploy a core server within a Java application, but in most cases we need to deploy a JMS server to be open to the JMS dialect. In this case we will show you how to deploy a JMS server. As we have seen when using the standalone configuration provided by the default server instance for deploying a JMS queue, we need to define it in the `horrnetq-jms.xml` file. So in this example we will deploy the usual JMS `ECGQueue` using the `hornetq-jms.xml` file and run our server within our application.

Before continuing we will need to create a new class named `JMSServerExample`, as shown in the following screenshot:



Name and Location

Class Name: JMSServerExample

Project: HornetQBookCode

Location: Source Packages

Package: chapter11

Created File: HornetQBookCode/src/chapter11/JMS

Now add the usual `HORNETQ_ROOT\config\standalone\non-clustered` folder to the project classpath so that both the `hornetq-configuration.xml` file and the `hornet-jms.xml` file are visible to the Java code at runtime.

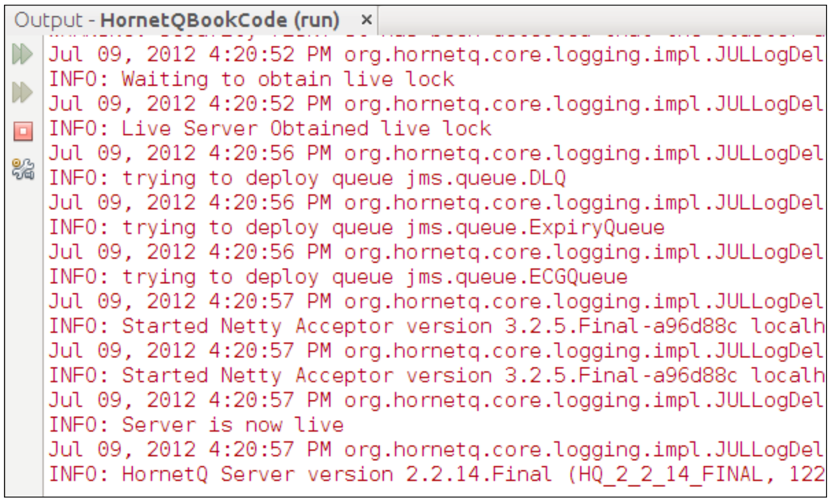
Now, using a text editor we will add the `ECGQueue` queue by adding the usual tag to the `hornetq-jms.xml` file:

```
<queue name="ECGQueue">
   <entry name="/queue/ECGQueue"/>
</queue>
```

Now moving on to the coding part we only need to start the `JMSServer` as we did before, again the code is pretty simple and is as follows:

```
EmbeddedJMS jmsServer = new EmbeddedJMS();
jmsServer.start();
```

In the output console, we have the server startup messages, as well as the queue deployed messages:



Output - HornetQBookCode (run) ×

```
Jul 09, 2012 4:20:52 PM org.hornetq.core.logging.impl.JULLogDel
INFO: Waiting to obtain live lock
Jul 09, 2012 4:20:52 PM org.hornetq.core.logging.impl.JULLogDel
INFO: Live Server Obtained live lock
Jul 09, 2012 4:20:56 PM org.hornetq.core.logging.impl.JULLogDel
INFO: trying to deploy queue jms.queue.DLQ
Jul 09, 2012 4:20:56 PM org.hornetq.core.logging.impl.JULLogDel
INFO: trying to deploy queue jms.queue.ExpiryQueue
Jul 09, 2012 4:20:56 PM org.hornetq.core.logging.impl.JULLogDel
INFO: trying to deploy queue jms.queue.ECGQueue
Jul 09, 2012 4:20:57 PM org.hornetq.core.logging.impl.JULLogDel
INFO: Started Netty Acceptor version 3.2.5.Final-a96d88c localh
Jul 09, 2012 4:20:57 PM org.hornetq.core.logging.impl.JULLogDel
INFO: Started Netty Acceptor version 3.2.5.Final-a96d88c localh
Jul 09, 2012 4:20:57 PM org.hornetq.core.logging.impl.JULLogDel
INFO: Server is now live
Jul 09, 2012 4:20:57 PM org.hornetq.core.logging.impl.JULLogDel
INFO: HornetQ Server version 2.2.14.Final (HQ_2_2_14_FINAL, 122
```

As for the usual standalone server, we can add more queues at runtime by modifying the `hornetq-jms.xml` file accordingly and the new queue will be automatically deployed by our embedded server.

By adding the complete `HORNETQ_ROOT\config\standalone\non-clustered` folder we add all the files needed but we should remember that the files that are used are the following ones:

- `hornetq-configuration.xml`
- `HornetQ-jms.xml`
- `HornetQ-users.xml`

As for the core case it is possible to have a fully hardcoded JMS server by specifying the connectors and the queues to be used. In this second example we will deploy the usual `ECGQueue` queue using only runtime code without any configuration files. So prior to starting our example, remove the `HORNETQ_ROOT\config\standalone\non-clustered` folder from the classpath just to be sure that the only configuration acceptors are the ones at code level.

