SAMPLE CHAPTER

ActiveMQ IN ACTION

Bruce Snyder
Dejan Bosanac
Rob Davies

MANNING

ActiveMQ in Action

by Bruce Snyder, Dejan Bosanac, Rob Davies

Chapter 1

brief contents

1 Introduction to Apache ActiveMQ

This chapter covers

- A high-level overview of ActiveMQ features and uses
- Downloading and installing ActiveMQ
- Understanding the ActiveMQ directory structure
- Running examples that come with ActiveMQ

Enterprise messaging software has been in existence since the late 1980s. Not only is messaging a style of communication between applications, it's also a style of integration. Therefore, messaging fulfills the need for both notification as well as interoperation among applications. But open source solutions have only emerged in the last 10 years. Apache ActiveMQ is one such solution, providing the ability for applications to communicate in an asynchronous, loosely coupled manner. This chapter will introduce you to ActiveMQ.

ActiveMQ is an open source, Java Message Service (JMS) 1.1–compliant, message-oriented middleware (MOM) from the Apache Software Foundation that provides high availability, performance, scalability, reliability, and security for enterprise messaging. ActiveMQ is licensed using the Apache License, one of the

most liberal and business-friendly Open Source Initiative (OSI)–approved licenses available. Because of the Apache License, anyone can use or modify ActiveMQ without any repercussions for the redistribution of changes. This is a critical point for businesses who use ActiveMQ in a strategic manner. As described later in chapter 2, the job of a MOM is to mediate events and messages among distributed applications, guaranteeing that they reach their intended recipients. So it's vital that a MOM be highly available, performant, and scalable.

The goal of ActiveMQ is to provide standards-based, message-oriented application integration across as many languages and platforms as possible. ActiveMQ implements the JMS spec and offers dozens of additional features and value on top of this spec. These additional features will be introduced and discussed in detail throughout this book.

Your first steps with ActiveMQ are important to your success in using it for your own work. To the novice user, ActiveMQ may appear to be daunting, and yet to the seasoned hacker, it might be easier to understand. This chapter will walk you through the task of becoming familiar with ActiveMQ in a simple manner. You'll not only gain a high-level understanding of the ActiveMQ feature set, but you'll also be taken through a discussion of why and where to use ActiveMQ in your application development. Then you'll be prepared enough to install and begin using ActiveMQ.

1.1 ActiveMQ features

ActiveMQ provides an abundance of features created through hundreds of man-years of effort. The chapters in this book break down ActiveMQ into sets of features to focus on describing many of them. The following is a high-level list of some of the features that will be discussed throughout this book:

- *JMS compliance*—A good starting point for understanding the features in ActiveMQ is that ActiveMQ is an implementation of the JMS 1.1 spec. As discussed later in this chapter, the JMS spec provides important benefits and guarantees, including synchronous or asynchronous message delivery, once-and-only-once message delivery, message durability for subscribers, and much more. Adhering to the JMS spec for such features means that no matter what JMS provider is used, the same base set of features will be made available.
- *Connectivity*—ActiveMQ provides a wide range of connectivity options, including support for protocols such as HTTP/S, IP multicast, SSL, STOMP, TCP, UDP, XMPP, and more. Support for such a wide range of protocols equates to more flexibility. Many existing systems utilize a particular protocol and don't have the option to change, so a messaging platform that supports many protocols lowers the barrier to adoption. Though connectivity is important, the ability to closely integrate with other containers is also important. Chapter 4 addresses both the transport connectors and the network connectors in ActiveMQ.
- *Pluggable persistence and security*—ActiveMQ provides multiple flavors of persistence and you can choose between them. Also, security in ActiveMQ can be

completely customized for the type of authentication and authorization that's best for your needs. For example, ActiveMQ offers its own style of ultra-fast message persistence via KahaDB, but also supports standard JDBC-accessible databases. ActiveMQ also supports its own simple style of authentication and authorization using properties files as well as standard JAAS login modules. These two topics are discussed in chapters 5 and 6.

