The Pragmatic Programmers

Deploying with JRuby 9k

Deliver Scalable Web Apps Using the JVM

Joe Kutner

Edited by Brian P. Hogan

Early praise for *Deploying with JRuby 9K*

Joe has pulled together a great collection of deployment knowledge from his years of experience building and supporting JRuby applications. He's an expert on this subject and *Deploying with JRuby 9k* is the definitive text for getting JRuby applications up and running.

➤ **Charles Oliver Nutter**
JRuby co-lead

Deploying with JRuby 9k answers the most frequently asked questions about real-world use of JRuby. Whether you're coming to JRuby from Ruby or Java, Joe fills in all the gaps you'll need to deploy JRuby with confidence.

➤ **Tom Enebo**
JRuby co-lead

I've been working with JRuby for years and I still learned several immediately actionable steps to improve the performance and maintenance of real-world JRuby apps.

➤ **Matt Margolis**
director, application development at Getty Images

Deploying with JRuby 9k is full of practical and actionable advice about how to get the most benefit out of the JVM when running your Ruby app on JRuby.

➤ **Chris Seaton**
Oracle Labs and JRuby contributor

Deploying with JRuby 9k is the essential guide for anyone building Ruby applications on the JVM. It's loaded with tips, tricks, and best practices that newcomers and experts can learn from.

➤ **Terence Lee**
Ruby task force member at Heroku

As a developer of MRI, I get super jealous reading about the JVM ecosystem and tooling. With this book, Joe has finally made that ecosystem approachable for JRuby applications.

➤ **Zachary Scott**
Ruby-core member and maintainer of Sinatra

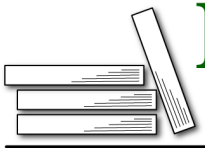
Deploying with JRuby 9k

Deliver Scalable Web Apps Using the JVM

Joe Kutner

The Pragmatic Bookshelf

Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

The team that produced this book includes:

Brian P. Hogan (editor)
Potomac Indexing, LLC (index)
Linda Recktenwald (copyedit)
Gilson Graphics (layout)
Janet Furlow (producer)

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-68050-169-8
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—July 2016

Contents

Acknowledgments

Writing a book is a lot like writing code. You need to know the rules, recognize patterns, and occasionally know when to break the rules. Both writing and coding are crafts. And like with any craft, you improve by getting advice from more experienced professionals and being critiqued by your peers. I'm so fortunate to have had this kind of help.

I'm inexpressibly thankful to those who reviewed this book prior to its publication. I was humbled by the attention to detail and wise feedback they provided in making it a finished product. Thank you, Jeff Holland, Margaret Le, Matt Margolis, Jay McGaffigan, Chris Seaton, and Tim Uckun. I consider you all to be my friends!

I'd also like to thank the staff at the Pragmatic Bookshelf: Susannah Pfalzer, Dave Thomas, Andy Hunt, and probably a whole bunch of other people I don't know about.

Above all, thank you, Brian P. Hogan, my editor. This is our fourth endeavor together, and as usual I've become a better writer because of it. I look forward to working on future projects together.

I must also thank the creators of the technologies I've written about. This book would not have been possible without your hard work. Thank you, Charles Nutter, Thomas Enebo, Karol Buček, Christian Meier, Chris Seaton, and the rest of the JRuby team. You're the most amazing group in all of the open source world. I owe you all my deepest gratitude and a free beverage.

Finally, I'd like to thank my wife and son. I could not have completed this project without your love and support.

Preface

Your website has just crashed, and you're losing money. The application is built on Rails, runs on MRI, and is served up with Unicorn and Apache. Having this kind of infrastructure means you're managing more processes than you can count on two hands.

The background jobs are run with Resque,[1] the scheduled jobs are run with `cron`, and the long-running jobs use Ruby daemons,[2] which are monitored by monit because they tend to crash.[3] It's going to take some time to figure out which component is the culprit because you have no centralized management interface. Standing up a new server will take almost as long because the infrastructure is so complex. But the website has to get back online if you're going to stay in business.

The problem I've just described is all too common. It has happened to everyone from small startups to large companies that use Rails to serve millions of requests. Their infrastructure is complex, and the myriad components are difficult to manage because they're heterogeneous and decentralized in nature. Even worse, Rubyists have become comfortable with this way of doing things, and some may think it's the only way of doing things. But that's not the case.

The recent growth and increased adoption of the Java Virtual Machine (JVM) as a platform for Ruby applications has opened many new doors. Deployment strategies that weren't possible with MRI Ruby are now an option because of the JVM's built-in management tools and support for native operating system threads. Ruby programmers can leverage these features by deploying their applications on JRuby.

It's common for Ruby programmers to think that JRuby deployment will look identical to deployment with MRI Ruby (that is, running lots of JVM processes

1. https://github.com/resque/resque
2. http://daemons.rubyforge.org/
3. http://mmonit.com/monit/

behind a load balancer and putting all asynchronous background jobs in a separate process). On the other hand, Java programmers tend to deploy JRuby applications the same way they deploy Java applications. This often requires lots of XML and custom build configurations, which negate many of the benefits of a more dynamic language such as Ruby. But there are much better options than both Ruby and Java programmers are used to.

In this book, you'll explore the most popular and well-supported methods for deploying JRuby. You have a surprising amount of flexibility in the processes and platforms to choose from, which allows Ruby and Java programmers to tailor their deployments to suit many different environments.

The No-Java-Code Promise

You won't have to write any Java code as you work your way through this book. That's not what this book is about. It's about deploying Ruby applications on the JVM. The technologies and tools you'll learn about in this book hide the XML and Java code from you. As the JRuby core developers like to say, "[They] write Java so you don't have to."[4]

You may want to include some Java code in your application. Or you may want to make calls to some Java libraries. That's entirely your choice. If you want to write your programs exclusively in Ruby and deploy them on the Java Virtual Machine—as so many of us do—then go ahead.

There are many reasons to deploy Ruby applications on the JVM, and using Java libraries and APIs is just one of them. In this book, you'll learn how to get the most out of the JVM without writing any Java code.

What's in This Book?

Over the course of this book, you're going to work on an application like the one described at the beginning of the preface. You'll port it to JRuby, add some new features, and simplify its infrastructure, which will improve its ability to scale.

The application's name is Twitalytics, and it's a rich Twitter client. (As you probably know, Twitter is a social networking website that's used to post short status updates, called *tweets*.) Twitalytics tracks an organization's tweets, annotates them, and performs analytic computations against data captured in those tweets to discover trends and make predictions. But it can't handle its current load.

4. http://vimeo.com/27494052

Twitalytics has several background jobs that are used to stream tweets into the application, perform analytics, and clean up the database as it grows. In addition, it receives a large volume of HTTP requests for traditional web traffic. But doing this on MRI means running everything in separate processes, which consumes more resources than its infrastructure can handle.

You'll begin working on the app in Chapter 1, *Getting Started with JRuby*, on page 1. You'll learn what makes JRuby a better deployment platform and why it's a good fit for this application. Then you'll extract a microservice from the Twitalytics monolith, port it to JRuby, and package it into an archive file with the Warbler gem. But before you can deploy it, you'll need to create an environment where it can run.

In Chapter 2, *Creating a Deployment Environment*, on page 17, you'll set up a containerization layer based on Docker and provision it with some essential components. You'll also learn how to automate this process to make it more reliable and reusable. You'll create a new server for each deployment strategy used in this book, and being able to reuse your configuration will save you time and prevent errors. In fact, this environment will apply not only to Twitalytics but to any JRuby deployment, so you're likely to reuse it on the job.

Once you've completed the production server setup, you'll be ready to deploy. You'll learn how JRuby deployment differs from the more common practice of traditional Ruby application deployment and how containerization technologies like Docker can simplify the process. In additional to using Docker, you'll deploy to the cloud on the Heroku platform as a service.

The Warbler gem gives you a quick way to get started with JRuby. But it's just a stepping-stone on your path to better performance. As the book progresses, you'll improve your deployment strategy by running Twitalytics on some other JRuby web servers.

The next chapter, Chapter 3, *Deploying a Rails Application*, on page 29, is dedicated to the Puma web server. Puma allows you to deploy applications much as you would with MRI-based Rails applications. But you'll find that JRuby reduces the complexity of this kind of deployment environment while increasing its reliability and portability. You'll deploy the Puma-based Rails app using both Docker and Heroku. The resulting architecture will be friendly and familiar to Rubyists.

But you still won't be making the most of what the JVM has to offer. To do that, you'll need a new kind of platform.

In Chapter 5, *Deploying JRuby in the Enterprise*, on page 73, you'll learn about a Ruby application server. You'll use TorqueBox, a server based on the popular JBoss application server, to run Twitalytics. This kind of deployment is unique when compared to traditional Ruby deployments because it provides a complete environment to run any kind of program, not just a web application. You'll learn how this eliminates the need for external processes. In the end, you'll have the most advanced deployment environment available to any Ruby application.

An overview of each strategy covered in this book is listed here:

	Warbler	Puma	TorqueBox
Internals	Jetty	Pure-Ruby	JBoss AS
Deployment type	WAR file	Traditional	Mixed
Docker deployment	Yes	Yes	Yes
Heroku deployment	Yes	Yes	Yes
Background jobs	No	No	Yes

Deciding on the right platform for each application is a function of these attributes. But getting an application up and running on one of these platforms is only a part of the job. You also need to keep it running. Fortunately, one of the many advantages of JRuby is the built-in JVM tooling.

Chapter 6, *Managing a JRuby Application*, on page 87 presents some tools for monitoring, managing, and configuring a deployed JRuby application. These tools are independent of any deployment strategy and can be used to monitor the memory consumption, performance, and uptime of any Java process. The information you gain from these tools can be used to improve the performance of JRuby, which you'll learn in Chapter 7, *Tuning a JRuby Application*, on page 109. You'll learn about different kinds of memory and the various knobs you can turn to optimize how the JVM allocates that memory. You'll even learn how to change garbage collectors and benchmark them.

In Chapter 8, *Monitoring JRuby in Production*, on page 123, you'll learn how to capture the same kind of metrics from a production runtime. You'll use some third-party apps to instrument your code, capture performance data, and log errors. Finally, Chapter 9, *Using a Continuous Integration Server*, on page 133 will introduce a tool for producing reliable and consistent deployments.

Twitalytics is a Rails application, and you'll use this to your advantage as you deploy it. But all of the server technologies you'll use work equally well with

any Rack-compliant framework (such as Sinatra[5]). In fact, the steps you'll use to package and deploy Twitalytics would be identical for these other frameworks. Warbler, Puma, and TorqueBox provide a few hooks that make deploying a Rails application more concise in some cases (such as automatically packaging bundled gems). But the workflow is the same.

When you encounter Rails-specific features in this book, be aware that this is only for demonstration purposes and not because the frameworks being used work exclusively with Rails. Rails works with these servers because it's Rack based.

What's Not in This Book?

This book won't teach you how to write code in the Ruby language. You'll write a bit of Ruby code in the course of this book, but you won't learn about specific features of the Ruby language. In particular, this book doesn't cover continuations, `ObjectSpace`, fibers, and other topics that have subtle differences when applied to JRuby. This book is specifically about *deploying* JRuby applications and how JRuby affects your production environments.

Other topics not addressed include zero-downtime deployments, database migrations, the asset pipeline, and content delivery networks (CDN). These are important aspects of Ruby web application development, but they're not notably different between MRI and JRuby. You can learn about these topics in books on the Ruby language and Rails. The same concepts will apply to JRuby.

Who Is This Book For?

This book is for programmers, system administrators, and DevOps[6] professionals who want to use JRuby to power their applications but aren't familiar with how this new platform will change their infrastructure.

You're not required to have any experience with JRuby. This book is written from the perspective of someone who is familiar with MRI-based Ruby deployments but wants a modern deployment strategy for their applications. Some of the concepts we'll discuss may be more familiar to programmers with Java backgrounds, but it's not required that you have any experience with Java or its associated technologies.

5. http://www.sinatrarb.com/
6. http://en.wikipedia.org/wiki/DevOps

Conventions

The examples in this book can be run on Linux, Mac, Windows, and many other operating systems. But some small changes to the command-line statements may be required for certain platforms.

We'll use notation from bash, which is the default shell on Mac OS X and many Linux distributions. The $ prompt will be used for all command-line examples. Windows command prompts typically use something like C:\> instead, so when you see a command like this

```
$ bundle install
```

you'll know not to type the dollar sign and to read it like this:

```
C:\> bundle install
```

Most commands will be compatible between Windows and bash systems (such as cd and mkdir). In the cases where they're not compatible, the appropriate commands for both systems will be spelled out. One case in particular is the rm command, which will look like this:

```
$ rm temp.txt
$ rm -rf tmp/
```

On Windows this should be translated to these two commands, respectively:

```
C:\> del temp.txt
C:\> rd /s /q tmp/
```

Another Unix notation that's used in this book is the ~ (tilde) to represent a user's home directory. When you see a command like this

```
$ cd ~/code/twitalytics
```

you can translate it to Windows 10 as this command:

```
C:\> cd C:\Users\yourname\code\twitalytics
```

On earlier versions of Windows, the user's home directory can be found in the Documents and Settings directory. You can also use the %USERPROFILE% environment variable. Its value is the location of the current user's profile directory.

Other than these minor notation changes, the examples in this book are compatible with Windows by virtue of the Java Virtual Machine.

Getting the Source Code

You're ready to set up the Twitalytics application. Start by downloading the source code from http://pragprog.com/titles/jkdepj2/source_code. Unpack the downloaded

file and put it in your home directory. This will create a `code` directory and inside that will be a `twitalytics` directory, which contains the baseline code for the application (in other words, the MRI-based code).

But you're not quite ready to run this code with JRuby. It needs to be ported first. You'll learn how to do that in the coming chapters.

Online Resources

Several online resources can help if you're having trouble setting up your environment or running any of the examples in this book.

For Java-related problems, the Java Community has forums[7] and numerous Java-related articles.

For JRuby-related problems, the official JRuby website[8] has links to several community outlets. The most useful of these are the mailing list[9] and the #jruby IRC channel on FreeNode.[10]

For TorqueBox-related problems, there are a mailing list,[11] extensive documentation,[12] and the #torquebox IRC channel on FreeNode.

7. https://community.oracle.com/community/java
8. http://jruby.org/community
9. https://github.com/jruby/jruby/wiki/MailingLists
10. http://freenode.net/
11. http://torquebox.org/community/mailing_lists/
12. http://torquebox.org/documentation/

CHAPTER 1

Getting Started with JRuby

JRuby is a high-performance platform that can scale to meet demand without the headaches of an MRI-based deployment. Those headaches are often the result of running a dozen or more processes on a single server that all need to be monitored, balanced, and occasionally restarted. JRuby avoids these problems by simplifying the architecture required to run an application. In this chapter, you're going to port a microservice to JRuby so that you can take advantage of this simplicity and scalability. But in order to run the microservice in production, you'll need a way to deploy it. For this, you'll use Warbler.[1]

Warbler is a gem used to package source code into an archive file you can deploy without the need for complicated configuration management scripts. This makes the process more flexible, portable, and faster.

In *Preface*, on page xi, you were introduced to Twitalytics, a Ruby on Rails app that needs help. Its infrastructure is too complex, and it can't handle the volume of requests the site is receiving. You don't have time to port the daemons and background jobs to a new framework, but you need to get one high-traffic HTTP service deployed on JRuby. If you can do that, you'll be able to handle lots of concurrent requests without hogging the system's memory. Later in the book, you'll consume this service from the main Rails app that makes up the bulk of Twitalytics.

Your time constraints make Warbler a great solution. It won't maximize your use of the JVM, but it will allow you to take advantage of the most important parts. You'll be able to service all of your site's web requests from a single process without changing much code. The drawback is that you'll have to

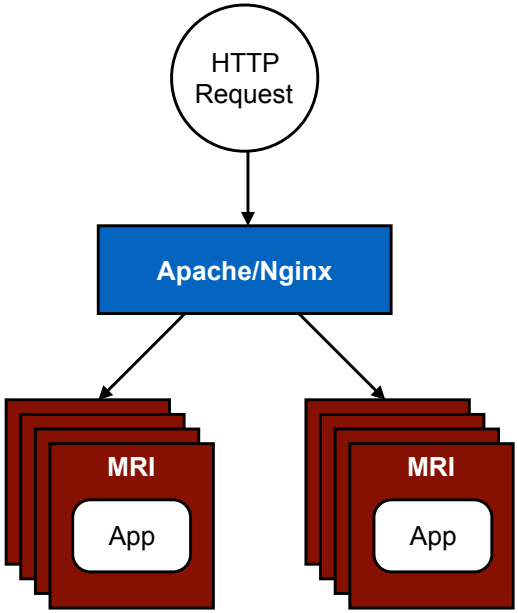
1. https://github.com/jruby/warbler

make changes to your deployment process, so there's much to learn. Let's begin by discussing why you might want to use JRuby in the first place.

What Makes JRuby So Great?

A production JRuby environment has fewer moving parts than traditional Ruby environments. This is possible because of the JVM's support for native operating system threads. Instead of managing dozens of processes, JRuby can use multiple threads of execution to do work in parallel. MRI has threads, but only one thread can execute Ruby code at a time. Recent versions of MRI, Rails, and other frameworks have improved the platform's ability to do some work in parallel. But true concurrency on MRI isn't possible, and the quest for better throughput has led to some complicated solutions.

Deployment with MRI usually requires a type of architecture that handles HTTP requests by placing either Apache[2] or a similar web server in front of a pool of application instances running in separate processes. An example of this is illustrated in the following figure.



2. http://httpd.apache.org/

There are many problems with this kind of architecture, and those problems have been realized by Twitter, GitHub, and countless others. They include the following:

Stuck processes Sometimes the processes will get into a stuck state and need to be killed by an external tool like god or Monit.

Slow restarts There's a lot of overhead in starting a new process. Several instances may end up fighting each other for resources if they're restarted at the same time.

Memory growth Each of the processes keeps its own copy of an application, along with Rails and any supporting gems, in memory. Each new instance means you'll also need more memory for the server.

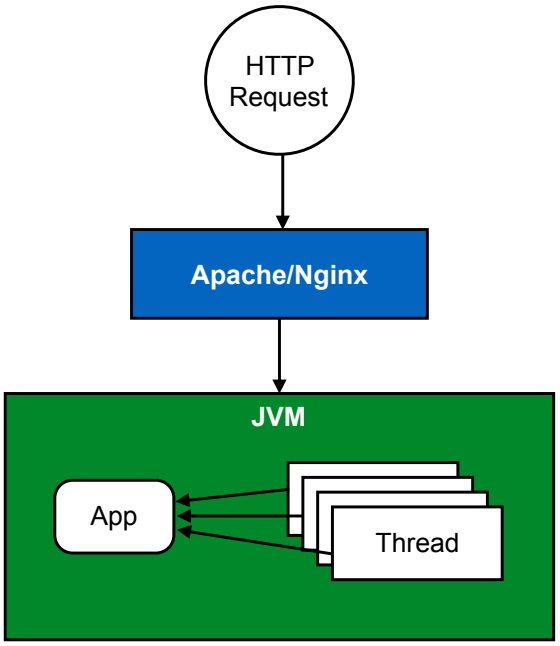
Several frameworks, such as Passenger and Unicorn, have improved on this model. But they all suffer from the same underlying constraint. A single MRI process has a scalability ceiling because Ruby code cannot execute in parallel. Other operations, such as I/O, can be done in parallel, and frameworks such as EventMachine and Celluloid leverage this to create event-based environments. But these frameworks also have an upper limit, and that's why some of their biggest implementers, including Logstash[3] and Venntro,[4] still choose to run them on JRuby.

JRuby enables a similar model to the one used by MRI but with only one JVM process. Inside this JVM process is a single application instance capable of handling all of a website's traffic. This works by allowing the platform to create many threads that run against the same application instance in parallel. You can create far more JVM threads than MRI processes because they're much lighter in weight. This model is illustrated in the figure on page 4.