For our example to work we need the following steps to be followed:

1. Create a HornetQ configuration and set the correct properties.
2. Create the JMS configuration and the JMS `connectionfactory` object.

3. Configure the queues.
4. Load the configuration and start the server.

So as we did for the core embedded example, we need to use the `configuration` object and also the `JMSConfiguration` object because the queues are stored on a separate object.

To start let us create the configuration:

```
Configuration configuration = new ConfigurationImpl();

configuration.setSecurityEnabled(false);
configuration.getAcceptorConfigurations().add(new TransportConfigurati
on(NettyAcceptorFactory.class.getName()));

TransportConfiguration connectorConfig = new TransportConfiguration(Ne
ttyConnectorFactory.class.getName())

configuration.getConnectorConfigurations().put("connector",
connectorConfig);
```

As the reader can see, we created a core `configuration` object and we disabled the security with the following line:

```
configuration.setSecurityEnabled(false);
```

But more importantly we add an acceptor of type `netty` and a connector of type `netty` on the same virtual machine that is running the server.

Next we need to create the JMS `configuration` and `jmsConnectionFactory` object using the following code:

```
JMSConfiguration jmsConfig = new JMSConfigurationImpl();
ArrayList<String> connectorNames = new ArrayList<String>();
connectorNames.add("connector");
ConnectionFactoryConfiguration cfConfig = new ConnectionFactoryConfigu
rationImpl("cf", false, connectorNames, "/cf");
jmsConfig.getConnectionFactoryConfigurations().add(cfConfig);
```

So we basically created a `JMSConfiguration` object and added to this a `ConnectionFactoryConfiguration` object.

We are now ready to add the queues. We need only one in this case:

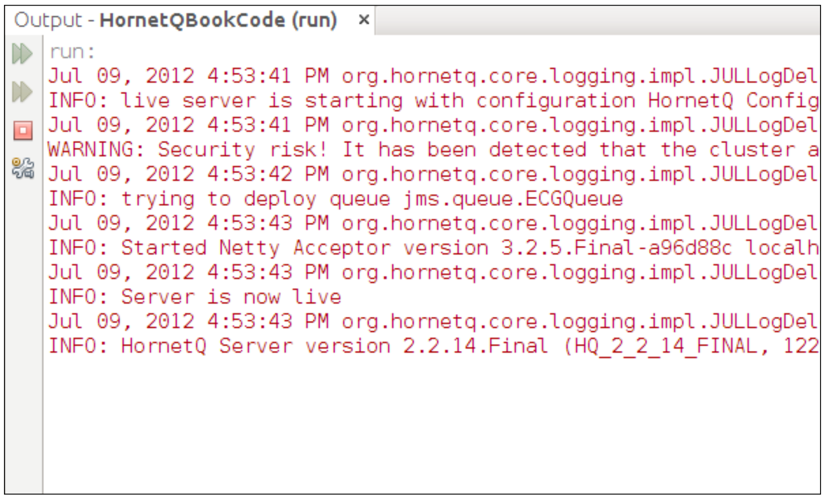
```
JMSQueueConfiguration queueConfig = new JMSQueueConfigurationImpl("ECG
Queue", null, false, "/queue/ECGQueue");
jmsConfig.getQueueConfigurations().add(queueConfig);
```

And at the end put all of them together and start the `EmbeddedJMS` server object:

```
EmbeddedJMS jmsServer = new EmbeddedJMS();
jmsServer.setConfiguration(configuration);
jmsServer.setJmsConfiguration(jmsConfig);
jmsServer.start();
```

Core implementation was simpler than the JMS one but again we point out that this is due to the fact that the JMS queues are deployed in a different way with respect to the core queue in HornetQ.

Lastly, just opening the `run.sh` file on NetBeans will produce the following output:



```
run:
Jul 09, 2012 4:53:41 PM org.hornetq.core.logging.impl.JULLogDel
INFO: live server is starting with configuration HornetQ Config
Jul 09, 2012 4:53:41 PM org.hornetq.core.logging.impl.JULLogDel
WARNING: Security risk! It has been detected that the cluster a
Jul 09, 2012 4:53:42 PM org.hornetq.core.logging.impl.JULLogDel
INFO: trying to deploy queue jms.queue.ECGQueue
Jul 09, 2012 4:53:43 PM org.hornetq.core.logging.impl.JULLogDel
INFO: Started Netty Acceptor version 3.2.5.Final-a96d88c localh
Jul 09, 2012 4:53:43 PM org.hornetq.core.logging.impl.JULLogDel
INFO: Server is now live
Jul 09, 2012 4:53:43 PM org.hornetq.core.logging.impl.JULLogDel
INFO: HornetQ Server version 2.2.14.Final (HQ_2_2_14_FINAL, 122
```

So here the `ECGQueue` queue was deployed successfully and the server was started correctly. We suggest the willing reader to try to familiarize himself/herself with the `Configuration` and `JMSConfiguration` objects to get used to all the `config` parameters that can be set at runtime. We are now ready to move to a more complex embedding.