- *Building messaging applications with Java*—The most common route with ActiveMQ is with Java applications for sending and receiving messages. This task entails use of the JMS spec APIs with ActiveMQ and is covered in chapter 7.
- *Integration with application servers*—It's common to integrate ActiveMQ with a Java application server. Chapter 8 provides examples of integrating with some of the most popular application servers, including Apache Tomcat, Jetty, Apache Geronimo, and JBoss.
- *Client APIs*—ActiveMQ provides client APIs for many languages besides just Java, including C/C++, .NET, Perl, PHP, Python, Ruby, and more. This opens the door to opportunities where ActiveMQ can be utilized outside of the Java world. Many other languages also have access to all of the features and benefits provided by ActiveMQ through these various client APIs. Of course, the ActiveMQ broker still runs in a Java VM, but the clients can be written using any of the supported languages. Client connectivity to ActiveMQ is covered in chapter 9.
- *Broker clustering*—Many ActiveMQ brokers can work together as a federated network of brokers for scalability purposes. This is known as a *network of brokers* and can support many different topologies. This topic is covered in chapter 10.
- *Many advanced broker features and client options*—ActiveMQ provides many sophisticated features for both the broker and the clients connecting to the broker. ActiveMQ also supports the use of Apache Camel within the broker's XML configuration file. These features are discussed in chapters 11 and 12.
- *Dramatically simplified administration*—ActiveMQ is designed with developers in mind. As such, it doesn't require a dedicated administrator because it provides easy-to-use yet powerful administration features. There are many ways to monitor different aspects of ActiveMQ, including via JMX using tools such as JConsole or the ActiveMQ web console, by processing the ActiveMQ advisory messages, by using command-line scripts, and even by monitoring various types of logging. This is all covered in chapter 14.

This is just a taste of the features offered by ActiveMQ. As you can see, these topics will be addressed through the rest of the chapters of the book. For demonstration purposes, a couple of simple examples will be carried throughout and these examples will be introduced in chapter 3. But before we take a look at the examples, and given the fact that you've been presented with numerous different features, we're sure you have some questions about why you might use ActiveMQ.

1.2 Using ActiveMQ: why and when?

Back around 2003, a group of open source developers got together to form Apache Geronimo. In doing so, they discovered that there was no good message broker available that utilized a BSD-style license. Geronimo needed a JMS implementation for reasons of Java EE compatibility, so a few of the developers starting discussing the possibilities. Possessing vast experience with commercial MOMs and even having built a few MOMs themselves previously, these developers set out to create the next great open source message broker. Additional inspiration for ActiveMQ came from the fact that most of the MOMs in the market were commercial, closed source, and were costly to buy and support. The commercial MOMs were popular with businesses, but some businesses couldn't afford the steep costs required. This further increased the motivation to build an open source alternative. There was clearly a market available for an open source MOM using an Apache License. What evolved over time is Apache ActiveMQ.

ActiveMQ was meant to be used as the JMS spec intended, for remote communications between distributed applications. To better understand what this means, the best thing to do is look at a few of the ideas behind distributed application design, specifically communications.

1.2.1 Loose coupling and ActiveMQ

ActiveMQ provides the benefits of loose coupling for application architecture. Loose coupling is commonly introduced into an architecture to mitigate the classic tight coupling of Remote Procedure Calls (RPC). Such a loosely coupled design is considered to be asynchronous, where the calls from either application have no bearing on one another; there's no interdependence or timing requirements. The applications can rely upon ActiveMQ's ability to guarantee message delivery. Because of this, it's often said that applications sending messages just fire-and-forget—they send the message to ActiveMQ and aren't concerned with how or when the message is delivered. In the same manner, the consuming applications have no concern with where the messages originated or how they were sent to ActiveMQ. This is an especially powerful benefit in heterogeneous environments, allowing clients to be written using different languages and even possibly different wire protocols. ActiveMQ acts as the middleman, allowing heterogeneous integration and interaction in an asynchronous manner. More on this in the next section.