Apache is included in the architecture diagram, but its role for a single instance is greatly reduced. It may be used to serve up static content and load balance a distributed cluster, but it won't need to distribute requests across multiple processes on a single machine.

In the coming chapters, you'll build an architecture like the one just described with each of the technologies you use. You'll start by using Warbler to package a simple Rack application, which will get you up and running quickly. But first, you'll need to install JRuby and a few other pieces of software.

3. https://www.elastic.co/products/logstash
4. http://dev.venntro.com/2013/07/euruko-2013-summary/



Preparing Your Environment

Four software packages are required to run the examples in the book. They're listed here along with the version needed:

- Java Development Kit (JDK) 8 (aka 1.8)
- JRuby 9.0.5.0
- Git 2.5
- Bundler 1.11

Java is supported in one form or another on a wide range of operating systems including Linux, Mac, Windows, and more. But the installation process will be different for each platform.

Installing Java

On Debian-based Linux platforms such as Ubuntu, the JVM can be installed with APT, like this:

```
$ sudo apt-get install openjdk-8-jdk
```

On Fedora, Oracle Linux, and Red Hat, the JVM is installed with the yum command, like this:

```
$ su -c "yum install java-1.8.0-openjdk"
```

For Mac OS X and Windows systems, you can download the latest version of Java 8 directly from Oracle's website.[5]

For Windows systems, you'll need to set the JAVA_HOME variable. (The exact path may vary.)

```
C:\> set JAVA_HOME="C:\Program Files\Java\jdk1.8.0_72"
```

In all cases, you can check that the JVM was installed correctly by running this command:

```
$ java -version
java version "1.8.0_72"
Java(TM) SE Runtime Environment (build 1.8.0_72-b15)
Java HotSpot(TM) 64-Bit Server VM (build 25.72-b15, mixed mode)
```

Now that the JVM is ready, you can put JRuby on your machine.

Installing JRuby and Bundler

The preferred method for installing JRuby on Unix and Linux systems requires the Ruby Version Manager (RVM). It's preferred not only because it makes JRuby easy to install but also because it treats JRuby just like any other Ruby platform. This allows you to use the ruby and gem commands without putting the j character in front of them or prefixing every other command with jruby -S. RVM is compatible only with bash systems, which don't include Windows. Installing JRuby on Windows will be described in a moment. If you're using a bash system, run this command to download the GPG key for RVM:

```
$ gpg --keyserver hkp://keys.gnupg.net \
  --recv-keys 409B6B1796C275462A1703113804BB82D39DC0E3
```

Then run this command to install RVM:

```
$ \curl -sSL https://get.rvm.io | bash -s stable
```

The leading backslash in the command disables any aliases you may have set in your shell and runs the curl binary directly.

You'll also need to reload your shell. The most dependable way to do this is by closing your terminal and opening a new one. In the new terminal, use RVM to install JRuby with this command:

5. http://www.oracle.com/technetwork/java/javase/downloads/index.html

```
$ rvm install jruby-9.0.5.0
Searching for binary rubies, this might take some time.
Unknown ruby string (do not know how to handle): jruby-9.0.5.0.
Found remote file /Users/jkutner/.rvm/archives/jruby-bin-9.0.5.0.tar.gz
Checking requirements for osx.
Requirements installation successful.
jruby-9.0.5.0 - #configure
Unknown ruby string (do not know how to handle): jruby-9.0.5.0.
jruby-9.0.5.0 - #download
jruby-9.0.5.0 - #validate archive
jruby-9.0.5.0 - #extract
jruby-9.0.5.0 - #validate binary
jruby-9.0.5.0 - #setup
jruby-9.0.5.0 - #gemset created /Users/jkutner/.rvm/gems/jruby-9.0.5.0...
jruby-9.0.5.0 - #importing gemset /Users/jkutner/.rvm/gemsets/jruby/gl...
jruby-9.0.5.0 - #generating global wrappers........
jruby-9.0.5.0 - #gemset created /Users/jkutner/.rvm/gems/jruby-9.0.5.0
jruby-9.0.5.0 - #importing gemsetfile /Users/jkutner/.rvm/gemsets/defa...
jruby-9.0.5.0 - #generating default wrappers........
Making gemset jruby-9.0.5.0 pristine.................................
Making gemset jruby-9.0.5.0@global pristine..........................
```

Set JRuby as the default Ruby on your platform by running this command:

```
$ rvm --default use jruby-9.0.5.0
Using /Users/jkutner/.rvm/gems/jruby-9.0.5.0
```

On Windows, you can install JRuby by downloading and running the Windows installer from the official JRuby website.[6]

If you're using any other kind of system or if you prefer not to use RVM, then JRuby can be installed manually with these three steps:

1. Download the JRuby binaries from the official website.[7]
2. Unpack the downloaded file, which will create a jruby-<version> directory.
3. Add jruby-<version>/bin to the PATH.

You can check that JRuby was installed correctly with this command:

```
$ ruby -v
jruby 9.0.5.0 (2.2.3) 2016-01-26 7bee00d Java HotSpot(TM) 64-Bit
Server VM 25.72-b15 on 1.8.0_72-b15 [darwin-x86_64]
```

Without RVM, you'll have to modify the commands used in this book. RVM lets you invoke JRuby without the jruby or jgem commands, so you'll need to change all ruby commands in this book to jruby commands. You'll also need to

6. http://jruby.org/files/downloads/9.0.5.0/index.html
7. http://jruby.org/files/downloads/9.0.5.0/index.html

prefix other commands (such as bundle, gem, and rails) with the jruby -S prefix, like this:

```
$ jruby -S bundle install
```

Of course, before you can run bundle install, you'll need to install Bundler. If you're using RVM, run this command:

```
$ gem install bundler -v 1.11.2
```

If you're not using RVM, run this command:

```
$ jgem install bundler -v 1.11.2
```

You will *never* be asked to run any of the examples in this book with MRI Ruby. Remember, when you see the ruby, gem, rake, or similar commands, you're expected to run them with JRuby.

Let's move on to the next package.

Installing Git

Git is a source control management tool that allows you to track versions of your code. You'll use Git to switch between different versions of Twitalytics as you deploy it to new platforms. Follow the instructions for downloading and installing Git from the official website.[8]

It's OK to use some other form of version control if you'd prefer, but the examples in this book will be specific to Git. Most of the examples will even work without version control software, but that's not recommended. The source code for each branch you'll create is available from http://pragprog.com/titles/jkdepj2/source_code, so instead of switching branches, you can change to the directory that corresponds to the chapter you're reading. If you don't use Git, some of the Heroku examples later in the book won't work.

Now that your software dependencies are installed, let's move on and run some actual code.

Introducing Warbler

Warbler is a gem that creates a web application archive (WAR) file from a Rails- or Rack-based application.

A WAR file is a zip file that follows a few conventions. Warbler takes care of packaging an application according to these conventions, so all you need to do is run the Warbler commands.

8. http://git-scm.com/download

Joe asks:
What's in a WAR File?

A WAR file is a special case of Java archive (JAR) file; both are really just zip files. But a WAR file is structured according to a standard that's recognized by all Java web servers. You can take a closer look at this by extracting the WAR file you created in this chapter with any unzipping tool. Inside it, you'll find these essential components (among many other things):

```
twitalytics.war
|-- index.html
|-- META-INF/
    `-- MANIFEST.MF
`-- WEB-INF/
    |-- lib/
    `-- web.xml
```

The top-level directory contains all client-accessible content, which is equivalent to the public directory in a Rails application. This is where you'll find all of the HTML files, images, and other static content. The WEB-INF directory contains all the dynamic content for your web application. This includes your Ruby scripts and the Java libraries needed to run a JRuby application. The META-INF directory contains basic metadata about the WAR file, such as who created it and when it was created.

Inside the WEB-INF directory is the web.xml file, which is the most important part of the archive. It contains a description of how the components in the web application are put together at runtime. It's similar to the config/application.rb, config/environment.rb, and config/routes.rb files of a Rails application all combined into a single descriptor. Fortunately, Warbler handles the creation of this file for you based on the settings in the config/warbler.rb file.

You can digitally sign a WAR file, which creates a checksum for each file contained in the archive. This is used by a web server to ensure that no one has tampered with it or that it has not been corrupted in some way. If the checksums don't match, then the server won't load the files.

The WAR file that Warbler creates will be completely self-contained and ready to be deployed to a Java web server. Warbler bundles JRuby, your web framework, a web server, and all of the dependencies needed to adapt a Ruby web application to the JVM.

To demonstrate Warbler, you'll create the simplest web application you can. Create a directory called myapp, and in that directory create a config.ru file. Then put the following code into it:

Warbler/myapp/config.ru

```
run lambda { |env|
    [200, {'Content-Type' => 'text/html'}, 'Hello, World']
}
```

Install the Warbler gem to your JRuby gem path by running this command:

```
$ gem install warbler -v 2.0.1
Fetching: rubyzip-1.2.0.gem (100%)
Successfully installed rubyzip-1.2.0
Fetching: jruby-rack-1.1.20.gem (100%)
Successfully installed jruby-rack-1.1.20
Fetching: jruby-jars-9.0.5.0.gem (100%)
Successfully installed jruby-jars-9.0.5.0
Fetching: warbler-2.0.1.gem (100%)
Successfully installed warbler-2.0.1
4 gems installed
```

Warbler has two JRuby-specific dependencies. The jruby-jars gem includes the core JRuby code and standard library files. This allows other gems to depend on JRuby without freezing to a specific version. The other dependency, the jruby-rack gem, is responsible for adapting the Java web server specification to the Rack specification.

Next, use the warble command to create the archive file. Run it with the war option from the same directory as the config.ru file you created earlier.

```
$ warble war
```

This creates a myapp.war file. In Chapter 2, *Creating a Deployment Environment*, on page 17, you'll learn about all the different ways you can deploy this WAR file. For now, you just need to be able to run it so you can see how Warbler works. To do this, you'll create an executable WAR file by running the same command with the executable option.

```
$ warble executable war
```

This creates a WAR file capable of running without the need for a freestanding Java web server like Tomcat. You can run the WAR file with this command:

```
$ java -jar myapp.war
```

When the server is started, you'll be able to access the application at http://localhost:8080.

That's all you need to know to get started with Warbler. Now let's make some adjustments to the Twitalytics application. It wasn't built to run on JRuby, so it has some code that's specific to MRI. You're going to fix these parts so they work on the new platform.

Creating a JRuby Microservice

In the previous section, you packaged a simple Rack application that was compatible with JRuby, but a real application will require more than just Rack. In this section, you'll package a small Sinatra-based microservice into a WAR file. Warbler is great for small services like this because it produces a portable lightweight artifact you can deploy quickly without any baggage.

Unfortunately, this service is an integral part of Twitalytics and it's under more load than MRI can handle. Porting it to JRuby to will increase its throughput by allowing the application to process each request asynchronously. In this way, the request threads won't block while waiting for external services or doing data processing. To begin, move into the stock-service sample code.

```
$ cd ~/code/stock-service
```

This directory contains the code for a small pure-Ruby HTTP service. The service accepts a POST request with some text. It searches the text for the names of publicly traded companies and then annotates the text with current stock price quotes for those companies. Open the config.ru file and you'll see the handler:

stock-service/config.ru

```
post '/stockify' do
  text = request.body.read.to_s
  stocks = Stocks.parse_for_stocks(text)
  quotes = Stocks.get_quotes(stocks)
  new_text = Stocks.sub_quotes(text, quotes)
end
```

The first line in the handler for the /stockify route captures the body of the request. The second line passes the text to the parse_for_stocks function, which returns a list of symbols matching any company names mentioned in the text. The third line uses the get_quotes function to retrieve current prices for the stocks from a Yahoo! API. The last line combines it all by adding the markup to the text.

Before making any changes, initialize a Git repository and create a branch by running these commands:

```
$ git init
$ git add -A
$ git commit -m "initial commit"
$ git checkout -b warbler
Switched to a new branch 'warbler'
```

Now you can safely configure Warbler while preserving your master branch.

The first step in porting this service to JRuby is adding Warbler to the application's dependencies. Open the Gemfile and put this code at the end of it:

Warbler/stock-service/Gemfile

```
group :development do
  gem 'warbler', '2.0.1'
end
```

The Warbler dependency is in a development group because it's only needed to build a WAR file. You don't need it in production.

Now run Bundler to install the service's dependencies.

```
$ bundle install --binstubs
```

You're ready to package the app into an executable WAR file with Warbler. Since you don't want to type the executable directive every time you package the app, you'll begin by adding a Warbler configuration file. Create a config/warble.rb file by running this command:

```
$ bin/warble config
```

The new file contains a wealth of instructions and examples for the different configuration options, which are helpful to read because you never know what you'll want to change. Don't worry about preserving its contents. You can always re-create it by running warble config again. Given that safety net, replace the entire contents of the config/warble.rb file with this code:

Warbler/stock-service/config/warble.rb

```
Warbler::Config.new do |config|
  config.features = %w(executable)
  config.jar_name = "stock-service"
end
```

Now when you run the warble command, it will detect this configuration and generate an executable WAR file even when you omit the executable directive from the command line. Give it a try:

```
$ bin/warble war
```

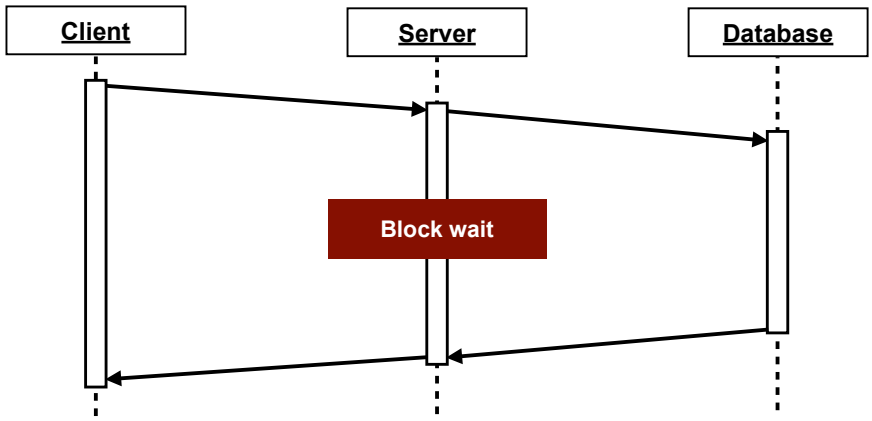
This generates a stock-service.war file, which you can execute by running this command:

```
$ java -jar stock-service.war
```

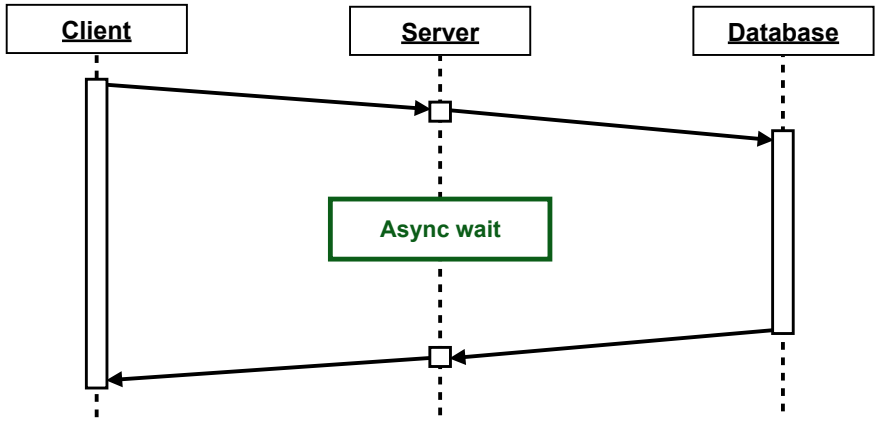
With the Java process running, test out the service by opening another terminal window and executing this command:

```
$ curl -d "Hello Apple, a computer company" http://localhost:8080/stockify
"Hello <div class="stock" data-symbol="AAPL"
data-day-high="102.14">Apple</div>, a computer company"
```

The server responds with an marked-up version of the original text containing current stock price information. Because it depends on an external API, the service does a lot of waiting. This causes the threads that are handling incoming HTTP requests to be blocked. It looks like the following figure.



Now imagine a request thread being freed up to handle other requests instead of blocking for a single request to finish. It looks like the following figure. That's called asynchronous request processing, and it can dramatically improve throughput in an I/O-constrained application (such as an app that relies heavily on a database or external service).



The JVM supports asynchronous I/O in several forms. For this microservice, you'll use an asynchronous context, which is a standard feature of Java, with a background thread to free up your request thread. First, enable the asynchronous capabilities of the web server by adding this line to the config block in your config/warble.rb file:

Warbler/stock-service/config/warble.rb

```
config.webxml.servlet_filter_async = true
```

Then, add these two lines of code to the beginning of the POST handler:

Warbler/stock-service/config.ru

```
response.headers["Transfer-Encoding"] = "chunked"
async = env['java.servlet_request'].start_async
```

The first line sets a standard HTTP header that will ensure the client's request is kept open while the app does its asynchronous processing. The second line creates a new asynchronous context. Now wrap the original four lines of the POST handler in a Thread like this:

Warbler/stock-service/config.ru

```
text = request.body.read.to_s
Thread.new do
  begin
    puts "Thread(async): #{Thread.current.object_id}"
    stocks = Stocks.parse_for_stocks(text)
    quotes = Stocks.get_quotes(stocks)
    new_text = Stocks.sub_quotes(text, quotes)
    async.response.output_stream.println(new_text)
  ensure
    async.complete
  end
end
```

The new Thread will allow the processing to happen in the background so the POST handler can return. And instead of the handler simply returning some string, it will write the output to the asynchronous context. You'll also add a puts statement that logs the ID of the request thread. Add this line to the end of the POST handler (outside the Thread body).

Warbler/stock-service/config.ru

```
puts "Thread(main) : #{Thread.current.object_id}"
```

Now repackage the WAR file and run it again:

```
$ bin/warble
$ java -jar stock-service.war
```

And invoke the service with the same curl command as before:

```
$ curl -d "Text about Apple, a computer company" http://localhost:8080/stockify
```

The output's the same, but in the logs you'll see the different thread identifiers:

```
Thread(main) : 2332
Thread(async): 2330
```

Keep in mind that puts is not atomic, so you might get a bit of interweaving in the output.

This is great, but there's still a problem with the code. The number of threads this service can create is unbounded, which could overrun your system. To make things worse, creating a new thread is an expensive operation. You can fix both of these issues by using a thread pool executor. This is a great example of a kind of concurrency issue you must consider when using JRuby.

You can add a thread pool to the application with only a few lines. First, add a dependency on the concurrent-ruby gem to the Gemfile by adding this code to it:

Warbler/stock-service-thread-pool/Gemfile

```
gem 'concurrent-ruby', require: 'concurrent'
```

And run Bundler to install it:

```
$ bundle install --binstubs
```

Now modify the config.ru file to use the new gem by creating a thread pool. Immediately after the end of the App class, add this line of code:

Warbler/stock-service-thread-pool/config.ru

```
App.set :thread_pool,
  Concurrent::ThreadPoolExecutor.new(max_threads: 100)
```

This uses the ThreadPoolExecutor class to create a cached thread pool and adds it as a setting on the App class. A cached thread pool will grow organically and reuse threads as needed. It also prevents thread starvation by setting an upper bound on the number of threads with the max_thread option.

You can use the thread pool by replacing the Thread.new invocation in the POST handler with a call to settings.thread_pool.post, as shown here:

Warbler/stock-service-thread-pool/config.ru

```
settings.thread_pool.post do
  begin
    puts "Thread(async): #{Thread.current.object_id}"
    stocks = Stocks.parse_for_stocks(text)
    quotes = Stocks.get_quotes(stocks)
```

```
    new_text = Stocks.sub_quotes(text, quotes)
    async.response.output_stream.println(new_text)
  ensure
    async.complete
  end
end
```

Now run Bundler again, repackage with Warbler, run the app, and make the curl request a few more times. In the logs, you'll see that the same thread is being used for the asynchronous part of the service each time it's invoked.

```
Thread(main) : 2332
Thread(async): 2330
Thread(main) : 2334
Thread(async): 2330
Thread(main) : 2336
Thread(async): 2330
```

In practice, you could make this service even more reactive by using an asynchronous HTTP client to invoke the Yahoo! service. And if the parse_for_stocks is going to be expensive or invoke an external service, you could put it in its own thread. Steps like these further eliminate bottlenecks in the system, increasing the potential throughput. But they're possible only with a truly concurrent platform such as JRuby. You'll learn to implement some of these ideas later in the book.