Embedding HornetQ to create a cluster

We have seen in *Chapter 6, Clustering with HornetQ*, that there is the possibility for various instances of HornetQ to act like a cluster so that the messages received from a node are automatically distributed to the other ones. Or we can share a topic among different nodes, so you can have a high-frequency asynchronous cluster shared within your Java application.

As the reader may have understood, we will give the same example of how to create a HornetQ cluster node in a Java application. We will use the default configuration file for a cluster environment without changing any setting about port communication.

This is only an example that is based on the fact that we imagine that our application will run one instance for one machine without all the port binding issues that we have seen in *Chapter 6*, *Clustering with HornetQ*.

So the only thing you have to change is the configuration file (`hornetq-configuration.xml`) that should be shared between every node. For the rest, the code remains the same.

Using STOMP with HornetQ

The user should now be aware of the power of HornetQ but a question still remains. At the moment we have only used a Java client that embedded the client JAR files that are needed to create the connection, the session, and finally send the messages in core or JMS format.

In all our examples we assumed that the client producer was coded in Java; nevertheless it is possible, in real world applications, that other clients would need to interact with a JMS queue for sending messages. In this final chapter we will see how to interact with an embedded HornetQ STOMP server using a client that is not coded in Java.

HornetQ Version 2.2.2 also integrates the REST interface paradigm, but we will not cover it.

STOMP (**Simple Text Oriented Messaging Protocol**) is a protocol that is used to send text messages; it has lots of client implementations in different programming languages. In this chapter, we will see how to enable the STOMP protocol both in the standalone and in the embedded version.

Enabling STOMP in the standalone version

In this example, we will run a standalone HornetQ STOMP server, which is able to receive STOMP messages from a command-line client; for other clients the reader can refer to the official STOMP site (`http://stomp.github.com`). To enable STOMP in the HornetQ server we need to use, as usual, the `hornetq-configuration.xml` configuration file. So open up the `hornetq-configuration.xml` file located at `HORNETQ_ROOT\config\standalone\non-clustered` with a text editor. Once opened, search for the `<acceptor>` tag. You will find the `netty-acceptor` tag. Add another acceptor tag as follows:

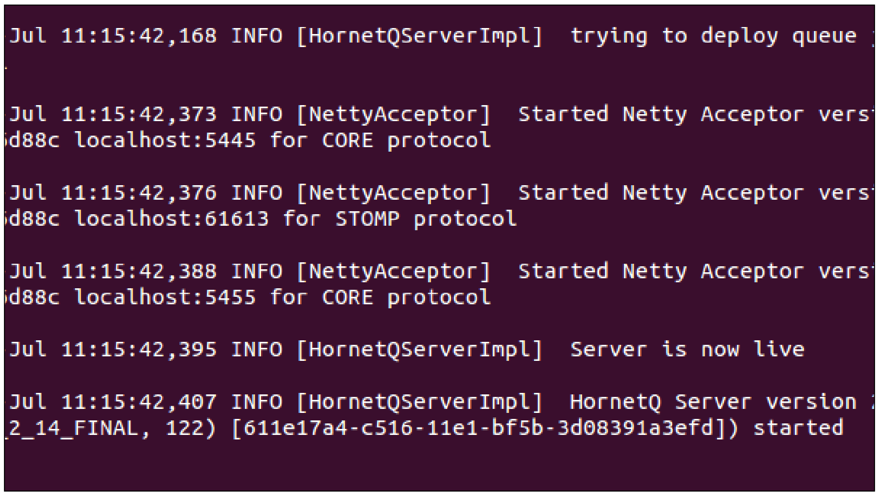
```
        <acceptor name="stomp-acceptor">
           <factory-class>org.hornetq.core.remoting.impl.netty.
NettyAcceptorFactory</factory-class>
           <param key="protocol" value="stomp" />
           <param key="port" value="61613" />
        </acceptor>
```

So we define a new acceptor called `stomp-acceptor` that is netty-based. It uses the STOMP protocol and replies on the default STOMP port that is the 61613 even if it can be changed by the user.