When considering distributed application design, coupling is important. *Coupling* refers to the interdependence of two or more applications or systems. An easy way to think about coupling is to consider the effect of changes to any application in the system: the implications across the other applications in the architecture as features are added. Do changes to one application force changes to other applications involved? If the answer is yes, then those applications are tightly coupled. But if one application can be changed without affecting other applications, then those applications are more loosely coupled. The overall lesson here is that tightly coupled applications are

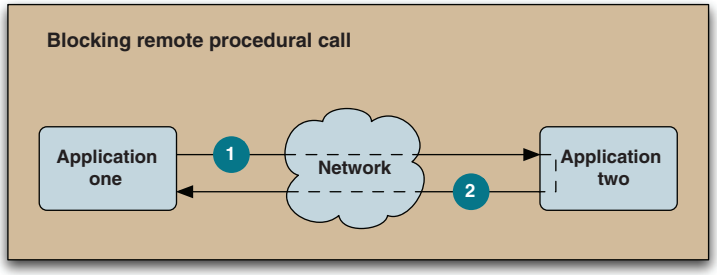

Figure 1.1 Two tightly coupled applications using remote procedure calls to communicate

more difficult to maintain compared to loosely coupled applications. Said another way, loosely coupled applications can easily deal with unforeseen changes.

Technologies such as those discussed in chapter 2 (COM, CORBA, DCE, and EJB) using RPC are considered to be tightly coupled. Using RPC, when one application calls another application, the caller is blocked until the callee returns control to the caller. The diagram in figure 1.1 depicts this concept.

The caller (application one) in figure 1.1 is blocked until the callee (application two) returns control. Many system architectures use RPC and are successful. But there are numerous disadvantages to such a tightly coupled design: most notable is the higher amount of maintenance required, since even small changes ripple throughout the system architecture. Correct timing between the two applications is a necessity. Both applications must be available at the same time for the request from application one to reach application two (1), and for the response to travel from application two to application one (2). Such timing requirements can be cumbersome, causing the application to be fragile. Compare such a tightly coupled design with a design where two applications are completely unaware of one another such as that depicted in figure 1.2.

Application one in figure 1.2 sends a message to the MOM in a one-way fashion. Then, possibly sometime later, application two receives a message from the MOM, in a one-way fashion. Neither application has any knowledge that the other even exists, and there's no timing between the two applications. This one-way style of interaction results in much lower maintenance because changes in one application have little to no effect on the other application. For these reasons, loosely coupled applications offer big advantages over tightly coupled architectures when considering distributed application design. This is where ActiveMQ enters the picture.

Consider the changes necessary when an application must move to a new location. This can happen when new hardware is introduced or the application needs to be moved. With a tightly coupled system design, such movement is difficult because all segments of the application must experience an outage. With an application designed using loose coupling, different segments of the system can be moved independent of one another. Consider a scenario where there are multiple instances of application A and multiple instances of application B, where each instance resides on a different machine. ActiveMQ is installed on still another machine independent of either

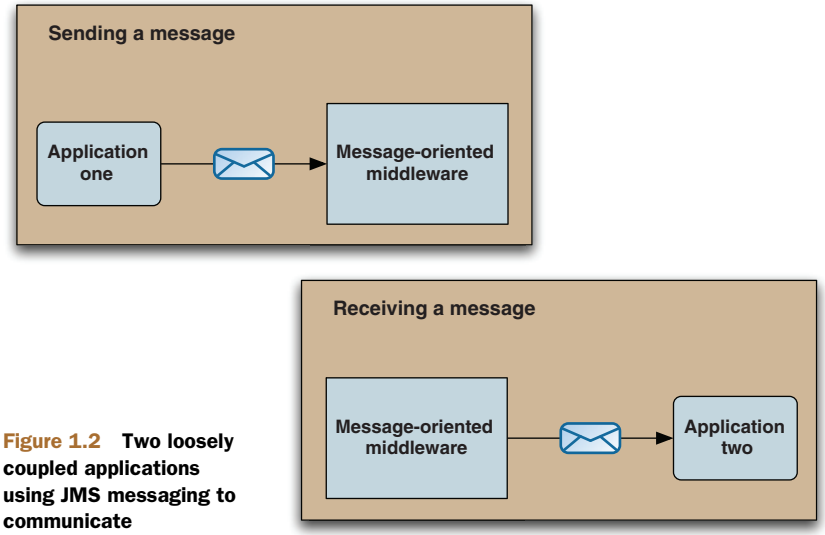

Figure 1.2 Two loosely coupled applications using JMS messaging to communicate

application A or application B. In this scenario, any one of the application A or application B instances can be moved around without affecting one another. In fact, multiple instances of ActiveMQ could be used in what's known as a *network of brokers* configuration. This would allow the ActiveMQ instances to be moved around without affecting either application A or application B. This means that any segment of this architecture can be taken down for maintenance at any time without taking down the entire system. More details about this are available in chapter 10.