Before moving on, commit your changes to the warbler branch with the git add and git commit commands:

```
$ git add Gemfile Gemfile.lock config config.ru
$ git commit -m "Updated for JRuby"
```

Your microservice is now ready to be deployed to production with Warbler.

Wrapping Up

You packaged a microservice into an archive file. That's a huge step for this application because it means you can deploy it to any environment that has a JVM available. There are many possibilities, including containers that run in the cloud, containers that run on embedded devices, and containers that run on a dedicated server.

You also learned how the JVM can simplify a Ruby architecture no matter what JRuby web framework you use. This will be important as you work your way through the book and as you continue to develop new applications on your own.

Joe asks:
What Is Truffle?

If you follow the JRuby project on Twitter or read the JRuby mailing list, you may have heard about a project called Truffle.

Truffle is a research project sponsored by Oracle Labs.[a] It's an implementation of the Ruby programming language on the JVM using the Graal dynamic compiler and the Truffle AST interpreter framework.[b]

In early 2014, Truffle was open sourced and integrated into the larger JRuby project. The Truffle developers and JRuby developers have been working alongside each other, sharing code, and even sharing a mailing list for a while now. They're not so much competitors as they are contemporaries.

Truffle has the potential to achieve peak performance well beyond what's possible with standard JRuby, but it's not production ready. Major components such as OpenSSL and networking are yet to be completed. It also requires an experimental JVM (Graal) and doesn't work with a standard JVM.

You can learn more about Truffle from the project's official website,[c] which is hosted by its lead developer.

---

a. http://labs.oracle.com/
b. http://openjdk.java.net/projects/graal/
c. http://chrisseaton.com/rubytruffle/

Having a JRuby application packaged into a WAR file is a good first step, but you still need to deploy it and consume it. In the coming chapters, you'll learn how to get this WAR file into production and how to use JRuby to invoke the services it exposes. But first, you need to create a production environment in which it can run.

CHAPTER 2

Creating a Deployment Environment

A production JRuby environment is simpler than you might expect. There aren't any C extensions to compile, which means you won't need to install any native libraries. And you'll run your entire app in a single process, which eliminates the need for tools that do coordination and load balancing. In fact, many JRuby deployments have only one external dependency: the JVM itself. For that reason, the steps in this chapter form the basis for every kind of JRuby app.

In Chapter 1, *Getting Started with JRuby*, on page 1, you created a small JRuby microservice. Now you'll provision a new environment for this service and package it with the essential software it needs to run in production. But just because you're deploying a JRuby application doesn't mean you have to turn your world upside down. The tools you'll use, such as Docker and Heroku, may be familiar. And you can use them to build environments for any kind of app. At the end of this chapter, you'll have a deployment environment that's ready to scale up to meet demand and take advantage of everything JRuby has to offer.

Installing Docker

Deployment is the process of taking code or binaries from one environment and moving them to a another environment where you execute them. In the case of a web app case, you'll move code from your development machine to a production server. You've already configured a development environment, but you still need to create a production environment you can use as the target of your deployments. For this, you'll use Docker,[1] which reduces the process of provisioning a production environment to just a few steps.

1. http://docker.com/

Docker is a Linux-based containerization platform. It runs processes in isolated environments without the need for a complete visualization layer for each process. You can use Docker to run multiple isolated processes on the same host without excessive overhead. You'll use Docker primarily as a development tool, allowing you to run a simulated production environment on a local machine. There is one catch, though. If your local machine isn't running Linux, you'll need a virtual machine to run Docker. Fortunately, Docker distributes a lightweight headless virtual machine, called Docker Machine, that provides many advantages over traditional virtualization.

Docker Machine runs on VirtualBox,[2] an open source virtualization platform. If you're not running Linux, you'll need to install both of these. If you are running Linux, you'll need to install only Docker.

Installing Docker on Mac OS X or Windows

You can install both Docker and Docker Machine on Mac and Windows using the Docker Toolbox native installer. Open a browser and navigate to the Docker Toolbox website.[3] Select the Installer for Mac and download it. Once the package is downloaded, open it to run the wizard. Follow the steps in the wizard to complete the installation as described on the Docker website.[4]

Now open a terminal. If you're on Windows, you'll need to double-click the Docker CLI shortcut on your Desktop to open a Docker terminal instead of a standard terminal. If the system displays a User Account Control prompt to allow VirtualBox to make changes to your computer, choose Yes. From the terminal, run these commands:

```
$ docker-machine -v
docker-machine version 0.4.0 (9d0dc7a)

$ docker-machine ls
NAME      ACTIVE   DRIVER       STATE     URL
default   *        virtualbox   Running   tcp://192.168.99.100:2376
```

This shows that a single Docker Machine, named default, is running.

On Mac only, you must set a few environment variables so the Docker client can communicate with the Docker Machine. Run this command:

```
$ eval "$(docker-machine env default)"
```

2. https://www.virtualbox.org
3. https://www.docker.com/toolbox
4. http://docs.docker.com/mac/step_one/

On Windows, those environment variables are set for you when you run the Desktop app. On Mac, if you don't want to run that command every time you open a new terminal, then run this command to add the environment variables to your profile:

```
$ docker-machine env default >> ~/.profile
```

Now you're ready to use the Docker client. You can move on to *Getting Started with Docker*, on page 20.

Installing Docker on Linux

Docker runs natively on Ubuntu, but it requires a 64-bit architecture and a 3.10 kernel version or later. You can check your architecture by running this command:

```
$ uname -p
x86_64
```

And you can check your kernel version with this command:

```
$ uname -r
3.13.0-57-generic
```

If the output of either `uname` command doesn't match the requirements, then you'll need to run Docker on a virtualization layer by installing Docker Machine.[5] Otherwise, you can install Docker natively. To begin the native installation, update your package manager by running this command:

```
$ sudo apt-get update
```

Then install the generic Linux kernel image. This kernel has the advanced multi-layered unification filesystem (AUFS) built in, which is required to run Docker.

```
$ sudo apt-get install linux-image-generic-lts-trusty
```

Now reboot your machine:

```
$ sudo reboot
```

After your computer has restarted, you can install Docker with this command:

```
$ curl -sSL https://get.docker.com/ | sh
```

Now you're ready to use the Docker CLI.

5. http://docs.docker.com/machine/install-machine/

Getting Started with Docker

Verify that Docker is installed by running this command:

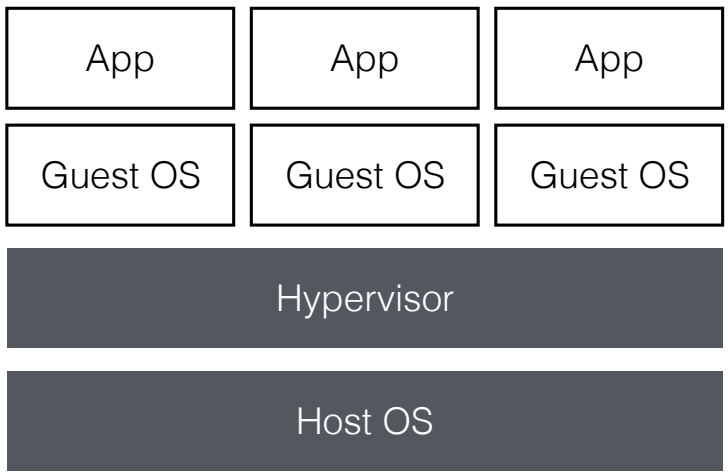
```
$ docker --version
Docker version 1.10.0, build 590d5108
```

Now check that Docker can communicate with Docker Machine by running this command:

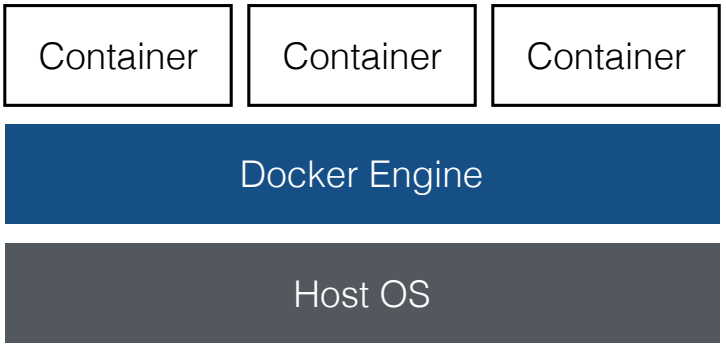
```
$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED    ...
```

This is an empty list, but you'd see a list of running Docker processes (known as containers) if there were any. A container represents some isolated process or processes that are running within the Docker context. They're isolated from the rest of the processes on your machine and even from the other Docker containers. This has many of the same benefits as traditional visualization but with a very different underlying implementation.

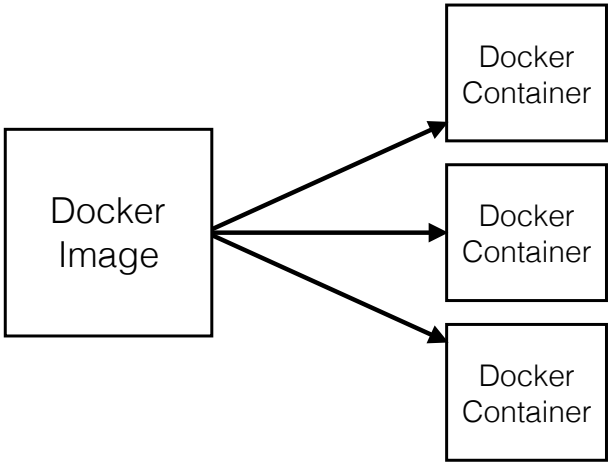
In traditional visualization, a host operating system runs a hypervisor that manages one or more complete guest operating systems. These guest operating systems are isolated from each other, but they also add a great deal of overhead. This model is shown in the following figure.



With Docker, each container runs natively on the host operating system via the Docker Engine. They're isolated from each other, but they still rely on the host operating system to schedule processes, allocate memory, and do other things that are common across containers. This model is shown in the figure on page 21.



Each Docker container is based on an image that defines the environment the container's processes will run in. Images are packages of software that are loaded into a container before it runs. You might have images that include a Java runtime or an image that includes your application's dependencies. You use these images as a base for creating an image that includes your entire application, as shown here.



You can list the Docker images available on your machine by running this command:

```
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED    ...
```

The command doesn't list any images because you haven't created any since installing Docker a moment ago. Go ahead and download your first image. You'll begin with the heroku/jvm image, which mirrors the cloud environment

running on the Heroku platform as a service.[6] You'll deploy to Heroku later in the chapter, so this allows you to replicate that environment locally first. Run the following command to pull the image (it's several hundred megabytes, so this may take some time):

```
$ docker pull heroku/jvm
Using default tag: latest
latest: Pulling from heroku/jvm
ecf3ac44a558: Pull complete
5c0c6781ba3b: Pull complete
75a40d761c97: Pull complete
c5b21110f7b8: Pull complete
Digest: sha256:a2483cf8906b2d14b7ff6fded15601ee3f00172cbfc21ab1e7d80c45a4ee0cfb
```

Run the images command again:

```
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED     ...
heroku/jvm          latest              35ecdbd5516b        6 days ago
```

You now have a local representation of the Heroku stack. This stack includes a JVM, so it's perfect for running an executable WAR file. You can inspect the JVM version by running the following command:

```
$ docker run -t heroku/jvm java -version
openjdk version "1.8.0_51-cedar14"
OpenJDK Runtime Environment (build 1.8.0_51-cedar14-b16)
OpenJDK 64-Bit Server VM (build 25.51-b03, mixed mode)
```

Notice that the version of the Java runtime is different from your local Java installation. The docker run command runs another command inside the context of a Docker container. The -t option defines the image to load, which in this case is the heroku/jvm image you downloaded a moment ago. The last part, java -version, is the command that Docker will run. When Docker ran the java command, it didn't use the Java runtime on your host system. It used the Java runtime contained in the heroku/jvm image.

To use this Java runtime with a Warbler WAR file, you must add the WAR artifact to the container. For this, you'll create your own Docker image.

Creating a Docker Image

A Docker image is defined by a Dockerfile, which is a text document containing instructions Docker follows as it provisions a new container. To create a Docker image for your WAR file, you must create a Dockerfile in the project.

6. https://heroku.com

To begin, move into the root directory of the stock-service application you implemented in Chapter 1, *Getting Started with JRuby*, on page 1.

```
$ cd ~/code/stock-service
```

Then create a Dockerfile in that directory and put the following code in it:

Warbler/stock-service-docker/Dockerfile

```
FROM heroku/jvm

ADD ./stock-service.war /app/user/
```

This defines an image that inherits from the heroku/jvm image. It then tells Docker to add the WAR file from the local machine to the /app/user directory in the image. This directory is what Heroku considers the root of any Docker-based application.

Now build the image by running the build command:

```
$ docker build -t stock-service .
Sending build context to Docker daemon 24.86 MB
Step 0 : FROM heroku/jvm
---> 35ecdbd5516b
Step 1 : ADD ./stock-service.war /app/user/
---> Using cache
---> 103d45860c83
Successfully built 103d45860c83
```

This creates a new image named stock-service, which is ready to run your application. Execute the images command again to see it in the list:

```
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED
heroku/jvm          latest              35ecdbd5516b        6 days ago
stock-service       latest              103d45860c83        26 minutes ago
```

Now run the application in a Docker container:

```
$ docker run --publish 8080:8080 -t stock-service java -jar stock-service.war
2015-08-15 20:01:58.590:INFO::main: Logging initialized @186ms
2015-08-15 20:01:58.597:INFO:oejr.Runner:main: Runner
2015-08-15 20:01:58.709:INFO:oejs.Server:main: jetty-9.2.9.v20150224
2015-08-15 20:02:03.891:WARN:oeja.AnnotationConfiguration:main: ServletCont...
2015-08-15 20:02:04.204:INFO:/:main: INFO: jruby 9.0.5.0 (2.2.3) 2016-01-26...
2015-08-15 20:02:04.206:INFO:/:main: INFO: using a shared (threadsafe!) run...
2015-08-15 20:02:09.534:INFO:oejsh.ContextHandler:main: Started o.e.j.w.Web...
2015-08-15 20:02:09.543:WARN:oejsh.RequestLogHandler:main: !RequestLog
2015-08-15 20:02:09.557:INFO:oejs.ServerConnector:main: Started ServerConne...
2015-08-15 20:02:09.557:INFO:oejs.Server:main: Started @11155ms
```

This command tells Docker to run a container with the java -jar stock-service.war command, based on the stock-service image, and with port 8080 published to the host system (so you can access it outside the container). To view the application, you'll need to know the hostname of the container. On Linux this is localhost, but on Mac and Windows it's the address of the Docker Machine. Open another terminal and run this command to see it:

```
$ docker-machine ip default
192.168.99.100
```

Now you can use curl to make a request to the service just as you did when it was running locally if you're on Linux, or with the following command on Mac and Windows:

```
$ curl -d "Hi Apple" http://$(docker-machine ip default):8080/stockify
Hi <div class='stock' data-symbol='AAPL' data-day-high='116.14'>Apple Inc.</div>
```

Your microservice is ready for production! Deploying your Docker image could involve a docker push command, which uploads the image to a Docker host and runs it. However, managing your own Docker infrastructure in production defeats much of the purpose of Docker, which abstracts away the underlying platform. For that reason, you'll deploy the stock-service Docker image to a mature and well-curated platform, Heroku.

But even without deploying to production, the work you've done to set up Docker is still invaluable. It gives you the ability to run a production environment locally. And in the coming chapters you'll enhance this environment to include a database and other services. Having a complete production environment you can run with a single command is great for debugging, on-boarding new employees, and scaling. Even better, because the Docker environment you created is based on the Heroku stack, you can deploy the WAR to Heroku with a great deal of confidence.

Deploying to the Cloud

Heroku is a cloud-based platform as a service that helps you deploy, run, and manage applications written in many languages, including Ruby, Java, and JRuby. You can deploy to Heroku by pushing source code to a Git repository, by uploading precompiled binaries, or by pushing an app configured for Docker. You'll start by deploying the stand-alone WAR file you created with Warbler directly to Heroku. This is the lightest and fastest way to get your app running in the cloud. Then you'll deploy the Docker image you created a moment ago. Deploying a Docker image is a heavier and slower process but offers advantages of its own.

First, create a free Heroku account by visiting the Heroku website[7] and filling out a few bits of information. You won't even need a credit card for now.

Once you've created a Heroku account, download and install the Heroku toolbelt.[8] This is a command-line interface (CLI) used to create, manage, and deploy your Heroku apps. You can do most of these things from the web-based dashboard,[9] but you'll need to use the CLI to deploy your app.

With the toolbelt installed, open a terminal and log in with the credentials you created earlier:

```
$ heroku login
Enter your Heroku credentials.
Email: jruby@example.com
Password:
```

Authenticating is required to allow both the heroku and git commands to work with the deployment examples in this book.

Note that if you're behind a firewall that requires the use of a proxy to connect with external HTTP/HTTPS services, you should set the HTTP_PROXY or HTTPS_PROXY environment variable in your local development environment before running the heroku command.

Once you're logged in, install the heroku-deploy toolbelt plugin by running this command:

```
$ heroku plugins:install https://github.com/heroku/heroku-deploy
```

This plugin helps you deploy WAR and JAR files from the Heroku CLI. But first you'll need an app to deploy to.

Make sure you're still in the root directory of the stock-service app, and run the following command to provision a new Heroku app and associate it with your local app:

```
$ heroku create
Creating app... done, stack is cedar-14
https://obscure-fjord-4138.herokuapp.com/ | https://git.heroku.com/...
```

Heroku will randomly assign your app a unique name based on a clever mash-up of terms. The example used here is obscure-fjord-4138, yours will be different.

Now you need to create one new file that tells Heroku how to run your app. Create a Procfile in the root directory of the project and put this code in it:

7. http://signup.heroku.com
8. http://toolbelt.heroku.com
9. http://dashboard.heroku.com

Warbler/stock-service-docker/Procfile

```
web: java -Xmx384m -Xss512k -jar stock-service.war
```

This tells Heroku that your application has a single process, called web, and gives it the command to run for that process. The command uses two options, Xmx and Xss, to optimize memory usage characteristics of the process for the Heroku platform. You'll learn what these mean and how to customize them in Chapter 7, *Tuning a JRuby Application*, on page 109. Otherwise, it's the same command you ran locally.

Now deploy your app by running this command from the same directory as the Procfile:

```
$ heroku deploy:jar --jar stock-service.war
Uploading stock-service.war....
-----> Packaging application...
...
-----> Done
```

The deploy process will take a minute to package your application and upload the WAR file to Heroku. When it's done, you can ensure the process is running with this command:

```
$ heroku ps:scale web=1
Scaling dynos... done, now running web at 1:Free.
```

And you can open the app in a browser like this:

```
$ heroku open
```

You'll see the landing page where you can test out the /stockify service. Notice that the URL begins with the name of the app and is followed by herokuapp.com. This is the standard convention for new Heroku apps, but you can always configure DNS for custom domains.

Return to the terminal, and exercise the /stockify service with curl but replace obscure-fjord-4138 with the name of your app:

```
$ curl -d "Hi Apple" http://obscure-fjord-4138.herokuapp.com/stockify
Hi <div class='stock' data-symbol='AAPL' data-day-high='116.14'>Apple Inc.</div>
```

Deploying only the executable WAR file to Heroku is the quickest way to get your app running in the cloud. But you can also deploy the entire Docker image you created earlier. You may prefer this approach if you're using Docker to set up some extra dependencies.