To make things easy as STOMP implements HornetQ security, we suggest to add the following tag to disable the security mechanism:

```
<security-enabled>false</security-enabled>
```

Once done this will run the main server and you will see the following output in the command-line console:



```
Jul 11:15:42,168 INFO [HornetQServerImpl]  trying to deploy queue

Jul 11:15:42,373 INFO [NettyAcceptor]  Started Netty Acceptor vers
d88c localhost:5445 for CORE protocol

Jul 11:15:42,376 INFO [NettyAcceptor]  Started Netty Acceptor vers
d88c localhost:61613 for STOMP protocol

Jul 11:15:42,388 INFO [NettyAcceptor]  Started Netty Acceptor vers
d88c localhost:5455 for CORE protocol

Jul 11:15:42,395 INFO [HornetQServerImpl]  Server is now live

Jul 11:15:42,407 INFO [HornetQServerImpl]  HornetQ Server version
2_14_FINAL, 122) [611e17a4-c516-11e1-bf5b-3d08391a3efd]) started
```

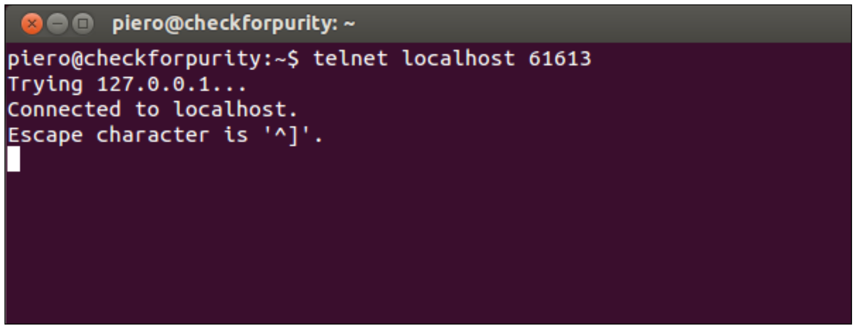
Here the reader can see that the STOMP server is running on port 61613.

Producing messages on a STOMP server can be done by a simple telnet series of commands; in this example we will send messages on the `ExampleQueue` queue, by specifying it directly. The reader can use the security setting to redirect the STOMP messages to another queue as we have seen in *Chapter 6, Clustering with HornetQ*, by simply changing the `hornetq-configuration.xml` file.

To send a message the reader should open another terminal window and start opening a STOMP session using the following command:

```
telnet localhost 61613
```

As a result the reader should see the following screen:



```
piero@checkforpurity:~$ telnet localhost 61613
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
```

Now give the following series of commands in sequence to see that the server is responding using the STOMP protocol:

1. `CONNECT`
2. `Login:`
3. `passcode`

Running an embedded STOMP server

In this part we will run an embedded STOMP server by using the configuration possibilities offered by the `config` HornetQ object.

This example has been coded using one created by one of the HornetQ developers Jeff Mesnil. For a full reference the reader could take a look at `http://github.com/jmesnil/hornetq-stomp`.

To start with our example we need to create a class named `StompEmbeddedServer`.

An extra JAR file is needed that is `stompconnect-1.0.jar` located at `http://repository.codehaus.org/org/codehaus/stomp/stompconnect/1.0/`.

Once done, open the `main` method and add the following code:

```
Configuration configuration = new ConfigurationImpl();

configuration.setSecurityEnabled(false);
configuration.getAcceptorConfigurations().add(new TransportConfigurati
on(InVMAcceptorFactory.class.getName()));
configuration.getQueueConfigurations().add(new
QueueConfiguration("jms.queue.ECGQueue", "jms.queue.ECGQueue",null,
true));



HornetQServer hornetqServer = HornetQServers.newHornetQServer(configu
ration);
JMSServerManager jmsServer = new JMSServerManagerImpl(hornetqServer);
jmsServer.start();

ConnectionFactory connectionFactory = HornetQJMSClient.
createConnectionFactory(
new TransportConfiguration(InVMConnectorFactory.class.getName()));

StompConnect stompConnect = new StompConnect(connectionFactory);
stompConnect.start();
```

Considering the version supported, the actual version of HornetQ Version 2.2.14 supports STOMP Version 1.0 protocol while the new one that is in alpha (Version 2.3.0) will also support Version 1.1.

We are now ready to move to another type of integration—the one with Spring.

Using HornetQ with Spring

Spring is a modular open source application framework that can be used to develop Java applications both for the Web as well as for other uses.

Spring does not impose any specific programming model and has become popular in the Java community as an alternative to the EJB model.

These last pages are not enough to clearly explain all the possibilities offered by Spring considering that it has several modules devoted to different purposes like mobile development, data integration, creating web services, and so on.

In our example we will create a simple application that will send and consume a message using the usual `ECGQueue` queue. We need to point out that it is possible to integrate HornetQ in Spring in a transparent way so that it is possible to automate message pushing when certain events occur, but we will not cover such advanced configurations.