So ActiveMQ provides an incredible amount of flexibility in application architecture, allowing the concepts surrounding loose coupling to become a reality. ActiveMQ also supports the request/reply paradigm of messaging if a completely asynchronous style of messaging isn't possible for a given use case. But when should ActiveMQ be used to introduce these benefits?

1.2.2 When to use ActiveMQ

There are many occasions where ActiveMQ and asynchronous messaging can have a meaningful impact on a system architecture. Here are just a few example scenarios:

- *Heterogeneous application integration*—The ActiveMQ broker is written using the Java language, so naturally a Java client API is provided. But ActiveMQ also provides clients for C/C++, .NET, Perl, PHP, Python, Ruby, and a few other languages. This is a huge advantage when considering how you might integrate applications written in different languages on different platforms. In cases such as this, the various client APIs make it possible to send and receive messages via ActiveMQ no matter what language is used. In addition to the cross-language capabilities provided by ActiveMQ, the ability to integrate such applications without the use of RPC is definitely a big benefit because messaging truly helps to decouple the applications.

- *As a replacement for RPC*—Applications using RPC-style synchronous calls are widespread. Consider that the vast majority of client-server applications use RPC including ATMs, most web applications, credit card systems, point-of-sale systems, and more. Even though many of these systems are successful, conversion to the use of asynchronous messaging can bring about benefits without giving up the guarantee of a response. Systems that rely upon synchronous requests typically have a limited ability to scale because eventually requests will begin to back up, thereby slowing the whole system. Instead of experiencing this type of a slowdown, using asynchronous messaging, additional message receivers can be easily added so that messages are consumed concurrently and therefore handled faster. This, of course, assumes that your applications can be decoupled.
- *To loosen the coupling between applications*—As already discussed, tightly coupled architectures can be problematic for many reasons, especially if they're distributed. Loosely coupled architectures, on the other hand, exhibit fewer dependencies, making them better at handling unforeseen changes. Not only will a change to one component in the system not ripple across the entire system, but component interaction is also dramatically simplified. Instead of using a synchronous scheme for component interaction (where one method calls another and the caller waits for a response from the callee), components utilize asynchronous communications (where they simply send a message without waiting for a response—also known as *fire-and-forget*). Such loose coupling throughout a system can lead to what's known as an *event-driven architecture* (EDA).
- *As the backbone of an event-driven architecture*—The decoupled, asynchronous style of architecture described in the previous point allows the broker itself to scale much further and handle considerably more clients via tuning, additional memory allocation, and so on (known as *vertical scalability*) instead of only relying upon the ability of the number of broker nodes to be increased to handle many more clients (known as *horizontal scalability*). Consider an incredibly high-traffic e-commerce site such as Amazon. When a user makes a purchase on Amazon, there are quite a few separate stages through which that order must travel including order placement, invoice creation, payment processing, order fulfillment, shipping, and more. But when a user actually places an order, the user is immediately taken to a page stating, "Thanks for your order." Not only that, but without delay, the user also receives an email stating that the order was received. The order placement process that's employed by Amazon is a good example of the first stage in a much larger set of asynchronous processes. Each stage of the order is handled discretely by a separate service. When the user places the order, there's a synchronous call to submit the order, but the entire order process doesn't take place behind a synchronous call via the web browser. Instead, the order is accepted and acknowledged immediately. The rest of the steps in the process are handled asynchronously. If a problem occurs that

prevents the process from proceeding, the user is notified via email. Such asynchronous processes are what afford massive scalability and high availability.

- *To improve application scalability*—Many applications utilize an event-driven architecture in order to provide massive scalability including such domains as e-commerce, government, manufacturing, and online gaming, just to name a few. By separating an application along lines in the business domain using asynchronous messaging, many other possibilities begin to emerge. Consider the ability to design an application using a service for a specific task. This is the backbone of service-oriented architecture (SOA). Each service fulfills a discrete function and only that function. Then applications are built through the composition of these services, and the communication among services is achieved using asynchronous messaging and eventual consistency. This style of application design makes it possible to introduce such concepts as *complex event processing* (CEP). Using CEP, the interactions among the components in a system are tracked for further analysis. Such possibilities are truly endless when you consider that asynchronous messaging is simply adding a level of indirection between components in a system.