Like the toolbelt plugin for WAR files, there's also a toolbelt plugin for Docker. Install the heroku-container-tools toolbelt plugin by running this command:

```
$ heroku plugins:install heroku-container-tools
```

Heroku also needs an app.json file, which is a descriptor containing a little metadata about your app. You can store many different things in this file, such as environment variables, a website URL, and more. But only one element is required for Docker:

Warbler/stock-service-docker/app.json

```
{
  "name": "stock-service"
}
```

Now you're ready to initialize the Heroku Docker config. Run this command:

```
$ heroku container:init --dockerfile Dockerfile
```

This reads the Dockerfile and Procfile and generates a docker-compose.yml, which describes the complete environment.

Finally, deploy to the same Heroku app you created earlier by running this command:

```
$ heroku container:release
Remote addons:  (0)
Local addons:  (0)
Missing addons:  (0)
Creating local slug...
Building web...
...
```

The release process will take some time to package your application and upload the image to Heroku. When it's done, you can run the open command again to view the app in a browser. Or you can use curl as before.

Your microservice is running in the cloud and can easily be scaled up to handle growing demand. When that time comes, you can convert your free Heroku account to a paid account and rapidly increase the number of instances using the heroku ps:scale command. Or you can scale vertically (increase RAM and CPU) with the heroku ps:resize command.

Wrapping Up

You've created an environment you can use to run your microservice or any other JRuby application in production. You've also set up Docker, which allows you to add new components to your infrastructure without running a bunch of commands or managing a complicated configuration management codebase. The tools you've used are state of the art for the industry, and it's

likely that you'll use them to set up new environments each time you embark with a new customer or employer.

When you need to scale this environment horizontally, you won't have to do much work. You'll start with the base image and launch Docker containers from it. Or you can simply scale up your Heroku app.

In the coming chapters, you'll build on this base Docker image. You'll add more components and deploy more complicated web applications. In the next chapter, you'll convert a full-blown Rails application to JRuby.

CHAPTER 3

Deploying a Rails Application

The JRuby microservice you built in Chapter 1, *Getting Started with JRuby*, on page 1 used lightweight technologies ideal for high-powered workhorse services. But now you'll build a JRuby application on Rails, which is better equipped to deal with the demands of a traditional customer-facing web app.

Ruby on Rails revolutionized the way programmers build web applications. It set a precedent that all other web frameworks are now compared to. The beauty of JRuby is that you get to use this robust and mature full-stack tool without giving up the power and maturity of the JVM.

In this chapter, you'll port an existing MRI-based Rails application to JRuby. You'll use Puma as the server because it works wonderfully on both JRuby and MRI. But with JRuby, you can take full advantage of the server's parallelism. With the app running on JRuby, you can deploy it to a curated cloud platform and a customized private server.

Puma works with JRuby almost exactly as it does with MRI, so you won't have to veer off the traditional Ruby path as you did with Warbler. Deploying a JRuby on Rails application with Puma looks very much like traditional Ruby deployment.

What Is Traditional Deployment?

Traditional Ruby deployment with MRI uses a type of runtime architecture that handles HTTP requests by placing a proxy in front of a pool of application instances. In *What Makes JRuby So Great?*, on page 2, we discussed some of the deficiencies of this architecture and showed how JRuby can improve it. But the way you ran your app and deployed your code wasn't traditional.

With traditional deployment, new versions of an application are released using tools like Capistrano and Git to pull the code from a repository and push it

to a production server. Once the code has been pushed, each application process is restarted. With JRuby, you can reduce the number of processes to one, which can make it faster to get back online after a deployment. But the architecture you built in Chapter 2, *Creating a Deployment Environment*, on page 17 with Warbler greatly impacted the way you deployed code to the server. Instead of pulling code from a repository, you packaged everything into an archive file. There are advantages to that kind of deployment, but it diverges from what traditional Rubyists expect.

You won't use Warbler in this chapter, and you won't deploy a prepackaged image or binary. Instead, you'll use Puma, which bridges the gaps between JRuby and MRI. But switching Twitalytics to Puma is only one of the steps required to get it running on JRuby.

Porting to JRuby

Before deploying any existing MRI-based Rails application on JRuby, you must make a few essential changes to its dependencies. To ensure that you keep track of your changes, initialize a Git repository and create a branch with the following commands:

```
$ cd ~/code/twitalytics
$ git init
$ git add -A
$ git commit -m "initial commit"
$ git checkout -b jruby
Switched to a new branch 'jruby'
```

In the past, preparing an app for JRuby required a number of library and code changes because many gems and commands such as `Kernel#exec` couldn't be used reliably. But today, JRuby 9k uses native operations for most I/O and process APIs. This makes it the only POSIX-friendly JVM language, with full support for spawning processes, inheriting open streams, performing nonblocking operations on all types of I/O, and generally fitting well into a POSIX environment. And for that reason, the work required to port an application to JRuby has been greatly reduced.

Many gems that use native code on MRI, like Nokogiri[1] and Typhoeus,[2] even offer JRuby compatibility today. There are still, however, a few gems you'll need to change. You can find an up-to-date list of incompatible gems and

1. http://www.nokogiri.org/
2. https://github.com/typhoeus/typhoeus

their alternative implementations on the JRuby wiki,[3] but we'll address the most common ones as we port Twitalytics over to its new runtime.

Open the Twitalytics Gemfile and see if you can find any incompatible gems from the list on the JRuby wiki. The first ones you'll notice are these:

twitalytics/Gemfile

```
gem 'sqlite3', group: :development
gem 'pg', group: :production
```

The pg gem is a database adapter for PostgreSQL and the sqlite3 gem is for the SQLite database, which you'll use in development. Both of these adapters make extensive use of native code and thus don't work on JRuby. There have been initiatives[4,5] to make then compatible with JRuby, but the majority of JRuby users swap them out for ActiveRecord JDBC adapters. Replace the two gem dependencies with the following:

Rails/twitalytics-jruby/Gemfile

```
gem 'activerecord-jdbcsqlite3-adapter', group: :development
gem 'activerecord-jdbcpostgresql-adapter', group: :production
```

This will load the ActiveRecord JDBC adapters for PostgreSQL and SQLite in development and production, respectively. JDBC is the standard Java Database Connectivity API. It's a mature and robust protocol for interfacing with many kinds of databases in a platform-independent way. There are adapters for MySQL, Oracle, SQL Server, and many other vendors. This also makes your app more portable because installing the JDBC adapter won't require the physical database to be present to compile.

Look down the list of gems and you'll see another one that needs replacing:

twitalytics/Gemfile

```
gem 'therubyracer'
```

This gem provides an embedded JavaScript interpreter for Ruby using the V8 engine, which can't run on the JVM. Fortunately, there's an alternative gem called therubyrhino, which embeds a JVM-friendly engine. It's even maintained by the same person.[6] To use it, replace the entry with this code:

Rails/twitalytics-jruby/Gemfile

```
gem 'therubyrhino'
```

3. https://github.com/jruby/jruby/wiki/C-Extension-Alternatives
4. https://github.com/ged/ruby-pg/pull/1
5. https://github.com/headius/jruby-pg
6. https://github.com/cowboyd

Keep looking down the list of gems. The next incompatibility you'll find is this:

twitalytics/Gemfile

```
gem 'unicorn'
```

Unicorn is a popular HTTP server for Rack applications designed to take advantage of features in Unix/Unix-like kernels. As a result, it doesn't work on Windows and its use of native code makes it a bad fit for JRuby. Fortunately, the Puma server is an excellent alternative. It's built for parallelism and portability, which means you'll be able to handle many requests concurrently on any platform. Replace the Unicorn gem with this:

Rails/twitalytics-jruby/Gemfile

```
gem 'puma'
```

The last gem you'll add is only for Windows. Rails requires a source for time zone information, which it can't collect natively on Windows. To make this easier, add the tzinfo-data gem. Put this statement at the end of the Gemfile.

Rails/twitalytics-jruby/Gemfile

```
gem 'tzinfo-data', platforms: [:mingw, :mswin, :jruby]
```

You're ready to install your new dependencies. Run the following command from the root directory of the project:

```
$ bundle install --binstubs
...
Bundle complete! 12 Gemfile dependencies, 53 gems now installed.
Use `bundle show [gemname]` to see where a bundled gem is installed.
```

The --binstubs option instructed Bundler to create scripts such as rake and puma in the bin/ directory of the project. Using these scripts has the same effect as running bundle exec but without creating an extra process from which to launch the main process. This saves several seconds on the execution time of each command.

Unfortunately, Bundler and Rails disagree on what some of the bin files should look like. To install the correct scripts for Rails, run this command and enter Y when prompted to overwrite the bin/rake and bin/rails files:

```
$ bin/rake rails:update:bin
```

You're almost ready to run the app. There's just one more little caveat, which you can thank the lawyers for.

Installing the Java Cryptography Extension

Rails uses a type of encryption referred to as unlimited-strength cryptography. But in order to comply with U.S. cryptography export laws, Oracle disables this kind of strong cryptography when it distributes software products. Thus, the Java Virtual Machine installed on your computer most likely has a maximum key size of 128 bits. If you live in a country to which Oracle can export strong encryption, you must download and install the Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files.[7] In most cases, this means that you need to download the JAR file and put it in your $JAVA_HOME/jre/lib/security directory.

If for some reason you cannot download the JCE extension, you may need to implement a simple workaround.[8,9]

Running the App

Let's make sure your application is in order by running this command:

```
$ bin/rake routes
   Prefix Verb   URI Pattern               Controller#Action
    posts GET    /posts(.:format)          posts#index
          POST   /posts(.:format)          posts#create
 new_post GET    /posts/new(.:format)      posts#new
edit_post GET    /posts/:id/edit(.:format) posts#edit
     post GET    /posts/:id(.:format)      posts#show
          PATCH  /posts/:id(.:format)      posts#update
          PUT    /posts/:id(.:format)      posts#update
          DELETE /posts/:id(.:format)      posts#destroy
```

The list shows the available routes the application can handle. You'll use some of these in a moment, but first you need a database. Run the following command to initialize SQLite:

```
$ bin/rake db:create db:migrate
== 20150914211918 CreatePosts: migrating ======================================
-- create_table(:posts)
   -> 0.0048s
   -> 0 rows
== 20150914211918 CreatePosts: migrated (0.0061s) =============================
```

The app is ready to run. Execute this command to start the server:

7. http://www.oracle.com/technetwork/java/javase/downloads/jce8-download-2133166.html
8. https://gist.github.com/jkutner/5abc59c7cafaf2132865
9. https://github.com/jruby/jruby/wiki/UnlimitedStrengthCrypto

```
$ bin/rails server
=> Booting Puma
=> Rails 4.2.4 application starting in development on http://localhost:3000
=> Run `rails server -h` for more startup options
=> Ctrl-C to shutdown server
The signal USR1 is in use by the JVM and will not work correctly on this platform
Puma starting in single mode...
* Version 3.0.1 (jruby 2.2.3), codename: Plethora of Penguin Pinatas
* Min threads: 0, max threads: 16
* Environment: development
* Listening on tcp://localhost:3000
Use Ctrl-C to stop
```

The Puma server is up and running. Open a browser to http://localhost:3000 and take a look at the app. You'll see your requests logged in the terminal session of the server process.

Before moving on, commit all of your changes to the jruby branch by running these commands:

```
$ git add .
$ git commit -m "Ported to JRuby"
```

The work you've done is essential in porting any Rails application and most MRI applications to JRuby. You've learned about a few important gems and some common problems. If you were creating a Rails app from scratch, the rails new command would have done many of these steps for you. But other frameworks don't provide so much magic.

Now it's time to leave the realm of development and get this app ready for the real world. In the next section, you'll prepare Twitalytics for deployment.

Configuring Rails for Production

When you ran the bin/rails server command, Puma started up with a nice set of defaults for development mode. But in production, you'll want a more explicit configuration suited for your production deployment platform. In this section, you'll create that platform and get Twitalytics ready for it.

To configure Puma, create a config/puma.rb file, open it in an editor, and put the following code in it:

Rails/twitalytics-jruby/config/puma.rb

```
port        ENV['PORT']     || 3000
environment ENV['RACK_ENV'] || 'development'
threads     (ENV["MIN_PUMA_THREADS"] || 0), (ENV["MAX_PUMA_THREADS"] || 16)
preload_app!
```

This sets the port, environment, and thread pool size based on environment variables. The defaults are intended for development. How you set these environment variables will depend on the platform you're deploying to. We'll address that later in the section.

Because Puma is a multithreaded server and JRuby has real threads, you'll want to configure your database connection pool size appropriately. To do this, create a config/initializers/database_connection.rb file, and put this code in it:

Rails/twitalytics-jruby/config/initializers/database_connection.rb

```
Rails.application.config.after_initialize do
  ActiveRecord::Base.connection_pool.disconnect!

  ActiveSupport.on_load(:active_record) do
    config = ActiveRecord::Base.configurations[Rails.env] ||
             Rails.application.config.database_configuration[Rails.env]
    config['pool'] = ENV['MAX_PUMA_THREADS'] || 16
    ActiveRecord::Base.establish_connection(config)
  end
end
```

This sets the maximum size of the connection pool to the same value as the maximum size of the thread pool. This is a good place to start, but as you profile the application under real load, you may eventually tweak this setting. We'll discuss how to analyze this in a later chapter.

With the configuration in place, you can start the server with a simple puma command. But first, set some environment variables to ensure the app detects them. On Windows, run these commands:

```
C:\> set PORT=5000
C:\> set MIN_PUMA_THREADS=1
C:\> set MAX_PUMA_THREADS=2
```

On Mac and Linux run these commands:

```
$ export PORT=5000
$ export MIN_PUMA_THREADS=1
$ export MAX_PUMA_THREADS=2
```

In the same terminal session, start the server by running this command:

```
$ bin/puma -C config/puma.rb
The signal USR1 is in use by the JVM and will not work correctly on this platform
Puma starting in single mode...
* Version 3.0.1 (jruby 2.2.3), codename: Plethora of Penguin Pinatas
* Min threads: 1, max threads: 2
* Environment: development
* Listening on tcp://0.0.0.0:5000
Use Ctrl-C to stop
```

Open a browser to http://localhost:5000 and confirm the app is running correctly. Then shut the server down by pressing Ctrl-C.

The Puma configuration is done, but now you need to configure the environment it runs in. Open the Gemfile and add this line below the source entry at the top of the file:

Rails/twitalytics-jruby/Gemfile

```
ruby '2.2.3', :engine => 'jruby', :engine_version => '9.0.5.0'
```

This ensures any environment you deploy to will be running the correct version of the Ruby runtime.

Creating the Deployment Environment

Now it's time to build a Docker image for this app. The image will be similar to the one you created in Chapter 2, *Creating a Deployment Environment*, on page 17, but it will need a few additional dependencies because JRuby won't be packaged with the app as it was with Warbler. To start, create a Dockerfile in the root directory of the repo and add the following code to it.

Rails/twitalytics-jruby/Dockerfile

```
FROM heroku/jvm
```

This instructs Docker to use the same JVM base image you used with Warbler. Now you can provision the Docker container with a JRuby installation by adding this code:

Rails/twitalytics-jruby/Dockerfile

```
RUN mkdir -p /usr/lib/jruby
ENV JRUBY_HOME /usr/lib/jruby
RUN curl -s -L  \
 https://s3.amazonaws.com/jruby.org/downloads/9.0.5.0/jruby-bin-9.0.5.0.tar.gz \
 --retry 3 | tar xz -C /usr/lib/jruby --strip-components=1
ENV PATH /usr/lib/jruby/bin:$PATH
```

The first line creates a directory for the JRuby installation, and the second line sets the environment variable for JRUBY_HOME to that location. Then it downloads the JRuby runtime from the official JRuby S3 bucket and installs it. Finally, it puts the JRuby command on the PATH.

Now you'll need to install Twitalytics's dependencies, which require a Bundler installation. Add this code to the end of the Dockerfile:

Rails/twitalytics-jruby/Dockerfile

```
RUN jruby -S gem install bundler -v 1.11.2 --no-ri --no-rdoc
```

Then add these lines, which will copy the app's code into the image and run Bundler:

Rails/twitalytics-jruby/Dockerfile

```
COPY . /app/user/
RUN bundle install
```

One disadvantage to using Docker in this way is that every time you rebuild the image it will reinstall your dependencies from scratch (downloading them from RubyGems.org) because Docker doesn't have a persistent storage area like you might have locally in ~/.gem. That means every time you change your Gemfile, you'll need to rebuild the image. You'll improve this later in the book.

The final piece of Docker configuration sets a few environment variables. Add these lines to the Dockerfile.

Rails/twitalytics-jruby/Dockerfile

```
ENV RACK_ENV development
ENV MAX_PUMA_THREADS 8
```

The first line sets the Rack environment to production, and the next line sets the maximum thread pool size to a value that's appropriate for the Docker container.

Now you must build the image. If you're using a Mac, make sure Docker Machine is running by executing this command:

```
$ docker-machine start default
```

If you're on Windows, double-click the Docker CLI icon on the Desktop to start a new session.

On all platforms, run this command from the root directory of the project to build the image:

```
$ docker build -t twitalytics .
Sending build context to Docker daemon 1.138 MB
Step 0 : FROM heroku/jvm
...
Removing intermediate container f38bac85c2b6
Successfully built d3b5ce3755e4
```

Before you can run the app, you'll need a database. In production, Twitalytics uses PostgreSQL and you want the Docker environment to simulate a real production environment. Thus, you'll also run the PostgreSQL server in a Docker container. Normally, this would require a great deal of configuration in order to get the two containers to talk to each other, but the Heroku

Docker CLI you used in Chapter 2, *Creating a Deployment Environment*, on page 17 simplifies the setup to just a few steps.

Create an app.json file in the root directory of the project. As before, this file will contain metadata about your app. In the case of Twitalytics, it will define the add-ons, which include a database. Put this code in the app.json file:

```
{
  "addons": ["heroku-postgresql"]
}
```

Next, create a Procfile in the same directory, and put the following code in it:

```
web: bin/puma -C config/puma.rb
```

This file tells other platforms and tools how to run the app. You'll notice that it contains the same command you used to run the app earlier.

Now run the Heroku Docker CLI to generate the Docker configuration:

```
$ heroku container:init --dockerfile Dockerfile
```

This generates a Docker Compose configuration file, a docker-compose.yaml file, which defines multiple Docker containers you can run in conjunction. Docker Compose is one of the tools in the Docker Toolbox you installed. The heroku container:init command reads your app.json file and Procfile so it knows everything needed to create each of the container environments and wire them together.

Open the docker-compose.yaml, but don't edit it. Here's what you'll see:

Rails/twitalytics-jruby/docker-compose.yml

```
web:
  build: .
  command: 'bash -c ''bin/puma -C config/puma.rb'''
  working_dir: /app/user
  environment:
    PORT: 8080
    DATABASE_URL: 'postgres://postgres:@herokuPostgresql:5432/postgres'
  ports:
    - '8080:8080'
  links:
    - herokuPostgresql
shell:
  build: .
  command: bash
  working_dir: /app/user
  environment:
    PORT: 8080
    DATABASE_URL: 'postgres://postgres:@herokuPostgresql:5432/postgres'
```

```
  ports:
    - '8080:8080'
  links:
    - herokuPostgresql
  volumes:
    - '.:/app/user'
herokuPostgresql:
  image: postgres
```

It defines three container types: web, shell, and herokuPostgresql. The first two use the image defined by your Dockerfile, while the third uses the standard Postgres image. The web container corresponds to the web process defined in your Procfile, and the shell allows you to run an interactive command line within a Docker container based on your image.