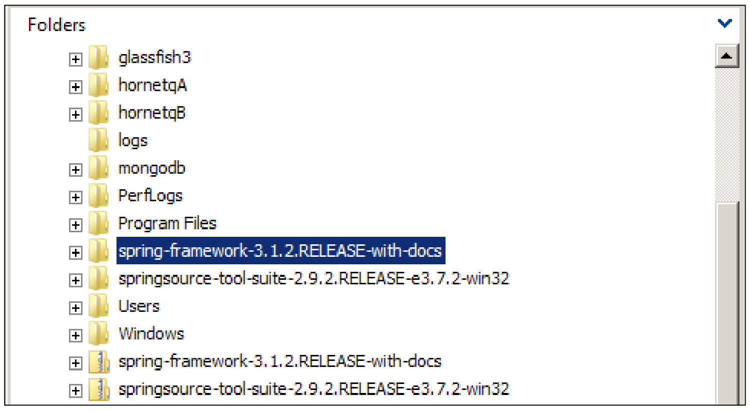
There are two main modules that allow Spring to be used both by Eclipse and NetBeans, but we will not cover such configuration steps. We refer the reader to `http://www.mkyong.com/spring/how-to-install-spring-ide-in-eclipse/` for Eclipse and to `http://netbeans.org/kb/docs/web/quickstart-webapps-spring.html` for NetBeans.

For a quick set up of a development environment we will use the Windows HornetQ installation that we set up in *Chapter 2*, *Setting Up HornetQ*, and we need the following:

- A stable Spring release: We will code our example using the `Spring-framework-3.1.2.RELEASE-with-docs.zip`, available at `http://www.springsource.org/download/community`.
- The SpringSource Tool Suite: This is an Eclipse development environment configured for being Spring ready. You can download the zip file from the URL `http://www.springsource.org/downloads/sts`.
- The Jakarta commons library: This can be found at `http://commons.apache.org/downloads/download_logging.cgi`.

The `commons-logging-1.1.1.jar` JAR file is required at runtime by the Spring framework to work correctly.

Once you have downloaded the files you can unzip the first two to your destination folder, in our case `C:\` so as to have the following folder configuration:


Folders
glassfish3
hornetqA
hornetqB
logs
mongodb
PerfLogs
Program Files
spring-framework-3.1.2.RELEASE-with-docs
springsource-tool-suite-2.9.2.RELEASE-e3.7.2-win32
Users
Windows
spring-framework-3.1.2.RELEASE-with-docs
springsource-tool-suite-2.9.2.RELEASE-e3.7.2-win32

Now to open the IDE browse to the `c:\springsource-tool-suite-2.9.2.RELEASE-e3.7.2-win32\springsource\sts-2.9.2.REALEASE` folder and double-click on the `sts.exe` file. You should see the following screenshot:

Spring - SpringSource Tool Suite
File Edit Navigate Search Project Run Window Help
Welcome
springsourcetoolsuite™
Open Dashboard
Install Extension

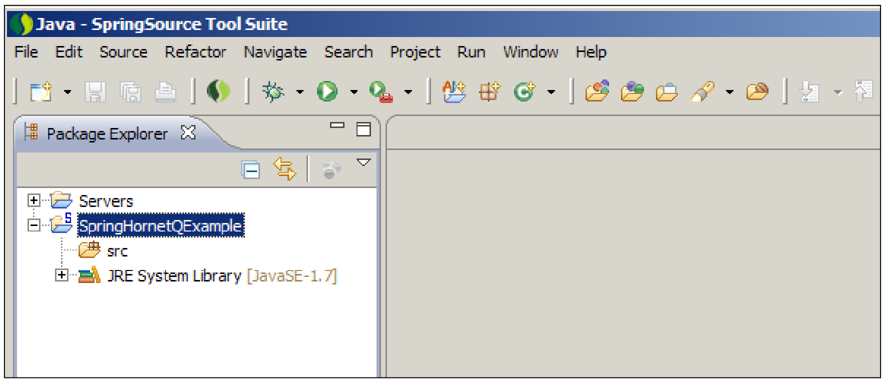
Coding our example

Without entering in to too many details what we are going to do is basically create a simple Spring Console application that will produce and consume the usual ECG message on the usual `ECGQueue` queue. To do this we need to follow these steps:

1. Create a Spring project, which is the easy task.
2. Create an interface for our Bean, where we will define the only method that will be called.
3. Create an implementation class for the Bean interface, where we will create and consume the JMS message.
4. Create a definition file that will map the method on the Bean implementation to an XML definition.
5. Create a method interceptor to correctly invoke the Bean.
6. Finally we will create a tester class that will be used to display the results.

So let us start with the project creation.