Now that you've been offered some examples of where to use ActiveMQ, it's time to install ActiveMQ and begin using it.

1.3 Getting started with ActiveMQ

Getting started with ActiveMQ isn't difficult. You simply need to start up the broker and make sure that it's capable of accepting connections and sending messages. ActiveMQ comes with some simple examples that will help you with this task, but first we need to install Java and download ActiveMQ.

In this section, you'll download and install the Java SE, download and install ActiveMQ, examine the ActiveMQ directory, and start up ActiveMQ for the first time.

1.3.1 Downloading and installing the Java SE

ActiveMQ requires a minimum of the Sun Java SE 1.5, though 1.6 is preferred. This must be installed prior to attempting this section. If you don't have the Sun J2SE installed and you're using Linux, Solaris, or Windows, download and install it from the following URL: http://www.oracle.com/technetwork/java/javase/downloads/index.html.

If you're using Mac OS X, you should already have Java installed. But just in case you don't, you can grab it from the following URL: http://developer.apple.com/java/download/.

Once you have the Java SE installed, you'll need to test that it is set up correctly. To do this, open a terminal or command line and enter the command shown in the following listing.

Listing 1.1 Check the Java version

```
[~]$ java version "1.6.0_20"
Java(TM) SE Runtime Environment (build 1.6.0_20-b02-279-10M3065)
Java HotSpot(TM) 64-Bit Server VM (build 16.3-b01-279, mixed mode)
```

Your output may be slightly different depending on the operating system you're using, but the important part is that there's output from the Java SE. The command tells us two things: that the J2SE is installed correctly and that Java version 1.6 is being used. If you don't see similar output, then you'll need to rectify this situation before moving on to the next section.

Downloading and Installing Ant

Ant will be used to build and run the examples that ship with ActiveMQ. Ant is available from the Apache Ant website at the following URL: http://ant.apache.org/bindownload.cgi.

Click on the link to the appropriate archive for your operating system (the tarballs are for Linux and Unix; the zip is for Windows). Please follow the instructions for intalling Ant at this URL: http://ant.apache.org/manual/install.html. Make sure to set up the $ANT_HOME environment variable and to put $ANT_HOME/bin in the $PATH environment variable. Once Ant is properly installed, you should be able to run the following command from a terminal to see the Ant version:

```
$ ant -version
Apache Ant version 1.8.1 compiled on April 30 2010
```

You may be using a slightly different version of Ant, but that shouldn't matter. Once Ant outputs its version as shown above, you know that both the Java SE and Ant have been installed properly.

1.3.2 Downloading ActiveMQ

ActiveMQ is available from the Apache ActiveMQ website at the following URL: http://activemq.apache.org/download.html.

Click on the link to the 5.4.1 release and you'll find both tarball and zip formats available (the tarball is for Linux and Unix; the zip is for Windows). Once you've downloaded one of the archives, expand it and you're ready to move along. Once you get to this point, you should have the Java SE all set up and working correctly, and you're ready to take a peek at the ActiveMQ directory.

1.3.3 Examining the ActiveMQ directory

From the command line, move into the apache-activemq-5.4.1 directory and enter the command shown here.

Listing 1.2 List the contents of the ActiveMQ directory

```
[apache-activemq-5.4.1]$ ls -1
LICENSE
NOTICE
```

```
README.txt
WebConsole-README.txt
activemq-all-5.4.1.jar
bin
conf
data
docs
example
lib
user-guide.html
webapps
```

The contents of the directory are fairly straightforward:

- *LICENSE*—A file required by the Apache Software Foundation (ASF) for legal purposes; contains the licenses of all libraries used by ActiveMQ.
- *NOTICE*—Another ASF-required file for legal purposes; it contains copyright information of all libraries used by ActiveMQ.
- *README.txt*—A file containing some URLs to documentation to get new users started with ActiveMQ.
- *WebConsole-README.txt*—Contains information about using the ActiveMQ web console.
- *activemq-all-5.4.1.jar*—A jar file that contains all of ActiveMQ; it's placed here for convenience if you need to grab it and use it.
- *bin*—The bin directory contains binary/executable files for ActiveMQ; the startup scripts live in this directory.
- *conf*—The conf directory holds all the configuration information for ActiveMQ.
- *data*—The data directory is where the log files and message persistence data is stored.
- *docs*—Contains a simple index.html file referring to the ActiveMQ website.
- *example*—The ActiveMQ examples; these are what we'll use shortly to test out ActiveMQ quickly.
- *lib*—The lib directory holds all the libraries needed by ActiveMQ.
- *user-guide.html*—A brief guide to starting up ActiveMQ and running the examples.
- *webapps*—The webapps directory holds the ActiveMQ web console and some other web-related demos.

The next task is to start up ActiveMQ and verify it using the examples.

1.3.4 Starting up ActiveMQ

After downloading and expanding the archive, ActiveMQ is ready for use. The binary distribution provides a basic configuration to get you started easily and that's what we'll use in the examples. So start up ActiveMQ now as shown next.

Listing 1.3 Start up ActiveMQ

```
$ ./bin/activemq console
INFO: Using default configuration
(you can configure options in one of these file: /etc/default/activemq
/Users/bsnyder/.activemqrc)
INFO: Invoke the following command to create a configuration file
./bin/activemq setup [ /etc/default/activemq | /Users/bsnyder/.activemqrc ]
INFO: Using java '/System/Library/Frameworks/JavaVM.framework/Home/bin/java'
INFO: Starting in foreground, this is just for debugging purposes
(stop process by pressing CTRL+C)
Java Runtime: Apple Inc. 1.6.0_20
/System/Library/Frameworks/JavaVM.framework/Versions/1.6.0/Home
Heap sizes: current=258880k free=253105k max=258880k
JVM args: -Xms256M -Xmx256M
-Dorg.apache.activemq.UseDedicatedTaskRunner=true
-Djava.util.logging.config.file=logging.properties
-Dcom.sun.management.jmxremote
-Dactivemq.classpath=/Users/bsnyder/amq/apache-activemq-5.4.1/conf;
-Dactivemq.home=/Users/bsnyder/amq/apache-activemq-5.4.1
-Dactivemq.base=/Users/bsnyder/amq/apache-activemq-5.4.1
ACTIVEMQ_HOME: /Users/bsnyder/amq/apache-activemq-5.4.1
ACTIVEMQ_BASE: /Users/bsnyder/amq/apache-activemq-5.4.1
Loading message broker from: xbean:activemq.xml
WARN | destroyApplicationContextOnStop parameter is deprecated,
please use shutdown hooks instead
INFO | PListStore:/Users/bsnyder/amq/apache-activemq-5.4.1/data/localhost/
tmp_storage started INFO | Using Persistence Adapter:
KahaDBPersistenceAdapter[/Users/bsnyder/amq/apache-activemq-5.4.1/data/
     kahadb]
INFO | KahaDB is version 2
INFO | Recovering from the journal ...
INFO | Recovery replayed 1 operations from the journal in 0.029 seconds.
INFO | ActiveMQ 5.4.1 JMS Message Broker (localhost) is starting
...
INFO | ActiveMQ Console at http://0.0.0.0:8161/admin
INFO | Initializing Spring root WebApplicationContext
INFO | Connector vm://localhost Started
INFO | Camel Console at http://0.0.0.0:8161/camel
INFO | ActiveMQ Web Demos at http://0.0.0.0:8161/demo
INFO | RESTful file access application at http://0.0.0.0:8161/fileserver
INFO | Started SelectChannelConnector@0.0.0.0:8161
```

NOTE The examples in the listings in this book were developed on Mac OS X, a Unix operating system. For readers who are using Windows, simply do not use the 'console' argument from any of the examples. To run the example command shown in Listing 1.3 above on Windows, use the following command from the command prompt:

```
C:\apache-activemq-5.4.1>bin\activemq
```

Please note that the command used to start up ActiveMQ on Windows should not contain the 'console' argument. This applies to all the example listings in the book.