Before you run the app, you'll need to prepare the production database inside the Docker container. For this, you can use the shell container. Run the following command to start the shell:

```
$ docker-compose run shell
root@9c9f633ea047:~/user#
```

The new prompt, which will look something like the one shown here, indicates that you're running inside the container. Execute some commands like ls to demonstrate that you're in the root directory of Twitalytics:

```
root@9c9f633ea047:~/user# ls
Dockerfile  Gemfile  Gemfile.lock Procfile  README.rdoc  Rakefile  app  app.json
bin         config   config.ru   db  docker-compose.yml  lib  log  public  test  vendor
```

Now run the following Rake task to migrate the database:

```
root@9c9f633ea047:~/user# bin/rake db:migrate
== 20150914211918 CreatePosts: migrating ====================================
-- create_table(:posts)
   -> 0.0048s
   -> 0 rows
== 20150914211918 CreatePosts: migrated (0.0061s) ===========================
```

The database is ready. Exit the shell by entering exit. Then from your local command line, run this to start the app:

```
$ docker-compose run web
The signal USR1 is in use by the JVM and will not work correctly on this platform
Puma starting in single mode...
* Version 2.13.4 (jruby 2.2.2), codename: A Midsummer Code's Dream
* Min threads: 0, max threads: 8
* Environment: development
* Listening on tcp://0.0.0.0:8080
Use Ctrl-C to stop
```

Determine the IP address of the Docker instance (recall that you can run docker-machine ip default), and use it to browse to your server running at http://docker-ip:8080.

Before moving on, commit all of your changes to the Git repository by running these commands:

```
$ git add .
$ git commit -m "Ported to JRuby"
```

Now you can push these changes to the cloud. You could deploy the app to Heroku with the heroku container:release command you used in Chapter 2, *Creating a Deployment Environment*, on page 17, but there are some drawbacks to this technique when deploying an app like Twitalytics. Deploying an app that isn't packaged into a WAR file could result in unwanted changes or dirty artifacts being accidentally included in the Docker image. And if corruption isn't bad enough, uploading a Docker image can take a long time because it includes the entire app environment. To avoid this, you can push only the app's source code to Heroku and let the magic happen in the cloud—saving you time and bandwidth. Let's deploy your Git repository.

Deploying to the Public Cloud

The first step in deploying to a new production environment is making sure it exists! Create a new Heroku app for Twitalytics using the CLI:

```
$ heroku create
Creating calm-ocean-4238... done, stack is cedar-14
https://calm-ocean-4238.herokuapp.com/ | https://git.heroku.com/...
Git remote heroku added
```

The new app on the Heroku servers includes its own Git repository. This Git repository is very similar to a GitHub repo you might push to, but it lacks the fancy user interface. This repo isn't for collaborating—it's for deployment. You can view the location of the repo by running the following command:

```
$ git remote -v
heroku  https://git.heroku.com/calm-ocean-4238.git (fetch)
heroku  https://git.heroku.com/calm-ocean-4238.git (push)
```

Now you can deploy your local code repository to Heroku by running this command:

```
$ git push heroku jruby:master
Counting objects: 218, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (136/136), done.
Writing objects: 100% (218/218), 36.63 KiB | 0 bytes/s, done.
```

```
Total 218 (delta 72), reused 208 (delta 67)
remote: Compressing source files... done.
remote: Building source:
remote:
remote: -----> Ruby app detected
...
remote: Verifying deploy.... done.
To https://git.heroku.com/calm-ocean-4238.git
* [new branch]      master -> master
```

When Heroku receives the code, it executes a Git hook that triggers the build process. First, it detects that your application is a Ruby app because it has a Gemfile. Then it detects that you're using JRuby because of the ruby entry you added to the Gemfile. Finally, it runs bundle install and rake assets:precompile before deploying your code into a new dyno, which is a Heroku container equivalent to a Docker container.

When the deployment is complete, you can run the migrations on Heroku by executing this command:

```
$ heroku run rake db:migrate
```

And you can view your app by running this command:

```
$ heroku open
```

A browser will open to the URL of your app and you'll see Twitalytics running in the cloud.

You can also scale the app, view logs, and change configuration settings just as you did with the Warbler application you deployed with Docker in Chapter 2, *Creating a Deployment Environment*, on page 17.

Deploying to Heroku is easy and powerful. But not every organization will want to deploy to the public cloud. In the next section, you'll learn how to deploy this app to a private server.

Deploying to Private Infrastructure

Deploying to private infrastructure or even public infrastructure as a service (IaaS), such as Amazon Elastic Compute Cloud (EC2), doesn't require abandoning the technologies you've just learned about. In fact, your existing Docker container is well suited for deployment on a custom-built server.

Docker is only a part of the solution, though. Docker provides the containerization layer that isolates your processes and makes it possible to scale easily.

But you still need to orchestrate and manage the many containers you'll run. For that, you'll use Rancher.[10]

Rancher is open source software that makes it possible to deploy and orchestrate private Docker containers. It provides all of the necessary infrastructure services, including networking, load balancing, and storage, to ensure an application runs well on any kind of infrastructure. In other words, you can use Rancher to build your own platform as a service for use in a private organization. Rancher is one of many products on the market that can do this, but it's unique in that it preserves the native Docker experience. It includes support for the Docker CLI, Docker API, Docker Swarm, Docker Machine, and Docker Compose. Other products tend to wrap Docker's functionality and present an alternate developer experience.

You'll deploy the Docker image you created in *Creating the Deployment Environment*, on page 36 to a Rancher server. Because every infrastructure environment is different, you'll simulate a private server using Vagrant, a tool for managing virtual machines. Vagrant will allow you to run Rancher in a local virtual server, but the same method will apply on any infrastructure.

Installing Vagrant and VirtualBox

Rancher includes native support for a number of different configuration management tools, such as Puppet and Ansible, which you may want to use when you install it on your own infrastructure. But it also comes with a preconfigured Vagrant environment, which is what you'll use to run it. You'll start by installing VirtualBox, which is Vagrant's only dependency.

VirtualBox is a virtualization platform that lets you run a guest operating system inside your primary operating system. To install it, go to virtualbox.org,[11] and download and run the installer. The VirtualBox user interface will open at the end of the installation, but you can close it. You're going to drive VirtualBox with Vagrant.

To install Vagrant, download the binary installer for your operating system from the official website[12] and run it. The installer adds a `vagrant` command to your path, so you can check that both Vagrant and its connection to VirtualBox are working by running the following:

```
$ vagrant --version
Vagrant 1.8.1
```

10. http://rancher.com/
11. https://www.virtualbox.org/wiki/Downloads
12. https://releases.hashicorp.com/vagrant/1.8.1/

Now you can use Vagrant to run Rancher.

Installing the Rancher Service

You'll be working with the Rancher source code, so you'll need to start by cloning the Rancher project from GitHub. Run these commands now:

```
$ git clone https://github.com/rancher/rancher
$ cd rancher
```

Then open the project's Vagrantfile in your preferred editor and add this line of code to the config.vm.define block (just before the other 'forwarded_port' directive).

```
server.vm.network 'forwarded_port', guest: 3000, host: 3000
```

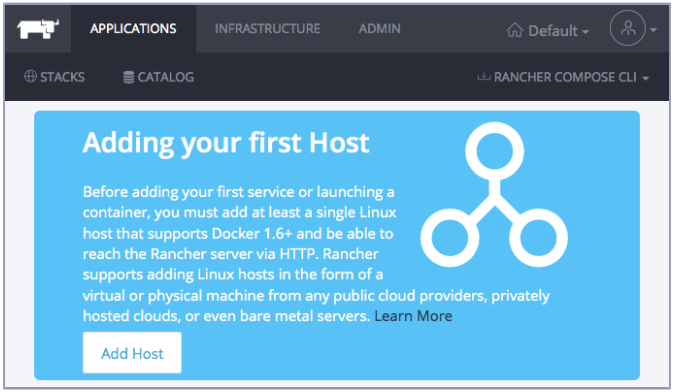
The Vagrantfile instructs the Vagrant runtime on how to prepare the box. Your change to the file adds a mapping from port 3000 on the Vagrant box to port 3000 on the host machine so you can access your web app when it runs inside the virtual machine.

Now start the Vagrant box and provision it by running this command:

```
$ vagrant up
Bringing machine 'rancher-server' up with 'virtualbox' provider...
==> rancher-server: Importing base box 'rancherio/rancheros'...
==> rancher-server: Matching MAC address for NAT networking...
==> rancher-server: Checking if box 'rancherio/rancheros' is up to date...
...
```

The first time you run the command will take a while because the server needs to download the various image layers it needs. But those images will be cached, which will make subsequent runs much faster.

Once the provisioning process is complete, the Rancher management server will be up and running. Test it by browsing to http://172.19.8.100:8080 and you'll see the Rancher dashboard as shown in the following figure.



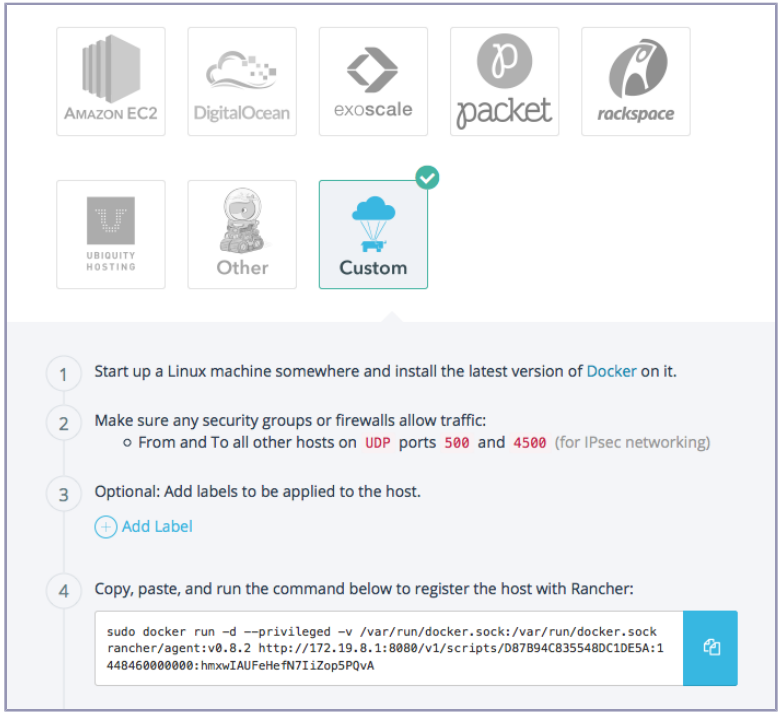
Now you can use this dashboard to configure your app.

Adding a Host

The Rancher deployment model consists of one or more hosts running one or more Docker containers each. These may include containers for web apps, background workers, databases, caching services, and whatever else your application architecture is composed of. In this example, you'll run a single host with a single container.

The type of host you use will differ depending on your infrastructure. You can create a host on AWS, DigitalOcean, and many other providers. You'll use the custom server setup to run a container on the same server as the Rancher management service (that is, the Vagrant box).

From the Rancher UI in your browser, select the Add Host button. You'll be prompted to configure the base URL all hosts should use to connect to the Rancher API. Accept the default value, which should be the same as the address you opened in the browser. Then click Save. From the server choices that follow, select Custom, as shown in the figure.



At the bottom of the page is a long `docker` command. Copy this command to your clipboard. You'll run it in the Vagrant box in a just a moment.

To run commands on the Rancher server, you'll need to log in to the Vagrant box by running this command:

```
$ vagrant ssh
```

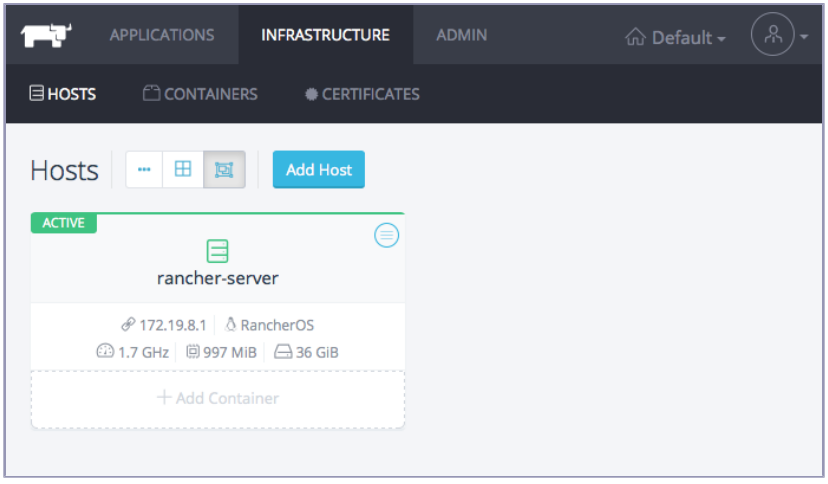
This command is synonymous with running an `ssh` command to log in to a remote DigitalOcean or AWS instead of Vagrant. But once you're in, the steps that follow are essentially the same. When the command completes, you'll see an interactive prompt like this:

```
[rancher@rancher-server ~]$
```

At this prompt, enter or paste the command you copied from the browser. But remove the `sudo` portion and add the option `-e CATTLE_AGENT_IP=172.19.8.100` immediately after the `docker run` portion. The IP address in the option should match the IP address of the Vagrant box, which is the same as what you used in the browser. The complete command will look like this:

```
[rancher@rancher-server ~]$ docker run -e CATTLE_AGENT_IP=172.19.8.100 \
-d --privileged -v /var/run/docker.sock:/var/run/docker.sock \
rancher/agent:v0.8.2 http://172.19.8.1:8080/v1/scripts/...
```

When this completes, the host will be running. Return to the dashboard and view the Hosts page. You'll see your rancher-server as shown here.



Exit the Vagrant box by running `exit`, and return to your local prompt. Now you must add a container to the host. This is the actual deployment step.

Deploying a Container

Deploying a Docker container requires publishing your Docker image to a registry service. There are many of registries to choose from—including privately hosted ones—but we'll use Docker Hub, which is run by Docker, Inc.

Create a Docker Hub account by browsing to https://hub.docker.com/ and completing the sign-up form. Once your account has been created, use your credentials to log in from the Docker CLI client by running this command:

```
$ docker login
```

Now, open a terminal and move to the Twitalytics directory:

```
$ cd ~/code/twitalytics
```

Open the project's Dockerfile in an editor and add this line of code to the end of the file:

```
ENTRYPOINT ["bin/puma", "-C", "config/puma.rb"]
```

The ENTRYPOINT is the command that Docker will use to launch your app when you start a container from the image. You'll notice that the command you've entered is identical to the one you put in the Procfile when deploying to Heroku. Now build a new image by running this command (but replace username with your Docker Hub username):

```
$ docker build -t username/twitalytics .
```

The username/twitalytics image contains a JVM, JRuby, and your app. Deploy this image to Docker Hub by running the following command:

```
$ docker push username/twitalytics
```

It will take some time to upload, but when it's complete, you can view the image on your dashboard at https://hub.docker.com/. With the image published, you can now consume it from your Rancher host.

Move back to the rancher directory and run vagrant ssh again to start a shell session in the rancher-server box. From the prompt, run the following command to pull the image you just deployed:

```
[rancher@rancher-server ~]$ docker pull username/twitalytics
```

Before you can run this image, you'll need a database. You could run another container with your database instance and share it among the web containers. That would be fine on a non-virtualized environment, but it would probably put too much strain on this virtual machine running on your desktop. Instead,

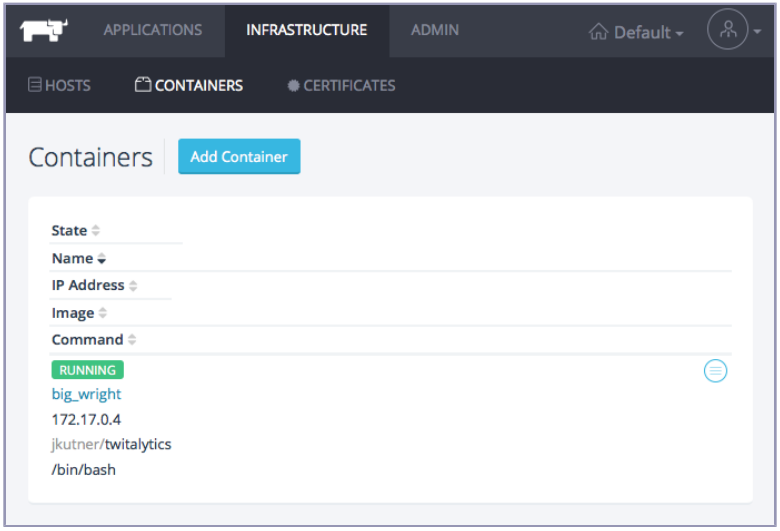
you can use the Heroku database provided by your Heroku app. To get the connection parameters run this command:

```
$ heroku config:get DATABASE_URL
postgres://user:password@host:port/db
```

Now you can launch a container from your Twitalytics image and use the database URL from the previous step as one of the environment variables. You could do this in the Rancher UI, but we'll do it from the Vagrant prompt. Run this command:

```
[rancher@rancher-server ~]$ docker run \
-e DATABASE_URL=postgres://user:password@host:port/db \
-e PORT=3000 --publish=3000:3000 \
-dit username/twitalytics
```

The container will start in the background and Twitalytics will bind to port 3000, which will be published to the Docker host so that you can access it. Now return to the Rancher UI and browse the list of containers on the server. You will see one container in a running state, like this:



Open a browser to http://172.19.8.100:3000 and you'll see Twitalytics running on your Rancher host.

Your Rancher deployment is ready for production. But any non-trivial production app will require the ability to scale up in order to provide redundancy and handle increases in traffic. Because the Rancher console and host are running on your own private infrastructure, scaling up might require the procurement of more servers—either virtual or physical. Fortunately, the Rancher and Docker architecture you've created is well suited for scaling.

You can add hosts from different types of platforms—even combining AWS, DigitalOcean, and your own private servers. As long as the Rancher API can reach the server, you can fold it into your system.

Wrapping Up

Not only have you ported Twitalytics to JRuby and provisioned a fully functioning local deployment environment, but you've also deployed it to the cloud. You have the ability to scale Twitalytics up without an in-house operations team that must purchase new hardware. This deployment setup also makes it easy to debug production problems in a controlled local setting by using a Docker container.

If the cloud isn't for you and your organization, that's OK too. Docker allows you to deploy the same application image to Heroku, AWS, DigitalOcean, or your own private servers. You even learned how to use Rancher to orchestrate a heterogeneous suite of servers.

The skills you learn in this chapter are highly desired in our industry. Many apps need porting to JRuby, and they all need to be deployed. You now have the ability to service some of the most powerful Ruby apps in the world.

In the next chapter, you'll build on this skill set by adding more capabilities to Twitalytics. You'll enhance your Docker setup by adding caching, background jobs, and more.

CHAPTER 4

Consuming Backing Services with JRuby

Twitalytics requires some new features that trigger each time a post is created. It must retain a per-user count of new posts, annotate each new post with the stock price markup you implemented in Chapter 1, *Getting Started with JRuby*, on page 1, and stream this information back to its clients.

Each feature adds a challenging concern to Twitalytics. One is stateful, one consumes a microservice over the network, and one must send the same data across multiple connections. None of these features are well suited to the synchronous request-response cycle of a traditional web application, which is why you'll need to implement these features with the help of some backing services.

In this chapter, you'll learn how to connect Twitalytics to Memcached for session storage, Redis for running background jobs, and RabbitMQ for message passing. These services will come in handy not only for Twitalytics but also for every app you work on in the future.

While the concept of a backing service may seem new, you've actually been using one this entire book. Let's take a look at what this term means.

What Are Backing Services?

A backing service is a resource an app consumes over the network as part of its normal operation. Your database is an example of a backing service, and so is the Yahoo! stock service you consumed in Chapter 1, *Getting Started with JRuby*, on page 1. Backing services do many different kinds of jobs, including storing data, sending email, and indexing text documents.

Until recently, most developers treated backing services as tightly coupled extremities of an application. The same system administrators who managed the app would also manage the backing services and even run them on the

same physical machine as the app itself. But in the era of DevOps, the cloud, and containerization, more organizations are treating backing services as third-party resources—even when they're managed in-house. A third-party service is decoupled from an application, runs in an isolated environment, and exposes its functionality via URLs that can be attached, detached, or replaced at any moment.