From the IDE go to **File** | **Menu** | **Spring Project** and name it `SpringHornetqExample`. You should have the following IDE perspective:


Java - SpringSource Tool Suite
File Edit Source Refactor Navigate Search Project Run Window Help
Package Explorer
Servers
SpringHornetQExample
src
JRE System Library [JavaSE-1.7]

Now we need some Spring JAR files and the usual HornetQ JAR files. The Spring JAR files to be added can be found in the Spring installation folder (in our case `C:\spring-framework-3.1.2.RELEASE-with-docs\spring-framework-3.1.2.RELEASE\dist`). You should add every JAR file you find, then add all the JAR files that HornetQ needs. Do not forget to also add the `commons-logging-1.1.1.jar` file to your libraries.

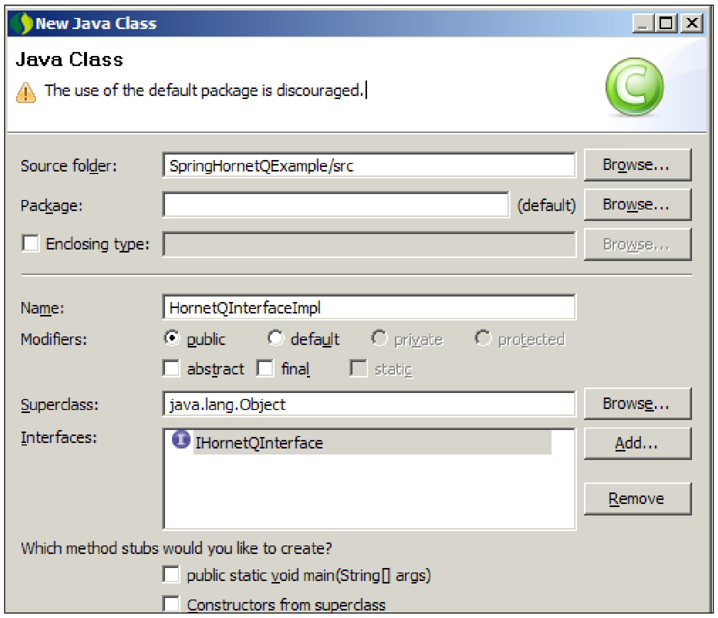
Adding the interface

Spring uses the interfaces in an extensive way to implement our Bean. We first need to create the interface. To do this on the `src` folder of your Eclipse project right-click and add a new interface called `IHornetqMessage` where we need to define the only method `ProduceCosumeMessage()` as underlined by the following code:

```
public interface IHornetQInterface{
    public void ProduceConsumeMessage();
}
```

Creating the Bean

Now we are going to implement the interface we just created using the Eclipse wizard. To do this right-click on the `src` folder and add a new class that implements the `IHornetQMessage` interface as shown in the following screenshot:



The use of the default package is discouraged.

Source folder: SpringHornetQExample/src
Package: (default)
Enclosing type:
Name: HornetQInterfaceImpl
Modifiers: [✓ public] default private protected
abstract final static
Superclass: java.lang.Object
Interfaces: IHornetQInterface
Which method stubs would you like to create?
public static void main(String[] args)
Constructors from superclass

Now in the `ProduceConsumeMessage` method we need to code the consuming and producing messages, so with some slight modification we can reuse the code provided in *Chapter 1, Getting Started with HornetQ*.

So basically the method should look like this:

```
try {
    Connection connection = null;
    InitialContext initialContext = null;
    try
    {
            java.util.Properties p = new java.util.Properties();
    p.put(javax.naming.Context.INITIAL_CONTEXT_FACTORY,"org.jnp.
interfaces.NamingContextFactory");
    p.put(javax.naming.Context.URL_PKG_PREFIXES,"org.jboss.naming:org.
jnp.interfaces");
            p.put(javax.naming.Context.PROVIDER_URL, "jnp://
localhost:1099");
```