This command starts up the ActiveMQ broker and some of its connectors to expose it to clients via a few protocols, namely, TCP, SSL, STOMP, and XMPP. Just be aware that ActiveMQ has started and is available to clients via TCP on port 61616. This is all configurable and will be discussed later in chapter 4. For now, the preceding output tells you that ActiveMQ is up and running and ready for use. Now it's ready to begin handling some messages. The best way to begin sending and receiving messages is by using some of the examples that come with ActiveMQ. The next section walks you through this in a step-by-step manner.

1.4 Running your first examples with ActiveMQ

The previous section walked you through starting up ActiveMQ in one terminal. For verification of this, you should open two more terminals to run the ActiveMQ examples. In the second terminal, move into the example directory and look at its contents as shown in the following listing.

Listing 1.4 List the contents of the ActiveMQ example directory

```
[apache-activemq-5.4.1]$ cd ./example/
bsnyder@mongoose [example]$ ls -1
build.xml
conf
perfharness
ruby
src
transactions
```

The example directory contains a few different items. Here's a quick description of each item in that directory:

- *build.xml*—An Ant build configuration for use with the Java examples.
- *conf*—The conf directory holds configuration information for use with the Java examples.
- *perfharness*—The perfharness directory contains a script for running the IBM JMS performance harness against ActiveMQ.
- *ruby*—The ruby directory contains some examples of using ActiveMQ with Ruby and the STOMP connector.
- *src*—The src directory is where the Java examples live; this directory is used by the build.xml.
- *transactions*—The transactions directory holds an ActiveMQ implementation of the TransactedExample from Sun's JMS Tutorial.

Using the second terminal, start up a JMS consumer as shown here.

Listing 1.5 Start up the ActiveMQ consumer example

```
[example]$ ant consumer
Buildfile: build.xml

init:
```

```
compile:

consumer:
     [echo] Running consumer against server at $url =
tcp://localhost:61616 for subject $subject = TEST.FOO
     [java] Connecting to URL: tcp://localhost:61616
     [java] Consuming queue: TEST.FOO
     [java] Using a non-durable subscription
     [java] Running 1 parallel threads
     [java] [Thread-2] We are about to wait until we consume:
2000 message(s) then we will shutdown
```

The command compiles the Java examples and starts up a simple JMS consumer. As you can see from the output, this consumer is

- Connecting to the broker using the TCP protocol (tcp://localhost:61616)
- Watching a queue named TEST.FOO
- Using nondurable subscription
- Waiting to receive 2000 messages before shutting down

Basically, the JMS consumer is connected to ActiveMQ and waiting for messages. Now you can send some messages to the TEST.FOO destination.

In the third terminal, move into the example directory and start up a JMS producer as shown below. This will immediately begin to send messages.

Listing 1.6 Start up the ActiveMQ producer example

```
[example]$  ant producer
Buildfile: build.xml

init:

compile:

producer:
     [echo] Running producer against server at $url =
tcp://localhost:61616 for subject $subject = TEST.FOO
     [java] Connecting to URL: tcp://localhost:61616
     [java] Publishing a Message with size 1000 to queue: TEST.FOO
     [java] Using non-persistent messages
     [java] Sleeping between publish 0 ms
     [java] Running 1 parallel threads
     [java] [Thread-2] Sending message: 'Message: 0 sent at: Thu Oct 14
21:24:07 MDT 2010  ...'
     [java] [Thread-2] Sending message: 'Message: 1 sent at: Thu Oct 14
21:24:07 MDT 2010  ...'
     [java] [Thread-2] Sending message: 'Message: 2 sent at: Thu Oct 14
21:24:07 MDT 2010  ...'
```

Although the output has been truncated for readability, the command starts up a simple JMS producer and you can see from the output that it

- Connects to the broker using the TCP connector (tcp://localhost:61616)
- Publishes messages to a queue named TEST.FOO
- Uses nonpersistent messages
- Doesn't sleep between publishing messages