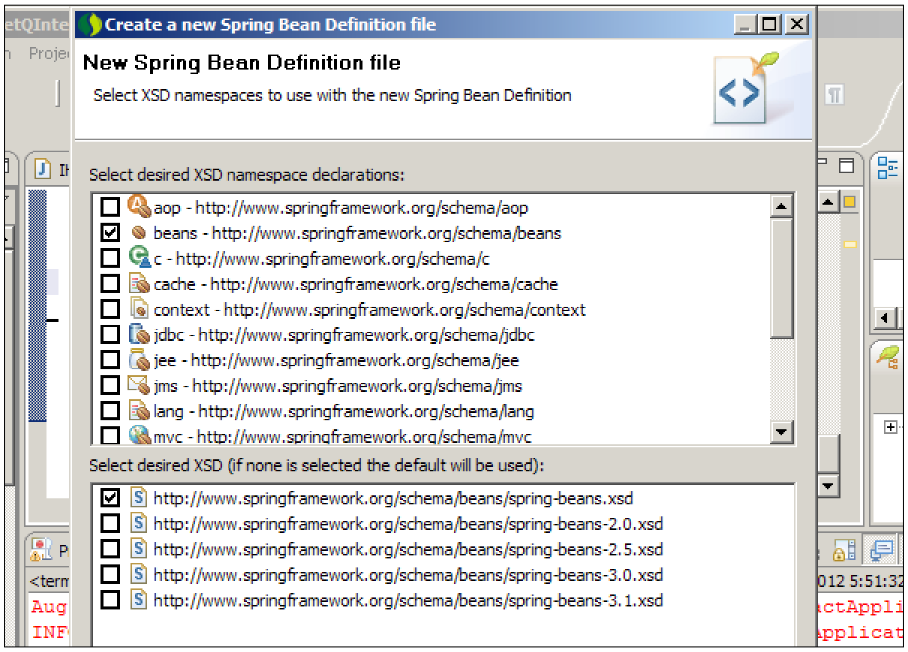
```
          initialContext = new javax.naming.InitialContext(p);
          Queue queue = (Queue)initialContext.lookup("/queue/
ECGQueue");
          ConnectionFactory cf = (ConnectionFactory)initialContext.
lookup("/ConnectionFactory");
          connection = cf.createConnection();
          Session session = connection.createSession(false, Session.
AUTO_ACKNOWLEDGE);

          MessageProducer producer = session.createProducer(queue);
          TextMessage message = session.createTextMessage("This is a
text message");
          System.out.println("Sent message: " + message.getText());
          producer.send(message);
          MessageConsumer messageConsumer = session.
createConsumer(queue);
          connection.start();
          TextMessage messageReceived = (TextMessage)messageConsumer.
receive(5000);
          System.out.println("Received message: " + messageReceived.
getText());
   }
   finally
   {
          if (initialContext != null)
          {
                initialContext.close();
          }
          if (connection != null)
          {
                connection.close();
          }
   }
} catch (Exception ex){
   ex.printStackTrace();
}
```

Do not forget to add the ECGQueue on your HornetQ server.

Creating the configuration XML

Spring needs an XML configuration file that is used to map the Bean-implemented class we just coded. To do this the following steps need to be followed:

1. Right-click on the `src` folder and go to **New | Spring Bean Configuration File**.
2. Add the name of the XML; in our case `Bean.xml`.
3. Then select the type of Bean file and the XSD associate (in our case the generic one), as shown in the following screenshot:



New Spring Bean Definition file

Select XSD namespaces to use with the new Spring Bean Definition

Select desired XSD namespace declarations:

- [ ] aop - http://www.springframework.org/schema/aop
- [✓] beans - http://www.springframework.org/schema/beans
- [ ] c - http://www.springframework.org/schema/c
- [ ] cache - http://www.springframework.org/schema/cache
- [ ] context - http://www.springframework.org/schema/context
- [ ] jdbc - http://www.springframework.org/schema/jdbc
- [ ] jee - http://www.springframework.org/schema/jee
- [ ] jms - http://www.springframework.org/schema/jms
- [ ] lang - http://www.springframework.org/schema/lang
- [ ] mvc - http://www.springframework.org/schema/mvc

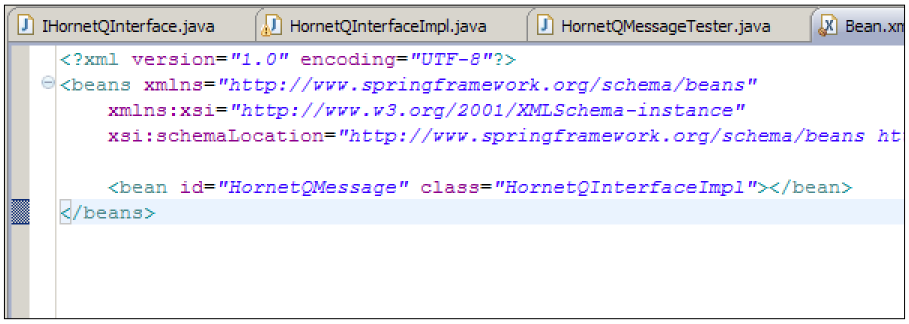
Select desired XSD (if none is selected the default will be used):

- [✓] http://www.springframework.org/schema/beans/spring-beans.xsd
- [ ] http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
- [ ] http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
- [ ] http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
- [ ] http://www.springframework.org/schema/beans/spring-beans-3.1.xsd

4. After clicking on **Finish** an XML file will be created and opened in your workspace. Now we need to map the Bean class name with a logical name that will be called in our test class.
5. Inside the XML file add the following tag between the `<beans>` tag:

```
<bean id="HornetQMessage" class="HornetQInterfaceImpl"></bean>
```