Once the JMS producer is connected, it then sends 2,000 messages and shuts down. This is the number of messages the consumer is waiting to consume before it shuts down. So as the messages are being sent by the producer in terminal three, flip back to terminal two and watch the JMS consumer as it consumes those messages. Here's the output you'll see in terminal two:

```
     [java] [Thread-2] Received: 'Message: 0 sent at: Thu Oct 14 21:23:56
MDT 2010  ...' (length 1000)
     [java] [Thread-2] Received: 'Message: 1 sent at: Thu Oct 14 21:23:56
MDT 2010  ...' (length 1000)
     [java] [Thread-2] Received: 'Message: 2 sent at: Thu Oct 14 21:23:56
MDT 2010  ...' (length 1000)
...
     [java] [Thread-2] Received: 'Message: 1999 sent at: Thu Oct 14 21:23:56
MDT 2010  ...' (length 1000)
```

Again, the output has been truncated for brevity but this doesn't change the fact that the consumer received 2,000 messages and shut itself down. At this time, both the consumer and the producer should be shut down, but the ActiveMQ broker is still running in the first terminal. Take a look at the first terminal again and you'll see that ActiveMQ appears to not have budged at all. This is because the default logging configuration doesn't output anything beyond what's absolutely necessary. If you'd like to tweak the logging configuration to output more information as messages are sent and received, you can do so. Logging will be covered further in chapter 14.

So what did you learn here? Through the use of the Java examples that come with ActiveMQ, it has been proven that the broker is up and running and can mediate messages. This doesn't seem like much but it's an important first step. If you were able to successfully run the Java examples, then you know that you have no networking problems on the machine you're using and you know that ActiveMQ is behaving properly. If you were unable to successfully run the Java examples, then you'll need to troubleshoot the situation. If you need some help, heading over to the ActiveMQ mailing lists is the best way to find help. These examples are just to get you started but can be used to test many scenarios. Throughout the rest of the book, some different examples surrounding a couple of common use cases will be used to demonstrate ActiveMQ and its features. These examples are explained further in chapter 3.

1.5 Summary

ActiveMQ is a versatile, easy-to-use messaging middleware. You learned about some of the ActiveMQ features that will be covered throughout this book and about some scenarios where ActiveMQ can be applied. The scenarios introduced in this chapter are real-world use cases that are deployed in businesses throughout the world. The JMS spec was designed for use in business applications with these scenarios in mind. For those who aren't familiar with the JMS spec, or even those who'd like a refresher on the topic, the next chapter covers enterprise messaging and provides an overview of JMS. If you're already fluent in these two topics, you can skip ahead to chapter 3 to explore the examples for the book.

ENTERPRISE JAVA

ActiveMQ IN ACTION

Snyder • Bosanac • Davies

The Apache ActiveMQ message broker is an open source implementation of the Java Message Service spec. It makes for a reliable hub in any message-oriented enterprise application and integrates beautifully with Java EE containers, ESBs, and other JMS providers.

ActiveMQ in Action is all you'll need to master ActiveMQ. It starts from the anatomy of a JMS message and moves quickly through connectors, message persistence, authentication, and authorization. By following a running example (a stock portfolio app), you'll pick up the best practices distilled by the authors from their long and deep involvement with this technology.

What's Inside

- How to design message-based apps
- How to implement EI patterns using Camel
- How to administer ActiveMQ
- How to integrate with Geronimo, JBoss, Spring, and more

This book requires a working knowledge of Java, but no previous experience with ActiveMQ or other message brokers is needed.

Bruce Snyder is a co-founder of Apache Geronimo, a committer for ActiveMQ, Camel, and ServiceMix, and a member of various JCP expert groups. **Dejan Bosanac** is an ActiveMQ committer. **Rob Davies** is a co-founder of ActiveMQ, ServiceMix, and Camel.

For online access to the authors and a free ebook for owners of this book, go to manning.com/ActiveMQinAction

"Covers everything you need to know about ActiveMQ."
—Pratik Patel, AT&T

"A vital resource."
—John Merryman, Yodle

"Complete and comprehensive, a must-have resource."
—Rod Biresch
Chariot Solutions

"Authors have in-depth knowledge of ActiveMQ."
—Roberto J. Rojas
Chariot Solutions

"Covers the basics, and then goes way beyond."
—Jeff Davis
Author of *Open Source SOA*

ISBN 13: 978-1-933988-94-8
ISBN 10: 1-933988-94-0
54499
9 781933 988948

MANNING $44.99 / Can $51.99 [INCLUDING eBOOK]