You should get the following configuration:



```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans ht

    <bean id="HornetQMessage" class="HornetQInterfaceImpl"></bean>
</beans>
```

6. Now as a final step we can code a test class.

Creating the test class

As the last step we can now write our test class to see that everything is correct. To do this create a new class with a `main` method and call it `HornetQMessageTester`. Inside the `main` method add the following code:

```
ApplicationContext context = new ClassPathXmlApplicationContext("Bean.
xml");

HornetQInterfaceImpl test = (HornetQInterfaceImpl) context.
getBean("HornetQMessage");

test.ProduceConsumeMessage();
```

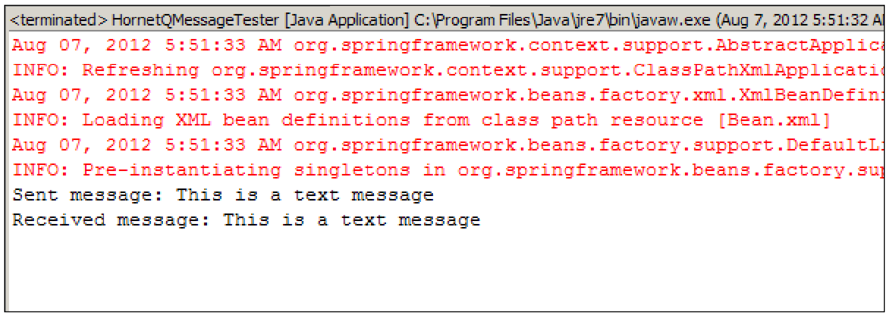
Only to comment on the three code lines, we create an application contest that is configured using the Spring configuration file we just created using the following line:

```
ApplicationContext context = new ClassPathXmlApplicationContext("Bean.
xml");
```

Then we create the object that implements the `HornetQInterfaceImpl` interface using the class mapped by the logical name `HornetQMessage`:

```
HornetQInterfaceImpl test = (HornetQInterfaceImpl) context.
getBean("HornetQMessage");
```

So we can call the `ProduceConsumeMessage` method. Now that we have coded everything we can run the example and as a result we will see the following screenshot:



```
<terminated> HornetQMessageTester [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (Aug 7, 2012 5:51:32 A
Aug 07, 2012 5:51:33 AM org.springframework.context.support.AbstractApplic
INFO: Refreshing org.springframework.context.support.ClassPathXmlApplicati
Aug 07, 2012 5:51:33 AM org.springframework.beans.factory.xml.XmlBeanDefin
INFO: Loading XML bean definitions from class path resource [Bean.xml]
Aug 07, 2012 5:51:33 AM org.springframework.beans.factory.support.DefaultL
INFO: Pre-instantiating singletons in org.springframework.beans.factory.su
Sent message: This is a text message
Received message: This is a text message
```

So after some effort we coded a basic example where we were able to use HornetQ for producing and consuming JMS messages.

Despite this very simple example, we think that this could be a good starting point for letting the reader understand that HornetQ can be used in nearly any Java-based environment, both if it is a framework or a simple client application.

Spring is a multipurpose framework, so its modularity allows the reader to interact with many different types of specific functionalities, but as we have seen this makes it easy to connect it to the high-performing HornetQ server.

We challenge the reader to a more difficult task that is to create a Spring-based web service using the Spring Web Services module so as to have a web frontend for allowing different clients to send messages to a protected HornetQ server.

Summary

In this chapter we learned how to embed a HornetQ server into a Java application using both the JMS and core API implementation, use HornetQ as a STOMP server for STOMP client, and use HornetQ with Spring to produce/consume JMS messages.

This trip has finished dear reader. We started our journey by coding a simple example and we finished by using one of the latest frameworks for development purposes. We have seen a lot of examples and different environments. However, as one of my university teacher once told me, "learning is what still remains when you forget all the notions".

We hope that this is what remains with you after nearly 300 pages. If you end this book thinking that your money was thrown away because you did not find a particular trick for your HornetQ installation, I think that we failed in writing it.

If you end this book with some adjustments in your mind to your day-to-day software using HornetQ, we have perfectly matched our initial scope.

From here where can you go? The first suggestion is to start giving feedback for our work to understand if something could be explained better and if some major mistakes have been written black on white. We really appreciate our readers' feedback.

Next, considering that HornetQ is built upon a very dynamic community with an open source minded approach, why not participate in some forums or even in the coding phase?

The ability of a software or a platform to manage, in an asynchronous way, millions of messages is becoming a "know-how" that cannot be avoided in a senior programmer's resume. So we hope you can profit from the knowledge we shared with you both from a personal and professional perspective. We also hope that you enjoyed reading this book as much as we enjoyed writing it. Good luck with your code!

Index