Treating backing services as third-party resources also makes consuming them from JRuby nearly identical to consuming them from any other Ruby deployment. In most cases, you use the same server technologies and even the same popular client libraries.

Let's begin with a service that you'll probably use for every request to your web app.

Storing Sessions in Memcached

Twitalytics needs to track how many posts users create between the time they log in and the time they log out. This count is stateful and must be carried across transactions, survive any restarts of the application process, and be accessible from multiple processes when the app is scaled out. For these reasons, the count must be stored in a user's session.

Each time a user starts an interaction with a web page, a session is created. The session stores state that's carried over from one request to another for the same user. It usually includes things like username, breadcrumbs to track where they've been in the app, and even security tokens.

The default session storage mechanism in Rails is cookie based, which means the session state is stored on the client machine. This is an ideal place to put sensitive information, and it has a limited storage capacity. A better system will store session state server-side.

When storing session state on the server, you have the option to keep it in memory or in an external backing service. Keeping the session in memory is convenient, but it has some serious drawbacks. If the server process is restarted, all users will lose their current state. They may even need to log into the app again. If they were about to make a credit card transaction, they would be most unhappy.

But storing session state in memory is also a problem for scalability. In-memory session data cannot easily be distributed among multiple processes. If you need to stand up additional instances of your server to handle high volumes of traffic or ensure redundancy, you'll be in for trouble.

The best way to store session state, even for the simplest of apps, is in a backing service. One example is Memcached, which you'll use for Twitalytics. Memcached is a free, open source, high-performance, distributed-memory, object-caching system. It's useful as a key-value store to cache results of database calls, API calls, page rendering, and session state.

Installing Memcached

The easiest way to run Memcached locally is with Docker. Download the official Memcached image from DockerHub by running this command:

```
$ docker pull memcached
Using default tag: latest
latest: Pulling from library/memcached
dbacfa057b30: Pull complete
...
Digest: sha256:b335e191aac685a7ee7b9e3b4bfceef184f315412733f1ea099463fc5dcdb25e
Status: Downloaded newer image for memcached:latest
```

Now launch a new Docker container from the image and publish port 11211 by running this command:

```
$ docker run -p 11211:11211 --name memcached-server -d memcached
```

The server was started in a container that's running in the background. You can check its status with `docker ps`:

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND                  ...
3d6a392002f6        memcached           "/entrypoint.sh memca"   ...
```

And you can use `telnet` to test that Memcached itself is working. If you're not running Linux, you'll need to capture your Docker Machine IP address first. Run these commands:

```
$ docker-machine ip default
192.168.99.100
```

```
$ telnet 192.168.99.100 11211
Trying 192.168.99.100...
Connected to 192.168.99.100.
Escape character is '^]'.
```

Memcached is ready to store your session state. Enter `quit` at the prompt to end the Telnet session. Now connect Twitalytics to the Memcached server.

Using Memcached with Rails

Before you make any changes to Twitalytics, branch your code base by running these commands:

```
$ cd ~/code/twitalytics
$ git checkout -b services
Switched to a new branch 'services'
```

Connecting to Memcached from any kind of Ruby code requires a client library that knows how to speak the Memcached protocol. The de facto standard in the Ruby ecosystem is Dalli,[1] which is one of the many modern Ruby gems that works equally well with MRI and JRuby. To install Dalli, add these lines to the Twitalytics Gemfile:

Services/twitalytics/Gemfile

```
gem 'dalli'
gem 'connection_pool'
```

The first gem is Dalli itself. The second gem, connection_pool, is what Dalli uses to pool Memcached connections, which ensures the Rails.cache singleton doesn't become a source of thread contention. This is important when using JRuby because it's a multithreaded runtime.

Save the Gemfile and run these commands to download and install the gems.

```
$ bundle install --binstubs
$ bin/rake rails:update:bin
```

Next, configure Rails to use Dalli as the default caching mechanism. Open the config/environments/development.rb file and add this line of code inside the Rails.application.configure block (but use the IP address of your Docker Machine in place of the IP address shown here):

Services/twitalytics/config/environments/development.rb

```
config.cache_store = :dalli_store, "192.168.99.100"
```

This sets Rails to use the Dalli client for all caching purposes in the app. But you also need to configure the session storage mechanism to use the cache instead of cookies. Open the config/initializers/session_store.rb file, and replace its contents with this code:

Services/twitalytics/config/initializers/session_store.rb

```
Rails.application.config.
  session_store :cache_store, key: '_twitalytics_session'
```

Now you can add the feature that tracks how many posts a user has created in a given session. Open the app/controllers/posts_controller.rb file, and add these lines of code to the create() method:

1. https://github.com/petergoldstein/dalli

Services/twitalytics/app/controllers/posts_controller.rb

```
count = session[:count] || 0
session[:count] = count + 1
```

This increments a counter each time the Post#create action is executed.

To show the current count, open the app/views/posts/index.html.erb file, and add this code above the main table:

Services/twitalytics/app/views/posts/index.html.erb

```
<p>
  Created: <%= session[:count] || 0 %>
</p>
```

Now start the Puma server, and browse to http://localhost:3000/posts. You'll see the count set at zero. Create a few posts, and you'll see the count increase. To confirm that the values are being stored in Memcached, check it with Telnet. Start a Telnet session and run the stats items command like this:

```
$ telnet 192.168.99.100 11211
Trying 192.168.99.100...
Connected to 192.168.99.100.
Escape character is '^]'.
stats items
STAT items:5:number 1
STAT items:5:age 5
STAT items:5:evicted 0
STAT items:5:evicted_nonzero 0
STAT items:5:evicted_time 0
STAT items:5:outofmemory 0
STAT items:5:tailrepairs 0
STAT items:5:reclaimed 0
STAT items:5:expired_unfetched 0
STAT items:5:evicted_unfetched 0
STAT items:5:crawler_reclaimed 0
STAT items:5:crawler_items_checked 0
STAT items:5:lrutail_reflocked 0
```

This displays an overview of the *items* in the cache. The number after the keyword items is the slab ID of the record. You can dump the record by running this command in the Telnet session:

```
stats cachedump 5 100
ITEM _session_id:fff8686f323aa547c936649e5a2f2131 [88 b; 1454260137 s]
```

It shows the item is keyed by a _session_id. The data isn't readable here, but it's enough to confirm that Rails is writing its session data to Memcached.

Your backing service is almost ready for production. The only problem is the hardcoded IP address of your Docker Machine in the configuration file and the lack of configuration for authentication credentials, which you'll need to secure your cache in production.

The IP address of the Memcached server and its credentials will be environment specific. That is, they will change depending on if the app is running locally, in test, or in production. For that reason, it's necessary to extract this information from environment variables rather than hard-coding it.

Open the config/environments/production.rb file, and add this code to the Rails.application.configure block:

Services/twitalytics/config/environments/production.rb

```
if ENV["MEMCACHEDCLOUD_SERVERS"]
  config.cache_store = :dalli_store,
                       ENV["MEMCACHEDCLOUD_SERVERS"].split(','),
                       { :username => ENV["MEMCACHEDCLOUD_USERNAME"],
                         :password => ENV["MEMCACHEDCLOUD_PASSWORD"],
                         :pool_size => ENV["MAX_PUMA_THREADS"] || 1 }
end
```

The MEMCACHEDCLOUD_SERVERS environment variable can contain multiple IP addresses because in production you'll want some kind of failover for this service. The other environment variables provide the username and password. MemcachedCloud is a cloud-based Memcached as a service provider that you'll use in a moment, which is why we've chosen the MEMCACHEDCLOUD_ prefix for these variables. The last parameter is the connection pool size, which is set to the same value as Puma's maximum thread count.

Shut down the Twitalytics server by pressing Ctrl-C. Then commit all of your changes to Git by running these commands:

```
$ git add .
$ git commit -m "memcached"
```

Now you can deploy.

Deploying with Memcached

The code you used in development is ready to run on Heroku. You just need to add a Memcached service to your app by running the following command:

```
$ heroku addons:create memcachedcloud:30
```

This creates a free MemcachedCloud backing service attached your Heroku app. It also sets the MEMCACHEDCLOUD_SERVERS environment variable and the

other environment variables you need to connect to it with the configuration in your config/environments/production.rb file.

Now deploy the app by running this command:

```
$ git push heroku services:master
```

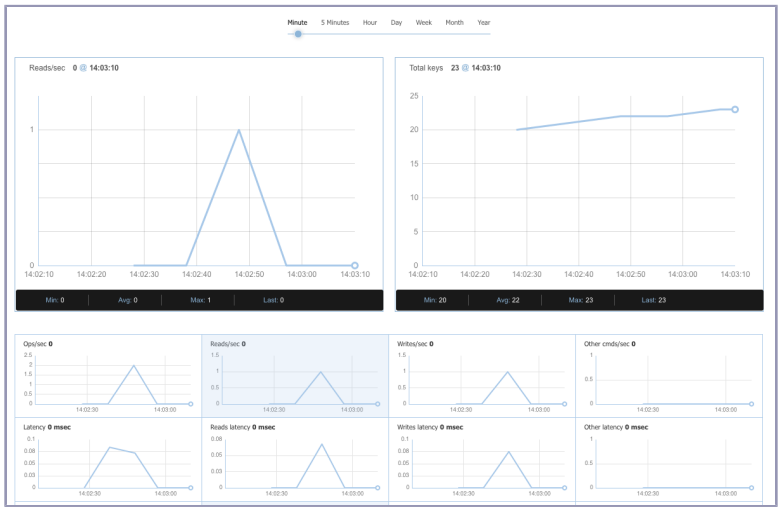
When the build is finished, open the Posts page with this command:

```
$ heroku open posts
```

Make a few requests to ensure that the session is exercised. Then verify that the session data is getting to Memcached by viewing the MemcachedCloud dashboard. Run this command to open it:

```
$ heroku addons:open memcachedcloud
```

From the dashboard, drill down into the Advanced Metrics view, and you'll see something like the following figure:



If you're not running on Heroku, you can use the Docker container you ran locally with the Rancher setup you created in Chapter 3, *Deploying a Rails Application*, on page 29. Log in to your Rancher virtual server by running vagrant ssh. Then pull the Docker image and run a new container just as you did locally. When you start your application containers, add the appropriate environment variable to the command options like this:

```
[rancher@rancher-server ~]$ docker run \
-e MEMCACHEDCLOUD_SERVERS=192.168.99.100:11211 \
-e DATABASE_URL=postgres://user:password@host:port/db \
-e PORT=3000 --publish=3000:3000 \
-dit username/twitalytics
```

In this case, you're reusing the MEMCACHEDCLOUD_SERVERS variable name, but you can use any name you'd like.

Memcached is an essential service—similar in importance to a relational database. Almost every web application will need to store session data, and doing so with a backing service ensures better failover and scalability. The next service you'll deploy is nearly as essential.

Running Background Jobs with Sidekiq

Twitalytics needs to annotate the text in each post with the stock information from the stock-service you created in Chapter 1, *Getting Started with JRuby*, on page 1. Invoking the service could take a long time, which makes this interaction a good candidate for a background job.

You've already learned how to use the JVM's concurrency features to perform asynchronous request processing. The power of a true multithreaded platform makes it possible to increase throughput by running operations in parallel, but there are still many cases when you may want to move some computation into a separate process. This is true for any compute-intensive job that might steal CPU time from the request threads. It's also useful for communicating with processes running behind a firewall or on platforms other than the JVM.

Sidekiq is a popular background-processing library for Ruby and JRuby. It uses Redis[2] to communicate between the process that creates a background job and the processes that execute the background job. Redis, in this case, is a backing service. It's a distributed in-memory data structure store, similar to Memcached, but Sidekiq uses it as a message broker.

Installing Sidekiq and Redis

You can run Redis locally with Docker much as you did with Memcached. Download the DockerHub Redis image by running this command:

```
$ docker pull redis
Using default tag: latest
latest: Pulling from library/redis
...
678a090a2546: Pull complete
Digest: sha256:de86bd14ab69c9b707fe5f3213f6e3c6f543df28bc05ae6cef7b61f2b12be343
Status: Downloaded newer image for redis:latest
```

Then launch a container from the image, and publish port 6379 by running this command:

2. http://redis.io/

```
$ docker run -p 6379:6379 --name redis-server -d redis
```

A Redis server starts in the background. You can test it by running a Redis command-line client in a separate container. You'll need to know the Docker Machine IP address, so execute these commands:

```
$ docker-machine ip default
192.168.99.100
```

```
$ docker run -it --rm redis sh -c 'redis-cli -h 192.168.99.100 -p 6379'
192.168.99.100:6379>
```

From the prompt, you can ensure the server is healthy with the ping command:

```
192.168.99.100:6379> ping
PONG
```

The PONG response means everything is working. Enter QUIT at the prompt to close the session.

Redis is only a part of the architecture you need to run background jobs. You also need to install Sidekiq, which is a Ruby gem. Add this line to your Gemfile:

Services/twitalytics/Gemfile

```
gem 'sidekiq'
```

Save the file and run these commands to download and install the new dependency:

```
$ bundle install --binstubs
$ bin/rake rails:update:bin
```

Sidekiq needs to know how to find the Redis server that backs it up. It starts by looking for the REDIS_URL environment variable and defaults to localhost if it's not set. But you point it to your Docker container as the default. Create a config/initializers/sidekiq.rb file with the following code, but replace the IP address with your Docker Machine's IP address:

Services/twitalytics/config/initializers/sidekiq.rb

```
Sidekiq.configure_server do |config|
  config.redis = { url: ENV['REDIS_URL'] || 'redis://192.168.99.100:6379' }
end

Sidekiq.configure_client do |config|
  config.redis = { url: ENV['REDIS_URL'] || 'redis://192.168.99.100:6379' }
end
```

This configures both the client (the process publishing messages) and the server (the background worker receiving them) to use the REDIS_URL environment variable when it's set and Docker Machine otherwise.

Now you're ready to run a background job with Sidekiq and Redis.

Creating a Rails Background Job

Sidekiq jobs are encapsulated in Worker classes that have a perform() method. By convention, Sidekiq workers are housed in the app/workers directory. Create this directory and add a file to it called posts_worker.rb by running this:

```
$ mkdir -p app/workers
$ touch app/workers/posts_worker.rb
```

Now open the posts_worker.rb file and put the following code in it:

Services/twitalytics/app/workers/posts_worker.rb

```
class PostsWorker
  include Sidekiq::Worker

  def perform(post_id)
    post = Post.find(post_id)
    url = ENV["STOCK_SERVICE_URL"] || "localhost:8080"
    host = url.split(":")[0]
    port = url.split(":")[1]
    Net::HTTP.start(host, port) do |http|
      http.request_post("/stockify", post.body) do |resp|
        post.update({ html: resp.body })
      end
    end
  end
end
```

The PostsWorker class includes the Sidekiq::Worker module and implements the perform() method. In the body of the perform() method it makes an HTTP request to the stock-service. The host and port for the service are determined from an environment variable but default to localhost:8080 if they're not set. Then it stores the HTML response from the stock-service in the database for the given Post object.

The PostsWorker does the heavy lifting, but you also need some code to trigger the background job. Sidekiq does this with the perform_async() method, which queues up the job on Redis for the worker process to retrieve. You'll execute this in the PostsController. Open the app/controllers/posts_controller.rb file, and put the following code in the body of the if statement conditional on the @post.save call in the create() method:

Services/twitalytics/app/controllers/posts_controller.rb

```
PostsWorker.perform_async(@post.id)
```

The call to perform_async() takes one argument, the Post ID, which is ultimately passed to the perform() method on the worker process.

Now test it out. Start the Puma server as normal, and open a second terminal session for the stock-service. Move into its directory, re-create the executable WAR file if necessary, and run it like this:

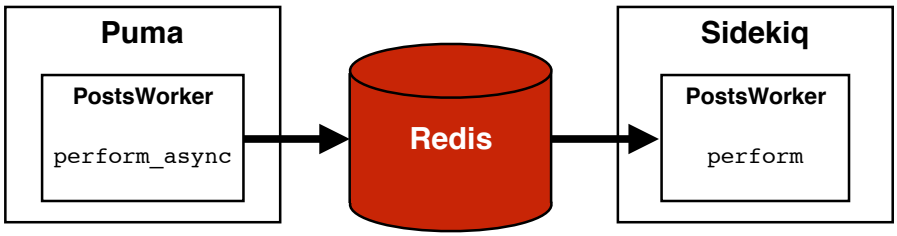
```
$ cd ~/code/stock-service
$ java -jar stock-service.war
```

Then open a third terminal session and start a Sidekiq worker by running this command:

```
$ bin/sidekiq
```

To exercise the job, point your browser to http://localhost:3000/posts and create a new Post instance. When you click the Save button, the job will trigger. You'll see the database interactions logged in the Sidekiq terminal. When that happens, return to the http://localhost:3000/posts page, and click the Show link for the post you just created. You'll see the HTML in pre-formatted text.

Let's review what happened at a high level because there are lots of moving parts in this example. As shown in the following figure, the web server creates a job by calling perform_async() on the worker class, which creates a record in Redis. The Sidekiq process detects a new record, retrieves it, and passes it to an instance of the PostsWorker class, which does the background work of making the HTTP request to the stock-service.



Shut down the Twitalytics server, the stock-service, and the Sidekiq process by pressing Ctrl-C in their respective terminals. Then commit all of your changes to Git by running these commands:

```
$ git add .
$ git commit -m "redis"
```

You're ready to put this system into production.

Deploying Sidekiq and Redis

Just as with Memcached, you can create a free Heroku Redis service in the cloud and attach it to your app. To do so, run this command:

```
$ heroku addons:create heroku-redis
```

This provisions a Redis add-on and sets the REDIS_URL environment variable on your app.

You need to set an environment variable defining the location of your production stock-service. From the Twitalytics app directory, run this command but replace the Heroku app name with the name of the Heroku stock-service app you created in Chapter 1, *Getting Started with JRuby*, on page 1:

```
$ heroku config:set STOCK_SERVICE_URL="obscure-fjord-4138.herokuapp.com"
```

This overrides the default *localhost* value used by the worker. Before moving on, make sure the stock-service Heroku app is still working by running this command:

```
$ curl -d "Hi Apple" http://obscure-fjord-4138.herokuapp.com/stockify
Hi <div class='stock' data-symbol='AAPL' data-day-high='86.73'>Apple Inc.</div>
```

Now define a new Procfile entry for your Sidekiq worker. Open the existing Procfile and replace its contents with this:

Services/twitalytics/Procfile

```
web: puma -C config/puma.rb
sidekiq: sidekiq -c 5
```

The web entry is the same as before, but the sidekiq entry is new. It uses the same command you ran locally but limits the number of threads in the worker to five. This prevents you from overrunning the maximum number of allowed connections on the Heroku Redis instance (don't forget that your web process is making connections as well). Add the file to Git and redeploy by running these commands:

```
$ git add Procfile
$ git commit -m "worker"
$ git push heroku services:master
```

When the build is finished, run heroku open posts to open the app. Then create a few new Post records. You won't see the background job execute, however, because the sidekiq process has not been scaled up yet. Run this command to start one instance of it:

```
$ heroku ps:scale sidekiq=1
```

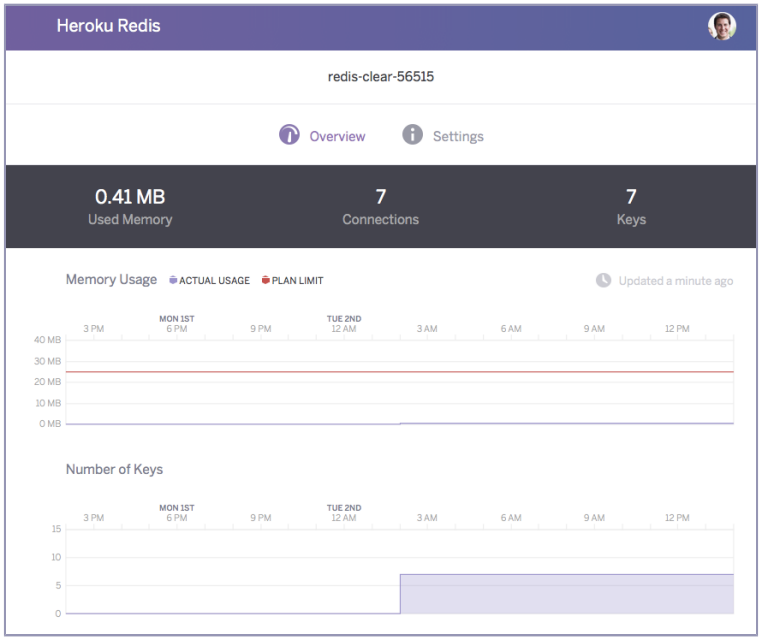
Then run the following command to tail the logs:

```
$ heroku logs -t
```

You'll see the background worker start up and request the HTML as you did locally. When you've finished, kill the logs by pressing Ctrl-C and open the Redis dashboard in a web browser by running this command:

```
$ heroku addons:open heroku-redis
```

You'll see a page like this:



The Redis dashboard displays the memory usage, number of keys, number of connections, and other data about your instance.

If you're not using Heroku for production, you can use the Docker container you ran locally with Rancher. Log in to your Rancher virtual server by running `vagrant ssh`. Then pull the Docker Redis image and run a new container just as you did in your local environment. When you start your application containers, add the appropriate environment variable to the command options like this:

```
[rancher@rancher-server ~]$ docker run \
-e MEMCACHEDCLOUD_SERVERS=192.168.99.100:11211 \
-e REDIS_URL=redis://192.168.99.100:6379 \
-e STOCK_SERVICE_URL=https://192.168.99.100:8080 \
-e DATABASE_URL=postgres://user:password@host:port/db \
-e PORT=3000 --publish=3000:3000 \
-dit username/twitalytics
```

To start a Sidekiq worker container, run this command:

```
[rancher@rancher-server ~]$ docker run \
-e MEMCACHEDCLOUD_SERVERS=192.168.99.100:11211 \
-e REDIS_URL=redis://192.168.99.100:6379 \
-e STOCK_SERVICE_URL=https://192.168.99.100:8080 \
-e DATABASE_URL=postgres://user:password@host:port/db \
-e PORT=3000 --publish=3000:3000 \
--entrypoint="bin/sidekiq"
-dit username/twitalytics
```

The `entrypoint` option tells Docker to start the Sidekiq process instead of the Puma server.

Sidekiq and Redis are easy to use and are probably the most common Ruby libraries for running background jobs. But they do have their drawbacks. Redis doesn't have a robust transaction mechanism. It's essentially a key-value store being repurposed as a work queue. A more advanced message queue should provide support for durability, routing, topics, remote procedure calls, and more. Let's move on and add these capabilities to Twitalytics.

Message Passing with RabbitMQ

The Twitalytics home page needs to shows the *stockified* HTML for new posts as they're created (in near real time). To implement this streaming behavior, you'll need a more powerful message broker than Redis. Twitalytics must be able to route messages to multiple specific clients, which will make it possible to stream updates to every browser with an open connection to the home page.

The messaging system you'll use is RabbitMQ, an open source message broker that implements the Advanced Message Queuing Protocol (AMQP). It's an ideal solution for building chat services, games, news apps that broadcast information, workflow engines, or any app with complex distributed computation requirements.

An important difference from Sidekiq is RabbitMQ's ability to send messages through a router, which allows it to send a single message to all browsers with an open connection to the Twitalytics home page. But this model, which is defined by AMQP, is a bit more difficult to understand than the Sidekiq and Redis architecture. Instead of sending messages directly to a queue, a RabbitMQ client sends messages to an exchange, which routes the messages to one or more queues.

There are different ways of binding exchanges to queues, and it's important that you understand the differences between them to choose the right pattern for streaming in Twitalytics. The various patterns are described here:

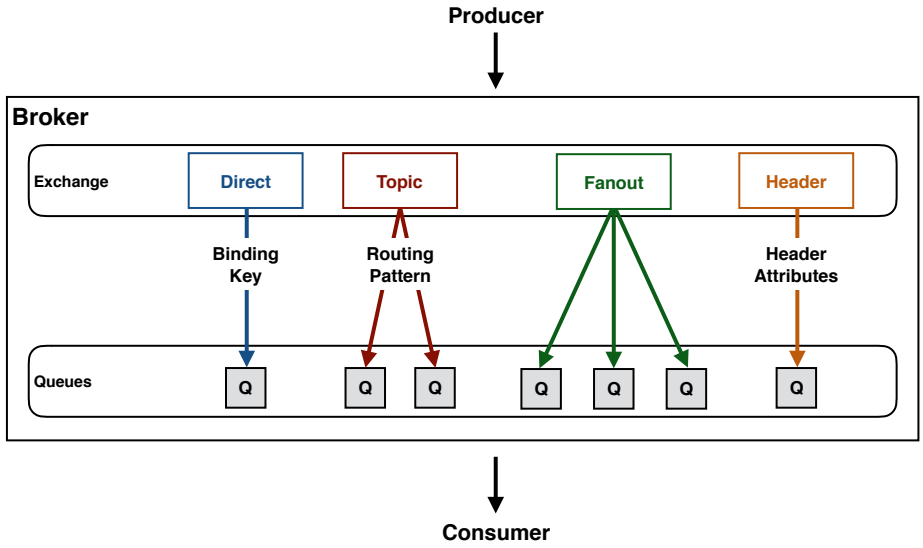
Direct A direct exchange delivers messages to queues based on a message routing key. Messages are routed to the queues whose binding key exactly matches the routing key of the message.

Topic A topic exchange does a wildcard match between the routing key and the routing pattern specified in the binding.

Fanout A fanout exchange routes messages to all of the queues that are bound to it.

Header Header exchanges use the message header attributes for routing. The attributes don't have to be strings. They could be integers or a hashes, for example, which makes the routing capability more powerful.

These patterns are illustrated in the following figure.



For Twitalytics's streaming feature, the fanout pattern is ideal because it can route a message to multiple queues, which can in turn stream the message to multiple browsers.

RabbitMQ has many Ruby clients, but not all of them take advantage of the JVM's powerful concurrency libraries. Fortunately, the ruby-amqp organization on GitHub[3] maintains a gem called March Hare[4] that wraps the RabbitMQ

3. https://github.com/ruby-amqp
4. http://rubymarchhare.info/

Java client.[5] This allows it to piggyback off the power and maturity of the underlying library.

Installing RabbitMQ and March Hare

It should come as no surprise that you can run RabbitMQ locally with Docker, just as you did with Memcached and Redis. But the RabbitMQ image has a bit more to it because its data isn't stored in memory. RabbitMQ supports persistent messages, which are written to disk to ensure they survive restarts. Download the Docker Hub RabbitMQ image by running this:

```
$ docker pull rabbitmq
Using default tag: latest
latest: Pulling from library/rabbitmq
...
```

Then start a container from the image and publish port 5672 by running the following command:

```
$ docker run -d -p 5672:5672 --hostname rmq1 --name rabbitmq-server rabbitmq
```

This makes port 5672 accessible, but it also provides an explicit `hostname` option. RabbitMQ uses the `hostname` as a unique key when storing data, which means you don't want to use the random value assigned to it by Docker.

Run this command to check the status of the server:

```
$ docker logs rabbitmq-server
...
=INFO REPORT==== 4-Feb-2016::15:43:10 ===
node           : rabbit@rmq1
home dir       : /var/lib/rabbitmq
config file(s) : /etc/rabbitmq/rabbitmq.config
cookie hash    : 10c+lZTnE9sNU/p1S3987w==
log            : tty
sasl log       : tty
database dir   : /var/lib/rabbitmq/mnesia/rabbit@rmq1
..
```

In the info report you'll see the `database dir`, which uses the `hostname` value as part of the path.

Now install March Hare by adding this line to your `Gemfile`:

Services/twitalytics/Gemfile

```
gem 'march_hare'
```

5. http://www.rabbitmq.com/api-guide.html

Then run Bundler to download and install it, and run Rake to regenerate your Rails binstubs:

```
$ bundle install --binstubs
$ bin/rake rails:update:bin
```

To create a connection to RabbitMQ, use a Rails initializer. Create a file named config/initializers/rabbitmq_connection.rb and put the following code in it:

Services/twitalytics/config/initializers/rabbitmq_connection.rb

```
Rails.application.config.after_initialize do
  $bunny = MarchHare.connect(
    :heartbeat => 5,
    :uri => ENV["RABBITMQ_URL"] ||
            ENV["CLOUDAMQP_URL"] ||
            "amqp://192.168.99.100:5672")
end

at_exit do
  $bunny.close
end
```

This creates a new global connection to RabbitMQ, called $bunny, using either the RABBITMQ_URL or CLOUDAMQP_URL environment variable. But it falls back to the hardcoded value of your Docker Machine IP address if neither of those is present. It also adds an at_exit procedure to close the connection when Rails shuts down.

Now you're ready to publish and consume some messages.

Using March Hare with Rails

Each time a post is created, Twitalytics needs to publish a message to notify clients that their view should be updated. Because Twitalytics already has a Sidekiq background job that runs every time a post is created, you already have an excellent trigger for this mechanism. You can publish the RabbitMQ message from the Sidekiq worker.

Open the Sidekiq worker you created earlier in the file app/workers/posts_worker.rb and add this code to the end of the perform() method:

Services/twitalytics/app/workers/posts_worker.rb

```
channel = $bunny.create_channel
exchange = channel.fanout("twitalytics.posts")
exchange.publish(post.html)
```

This creates a new channel from the $bunny connection. A channel is a virtual connection within a physical connection. You can have more than one channel

per connection. Then it gets a handle to a fanout exchange called twitalytics.posts and publishes a message to it containing the HTML.

That completes the publisher. Now you need to create the subscriber. Open the config/initializers/rabbitmq_connection.rb file again, and add this code to the end of the after_initialize() block:

Services/twitalytics/config/initializers/rabbitmq_connection.rb

```
channel = $bunny.create_channel
exchange = channel.fanout("twitalytics.posts")
queue = channel.queue("").bind(exchange)
```

This is very similar to the code in the worker. It creates a channel and gets a handle to the twitalytics.posts fanout exchange but then binds a queue to the exchange. This will cause all messages sent to the twitalytics.posts exchange to be routed to the given queue.

Next, you need to subscribe to the queue. You do this with subscribe(), which runs in the background and executes a block of code each time it receives a message. In this case, the block of code will write to a list of HTTP response streams (which you'll create in a moment). Put the following code immediately after the code that creates the queue:

Services/twitalytics/config/initializers/rabbitmq_connection.rb

```
$streams = Concurrent::Array.new
consumer = queue.subscribe do |metadata, payload|
  $streams.reject! do |stream|
    begin
      stream.write("data: #{payload}\n\n")
      false
    rescue IOError => e
      stream.close
      true
    end
  end
end
```

The $streams variable is a global list of HTTP response streams. By iterating over this list, you prevent the app from having one subscriber per HTTP request, which could quickly overrun the system. In the body of the subscribe() method's block, you'll write a string to the response streams and trap any IOErrors, which would result from the connection being closed by the client. If there is an error, you'll remove the stream from the list.

The response streams will run indefinitely in the background, which means they need to be closed when the process exits. Otherwise, the server will hang.

At the end of the config/initializers/rabbitmq_connection.rb file, after the at_exit() block, add this code to close the streams:

Services/twitalytics/config/initializers/rabbitmq_connection.rb

```
Signal.trap("INT") do
  $streams.each(&:close)
  raise Interrupt.new
end
```

This traps the SIGINT signal, closes each stream, and raises an Interrupt error to shut down the server naturally.

The last piece of server-side code will go in a new controller. Run this command to generate it:

```
$ bin/rails generate controller stream index \
    --assets=false --helper=false
```

This creates an app/controllers/stream_controller.rb file with an index() method. Open the file and replace its contents with the following code:

Services/twitalytics/app/controllers/stream_controller.rb

```
class StreamController < ApplicationController
  include ActionController::Live

  def index
    response.headers["Content-Type"] = "text/event-stream"
    $streams << response.stream
    response.stream.write("\n")
  end
end
```

At the beginning of the class, you included the ActionController::Live module, which enables live streaming in Rails. In the body of the index() method, you set a response header indicating to the client that the service is an event stream. You added the response stream to the global list of streams, and you're writing a newline character to the stream (this is effectively a no-op, because the service will error out if you don't write anything).

Because the service is persistent (that is, it will remain open while you do other things), you'll need to enable concurrent requests in Rails's development mode. Add these lines of code to the config/environments/development.rb file:

Services/twitalytics/config/environments/development.rb

```
config.preload_frameworks = true
config.allow_concurrency = true
```

Now you can write the client code that consumes the streaming service. Open the public/index.html file and add this code to the end of the <body> element:

Services/twitalytics/public/index.html

```
<div id="posts"></div>
```

This is the container where you'll display the posts as they're streamed from the server.

To connect to the stream, add the following code to the <head> element:

Services/twitalytics/public/index.html

```
<script>
  $( document ).ready(function() {
    var source = new EventSource('/stream/index');
    source.onmessage = function(event) {
      $("#posts").append("<p>" + event.data + "</p>");
    };
  });
</script>
```

This creates a new EventSource with the route for the StreamController#index method and listens to the stream for events. When it receives one, it adds the data from the event as a new paragraph in the posts element.

It's finally time to test the entire system. You'll need four terminal sessions open. In the first terminal, start the stock-service like this:

```
$ cd ~/code/stock-service
$ java -jar stock-service.war
```

In the second terminal, start the Sidekiq work like this:

```
$ cd ~/code/twitalytics
$ bin/sidekiq
```

In the third terminal, start the Puma server like this:

```
$ cd ~/code/twitalytics
$ bin/puma -C config/puma.rb
```

In the fourth terminal, use curl to make a request to the streaming service by running this command:

```
$ curl http://localhost:3000/stream/index
```

At first there won't be any response. The stream waits until a Post is created.

Open a browser to http://localhost:3000 and leave it on that page. Then point another browser window to http://localhost:3000/posts and create a new Post record with the text "Hello Apple computers." Click the Save button and return to

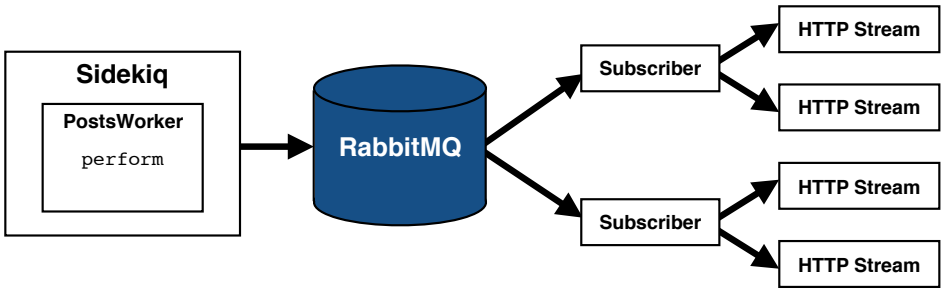
the other browser window. You'll see that the new *stockified* text has been appended to the page.

Then return to the terminal session by running the curl command. You'll see this output:

```
data: Hello <span class='stock' data-symbol='AAPL' data-day-high='92.71'>
Apple Inc.</span> computers
```

The event was sent to both streams. The curl command simply printed the event text to the console, while the browser's JavaScript updated the view.

Let's review at a high level what happened. The flow is illustrated here:



The Sidekiq worker published a message to RabbitMQ. The Puma server, which has one subscriber (aka consumer) listing to the queue, received the message from RabbitMQ. It then published an event to the event stream, which was received by the client in the browser. But it was also sent to the curl process.

Before moving on, commit your changes to Git by running these commands:

```
$ git add .
$ git commit -m "rabbitmq"
```

Now let's get this system running in production.

Deploying with RabbitMQ

CloudAMPQ is a commercial RabbitMQ hosting service you'll use to run Twitalytics in the cloud. Create a free instance of CloudAMPQ and attach it to your Heroku app by running the following command:

```
$ heroku addons:create cloudamqp:lemur
```

This sets the CLOUDAMQP_URL environment variable on your Heroku app. You've already prepared your RabbitMQ connection in rabbitmq_initializer.rb to detect this, so you don't need to make any changes to the code.

Deploy your code to Heroku as before by running git push like this:

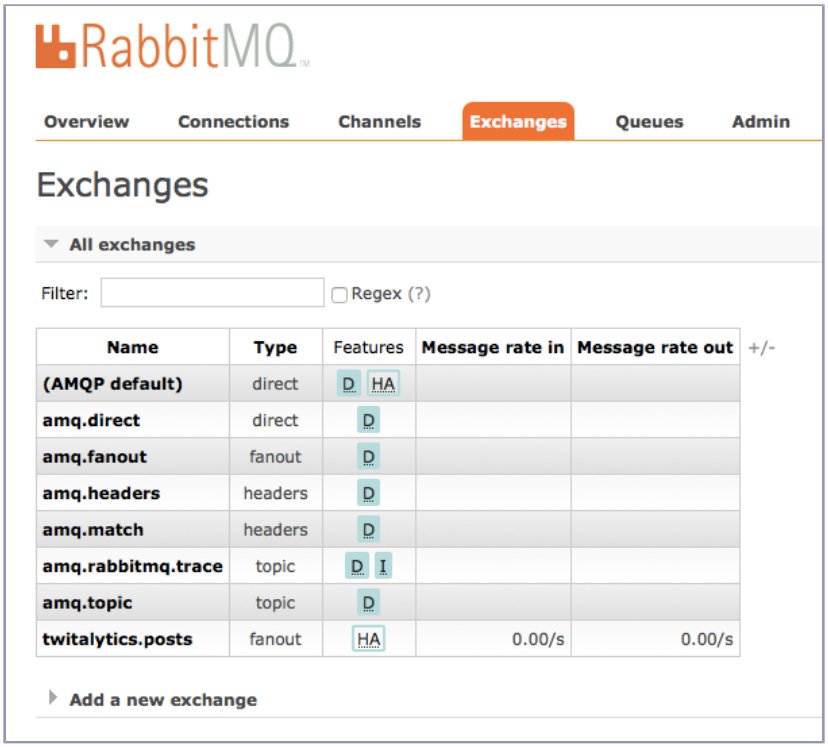
```
$ git push heroku services:master
```

When the build is finished, run heroku open to view the app. Leave the home page open as you did in development, and browse to the /posts routes in a new window. Create a few new Post records and then return to the home page to see it update.

Now run the following command to view the CloudAMQP dashboard:

```
$ heroku addons:open cloudamqp
```

Click the link to the RabbitMQ management interface and then click the link to Exchanges. You'll see something like this:



The amq-prefixed exchanges are special internal exchanges. But at the bottom you'll see your twitalytics.posts exchange. From this dashboard you can view connections, queues, and more.

If you're not using Heroku for production, you can use the RabbitMQ Docker container you ran locally with Rancher. Log in to your Rancher virtual server by running vagrant ssh. Then pull the Docker image and run a new container just as you did in your development environment. When you start your

application containers, add the appropriate environment variable to the command options like this:

```
[rancher@rancher-server ~]$ docker run \
-e MEMCACHEDCLOUD_SERVERS=192.168.99.100:11211 \
-e REDIS_URL=redis://192.168.99.100:6379 \
-e STOCK_SERVICE_URL=https://192.168.99.100:8080 \
-e RABBITMQ_URL=amqp://192.168.99.100:5672 \
-e DATABASE_URL=postgres://user:password@host:port/db \
-e PORT=3000 --publish=3000:3000 \
-dit username/twitalytics
```

Use the `RABBITMQ_URL` variable and the `entrypoint` option to start a new Sidekiq work too. Then test your app again in the browser.

When you've finished, shut down all of the Docker-based backing services in your development environment by running these commands:

```
$ docker kill memcached-server
memcached-server

$ docker kill redis-server
redis-server

$ docker kill rabbitmq-server
rabbitmq-server
```

You can always restart a container by passing the container's name to the `docker restart` command.

Wrapping Up

You've learned how to implement three of the most important backing services a JRuby app will need. You can cache data with Memcached, run out-of-process background jobs with Sidekiq, and send messages across a distributed system with RabbitMQ. But these are not the only resources you're able to consume now.

JRuby is well suited for use with many popular backing services such as Solr for full-text search and Neo4J for graph storage. Both of these services are JVM based, which means you'll be able to leverage the maturity and robustness of their native Java client libraries from Ruby code.

In the next chapter, you'll take advantage of more Java components by replacing Puma with a powerful JVM-based server. But you still won't need to write any Java code.

CHAPTER 5

Deploying JRuby in the Enterprise

Enterprise can be a dirty word to some developers. But in the context of the app we're working on, it means we simply have different constraints from those of a traditional web app being developed by a small company or startup. Often, this entails requirements for integrating with existing systems, scheduling batch jobs, publishing to message queues, and caching data from external services.

Traditional deployment with Puma has been a great solution for Twitalytics thus far. It simplified the infrastructure and deployment of your app. But as your website continues to grow, new requirements like messaging, caching, and background jobs will demand more than Puma can offer. You could tack on other systems to do this work, but for many environments the best solution is an all-in-one platform like TorqueBox.

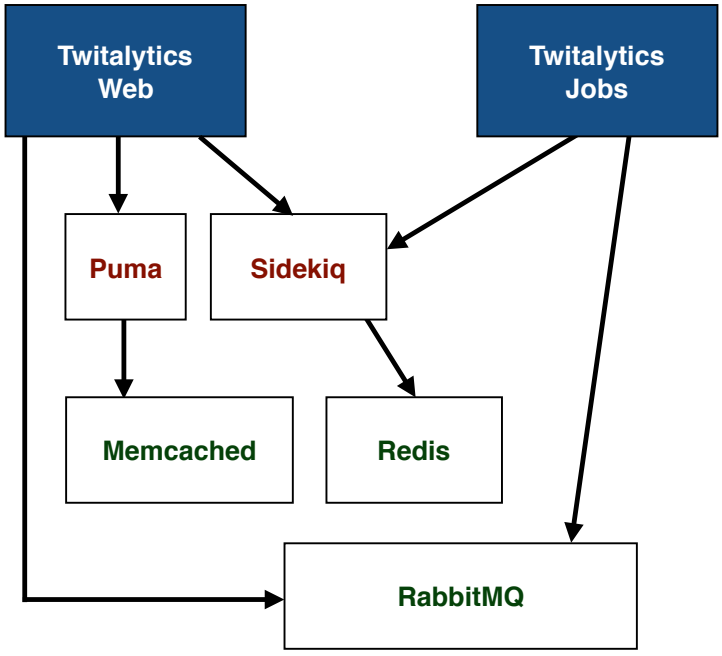
TorqueBox is the most powerful deployment environment available to any Ruby application. It's capable of boosting performance without changing a single line of code.[1] But it also has features that can improve the way an application is composed. In this chapter, you'll port Twitalytics onto TorqueBox and start taking advantage of this power.

Because of its built-in support for many advanced features, TorqueBox is often distinguished as enterprise-grade software. But it does this without many of the drawbacks programmers often associate with "enterprisey" things. Twitalytics is not enterprise software, but it has a need for many of these TorqueBox features. In fact, any application that's successful will eventually need these capabilities. Having them integrated into the platform results in a more cohesive, reliable, and manageable environment. This kind of platform is called an *application server*.

1. http://torquebox.org/news/2014/12/05/torquebox-4-0-0-alpha1-released/

What Is an Application Server?

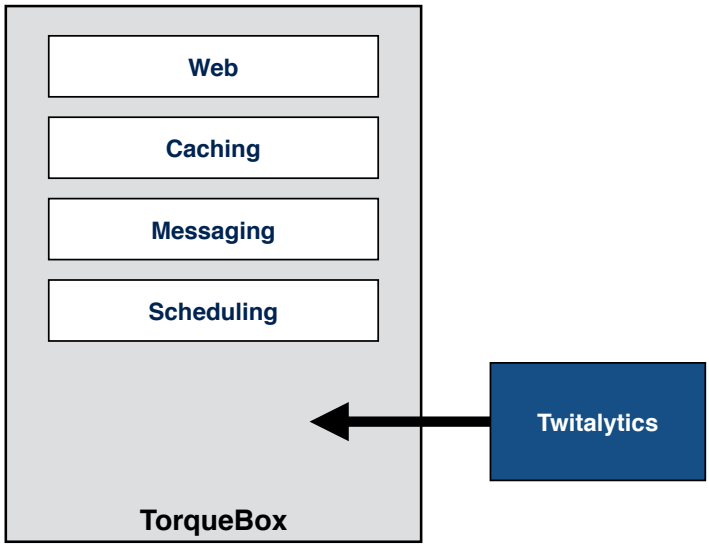
An application server is a different kind of platform from what most Rubyists are familiar with. Traditionally, a Ruby application is responsible for gathering together the libraries and tools it needs to run. This kind of architecture is illustrated in the following figure:



Ruby applications deal with many concerns like messaging and caching that are really outside the scope of their business requirements. When an MRI-based Ruby app needs to run a process asynchronously in the background, it must pull in Resque or SideKiq and integrate with it. Even worse, the developers need to manage and monitor the auxiliary processes!

The cumbersome chore of assembling infrastructure this way doesn't conform to traditional Ruby principles. Ruby is designed to be productive and fun. That's why frameworks like Rails are designed to get low-level details out of the way—so you can focus on writing business logic. Why then should you need to set up, integrate, and monitor a framework that runs background processes? You need a platform with an attitude. Such a platform—illustrated in the figure on page 75—is called an *application server*.

TorqueBox is a Ruby application server, and it's the only one of its kind. Let's port Twitalytics onto this platform as the next step in rescuing it.



Getting Started with TorqueBox

Before making any changes to Twitalytics, create and check out a torquebox branch based on the jruby branch you created in Chapter 3, *Deploying a Rails Application*, on page 29.

```
$ cd ~/code/twitalytics
$ git checkout -b torquebox jruby
```

Now you can add TorqueBox, which is distributed as both a gem and a binary file. You'll use the gem, which you can install by replacing the Puma gem in your Gemfile with the TorqueBox gem. Add this line now and remove the puma entry:

```
gem "torquebox", "4.0.0.beta2"
```

Then run Bundler to install TorqueBox and update the configuration:

```
$ bundle install --binstubs
...
Installing torquebox-core 4.0.0.beta2
Installing torquebox-caching 4.0.0.beta2
Installing torquebox-messaging 4.0.0.beta2
Installing torquebox-scheduling 4.0.0.beta2
Installing torquebox-web 4.0.0.beta2
Installing torquebox 4.0.0.beta2
...
```

You'll notice that the TorqueBox dependency pulls in a number of different gems. Each of these gems handles a different type of capability. If you need only certain features, you can pull their respective gems in individually. It also adds a torquebox executable to your path. You can run it without any arguments to get a list of available tasks:

```
$ bin/torquebox
Usage: torquebox [command] [options]

Commands:
    jar:     Create an executable jar from an application
    war:     Create a deployable war from an application
    run:     Run TorqueBox web server
...
```

The jar and war commands work much like Warbler, which you learned about in Chapter 1, *Getting Started with JRuby*, on page 1. But these are specific to TorqueBox and its capabilities. The run command launches a TorqueBox web server, which is how you'll run the app in development.

You don't need to make any changes to the code to run most web applications on TorqueBox, and Twitalytics is no exception. Start the TorqueBox server with the run task:

```
$ bin/torquebox run
14:31:47.265 INFO  XNIO version 3.3.0.Final
14:31:47.383 INFO  XNIO NIO Implementation Version 3.3.0.Final
14:31:47.428 INFO  Registered web context /
14:31:47.429 INFO  Starting TorqueBox::Web::Server 'default'
14:31:47.512 INFO  Listening for HTTP requests on localhost:8080
```

Now open a browser to http://localhost:8080 and you'll see the app running.

You've successfully integrated the TorqueBox web server with your app. If this were the only feature of TorqueBox you needed, you could reduce your Gemfile dependencies to torquebox-web. This gem is built on a lightweight, pluggable, polyglot server code-named WunderBoss. All features of TorqueBox are implemented in WunderBoss and then exposed via a Ruby API in the TorqueBox project. This lets other projects, in other languages, reuse the same functionality by creating small language-specific API wrappers. The web portion of WunderBoss uses JBoss Undertow, which is the same web server used in WildFly (the successor to JBoss Application Server).

Let's use the WunderBoss server to execute a job that runs periodically in the background.

Scheduling a Recurring Job

Jobs are components that execute on a schedule instead of in response to user action. In the case of Twitalytics, the schedule is recurring, but other jobs could be a one-time event. With TorqueBox, jobs like this run asynchronously in the background, but they still execute within the same JVM process as the rest of the application.

Twitalytics has a recurring job that removes old status updates from the database. It's located in the lib/jobs/delete_old_statuses.rb file, and it looks like this:

twitalytics/lib/jobs/delete_old_statuses.rb

```
class DeleteOldStatuses
  def run
    ids = Status.where("created_at < ?", 30.days.ago)

    if ids.size > 0
      Status.destroy(ids)
      puts "#{ids.size} statuses have been deleted!"
    else
      puts "No statuses have been deleted."
    end
  end
end

DeleteOldStatuses.new.run
```

When Twitalytics was running on MRI, this background job was scheduled by adding a crontab entry and having the cron daemon run the script with the rails runner command. But that increased the complexity of the infrastructure (since cron became another dependency) and made it less portable because Windows has no cron.

To port this job to TorqueBox, first move it to a new location under the app/jobs directory. Then use the git mv command so the repository stays in sync with your changes.

```
$ mkdir app/jobs
$ git mv lib/jobs/delete_old_statuses.rb app/jobs/
```

TorqueBox will pick up any jobs located in the app/jobs directory and run them with the full context of the application. That means it will have access to your ActiveRecord models without relying on rails runner or anything like that.

Remove the following statement, which instantiates the DeleteOldStatuses class:

twitalytics/lib/jobs/delete_old_statuses.rb

```
DeleteOldStatuses.new.run
```

Instead, you'll instantiate the class in an initializer. Create the file config/initializers/torquebox.rb and put the following code in it:

TorqueBox/twitalytics/config/initializers/torquebox.rb

```
@job = DeleteOldStatuses.new
TorqueBox::Scheduling::Scheduler.schedule(:job1, every: 1000) do
  @job.run
end
```

This initializer will run when Rails starts up. It creates a new DeleteOldStatuses object and schedules it to run in the background every 1000 milliseconds.

Now start the TorqueBox server again with the torquebox run command. After the application has booted, you'll see some output in the console, like this:

```
21:17:05,112 INFO  [stdout] ... No statuses have been deleted.
21:17:10,080 INFO  [stdout] ... No statuses have been deleted.
```

Your job is running on a schedule without any external infrastructure or supplemental processes.

Let's use some of the other TorqueBox subsystems to create Status records.

Using the Cache

TorqueBox provides a built-in caching mechanism using the Infinispan data grid. Infinispan can work in a distributed cluster to replicate storage, but its distributed features are available only when deployed to a WildFly or EAP cluster. In non-clustered mode Infinispan's cache still offers features such as eviction, expiration, persistence, and transactions that aren't available in typical caching implementations.

You can easily demonstrate the TorqueBox cache in an IRB session. Run jirb in a terminal and execute the following commands in the new session:

```
jruby-9.0.5.0 :001 > require 'torquebox-caching'
=> true
...
jruby-9.0.5.0 :002 > c = TorqueBox::Caching.cache("foo")
=> #<TorqueBox::Caching::Cache:0x7ee3d262 @cache={}, @options={}>
```

A cache is created, started, and referenced using the TorqueBox::Caching.cache method. It accepts a number of optional configuration arguments, but the only required one is a name, which uniquely identifies every cache. Now put some values in the cache and test that they are stored by running these commands:

```
jruby-9.0.5.0 :002 > c.put(:a, 42)
```

```
=> nil
jruby-9.0.5.0 :003 > c.get(:a)
=> 42
jruby-9.0.5.0 :004 > c.put(:a, 42, :ttl => 1)
=> nil
jruby-9.0.5.0 :005 > c.get(:a)
=> nil
```

This demonstrates how you can set, retrieve, and expire values. Now try some more advanced features. Create a new cache using the :max_entries options, and try to put more entries in it than are allowed:

```
jruby-9.0.5.0 :002 > e = TorqueBox::Caching.cache "baz", :max_entries => 3
=> nil
jruby-9.0.5.0 :003 > e.get(:a)
=> 42
jruby-9.0.5.0 :004 > e.put_all(:x => 42, :y => 18, :z => 1, :a => 2)
=> nil
jruby-9.0.5.0 :005 > e.cache
=> {:a=>2, :y=>18, :z=>1}
```

The cache purged the oldest values as new values were added. This prevents the cache from consuming all of your memory. Now try replacing some more values in the baz cache:

```
jruby-9.0.5.0 :006 > e.put(:y, 99)
=> 18
jruby-9.0.5.0 :006 > e.compare_and_set(:y, 99, 100)
=> true
jruby-9.0.5.0 :006 > e.get(:y)
=> 100
jruby-9.0.5.0 :006 > exit
```

All of the cache manipulation methods take the same options. You can use :ttl or :idle, which expires a value after inactivity, with each of these as well.

A great use case for this feature is template fragment caching, which allows you to cache widgets or partials in your application so they don't need to be rendered on every request. Twitalytics has a few ERB templates that could use this mechanism because they don't change very often and are the same for all users.

First, you must configure the TorqueBox cache as the default Rails cache. Open the file config/application.rb and add the following code to the Application class:

TorqueBox/twitalytics/config/application.rb

```
config.cache_store = :torque_box_store
```

Then open the file app/views/posts/index.html.erb and modify the @posts loop by adding a cache(post) call like this:

TorqueBox/twitalytics/app/views/posts/index.html.erb

```
<% @posts.each do |post| %>
  <% cache(post) do %>
    <tr>
      <td><%= post.body %></td>
      <td><%= link_to 'Show', post %></td>
      <td><%= link_to 'Edit', edit_post_path(post) %></td>
      <td><%= link_to 'Destroy', post,
              method: :delete,
              data: { confirm: 'Are you sure?' } %></td>
    </tr>
  <% end %>
<% end %>
```

Behind the scenes, a method called cache_key is invoked on the post model. This returns a string like "post/23-20130109142513", which represents the model name, ID, and updated_at timestamp. Thus, it will automatically generate a new fragment when the product is updated because the key changes. But if the key has not changed, it will use the cached value.

Test it out by starting up a server with torquebox run and browsing to http://localhost:8080/posts. The app starts up, and you'll see this in the logs:

```
18:16:36.224 INFO  Creating cache: __torquebox_store__
18:16:36.348 INFO  ISPN000128: Infinispan version: Infinispan ...
18:16:36.708 INFO  ISPN000031: MBeans were successfully registered ...
```

When you refresh the page, you won't see anything in the logs, but you might notice that it loads a bit faster.

It's also possible to use the TorqueBox cache for low-level caching, such as caching values returned from an external service or database. In this way, you'd manually create an instance of the cache by calling TorqueBox::Caching.cache just as you did in the IRB session. If you pass the name of an existing cache, a reference to it will be returned and any configuration you pass to the method will be ignored. Thus, two cache instances with the same name will be backed by the same Infinispan cache. Hence, you can ensure thread safety when retrieving the cache in two different requests.

Before moving on, commit your changes to Git by running these commands:

```
$ git add .
$ git commit -m "torquebox"
```

That completes your use of the TorqueBox features. Let's deploy this app to production.

Deploying to the Public Cloud

Because TorqueBox can be packaged into an executable JAR file, it can be deployed much like a Warbler WAR file. You can even continue using the same Heroku repository.

Make sure your Heroku app is still attached to your Git repo by running this command and confirming that there is a *heroku* remote listed:

```
$ git remote -v
```

If *heroku* is not a remote, run heroku git:remote to reattach the app you used in Chapter 3, *Deploying a Rails Application*, on page 29.

Next, you'll need to adjust the Procfile to run the TorqueBox JAR file. Open the file and put this code in it:

```
web: java -jar twitalytics.jar -p $PORT
```

Commit the file to Git with these commands:

```
$ git add Procfile
$ git commit -m "torquebox procfile"
```

Now deploy the app to Heroku by running these commands:

```
$ torquebox jar
$ heroku deploy:jar --jar twitalytics.jar
```

No other configuration is necessary. Heroku will run the app from the executable JAR file just as it did with your microservice in Chapter 1, *Getting Started with JRuby*, on page 1. You can scale it, manage, and monitor it just as you did before. Run heroku open to view the app in the browser.

Of course, public cloud deployment is not for everyone. Let's look at how to deploy this TorqueBox app to your Rancher infrastructure.

Deploying to Private Infrastructure

As mentioned earlier, TorqueBox supports three different ways of running an app. You've been using torquebox run in development, but now you'll switch to one of the other methods for production.

The first option is the torquebox war command, which generates a WAR file that's ready to be deployed into a JBoss WildFly container. If you're already running a JBoss server in your enterprise, then the TorqueBox WAR file will fit right

into your existing deployment strategy. But if you're not running JBoss, the executable JAR file is a better solution.

The torquebox jar command generates a completely self-contained executable JAR file from your application. Because you can run this JAR file with a simple java command, it's extremely portable. To begin, run the command:

```
$ torquebox jar
```

You can find the resulting JAR file in the root directory of the project, with the name twitalytics.jar. You can run it with a simple Java command to start your app, or you can add command-line options that allow it to do quite a bit more. To see a full list of features, run the JAR file with the -h option:

```
$ java -jar twitalytics.jar -h
```

Among the options, you'll see -S, which you can use to run one-off scripts and commands in the context of your application. For example, you can run Rake tasks or Ruby scripts. To try it out, run this command:

```
$ java -jar twitalytics.jar -S rake -T
```

Because the app is packaged as a JAR file, it will simplify your Docker configuration. You won't need to install JRuby or run Bundler. Open the Dockerfile and edit its contents to look like this:

```
FROM heroku/jvm

ADD ./twitalytics.jar /app/user/

ENTRYPOINT ["java", "-jar", "twitalytics.jar", "-p", "$PORT"]
```

You're still using the heroku/jvm image as a base, but now you're copying the JAR file into the image and you're defining the java command to run the app. It's similar to how your Warbler microservice was deployed in Chapter 2, *Creating a Deployment Environment*, on page 17.

Now build the image and publish it to Docker Hub by running these commands (but replace "username" with your Docker Hub username):

```
$ docker build -t username/twitalytics-tb .
$ docker push username/twitalytics-tb
```

When the push is complete, log in to your Rancher server:

```
$ cd ~/rancher
$ vagrant ssh
[rancher@rancher-server ~]$
```

Then pull the image:

```
[rancher@rancher-server ~]$ docker pull username/twitalytics-tb
```

Now use a one-off container to run the database migrations. As in Chapter 3, *Deploying a Rails Application*, on page 29, run this command with your Heroku database URL set for `DATABASE_URL` to reduce the load on the virtual machine:

```
[rancher@rancher-server ~]$ docker run \
-e DATABASE_URL=postgres://user:password@host:port/db \
--entrypoint="java -jar twitalytics.jar -S rake db:migrate" \
 -dit username/twitalytics
```

And finally, start a web server container from the image by running this command with your Heroku database URL substituted for `DATABASE_URL` again:

```
[rancher@rancher-server ~]$ docker run -e PORT=3000 \
-e DATABASE_URL=postgres://user:password@host:port/db \
--publish=3000:3000 -dit username/twitalytics
```

You can manage this container just as you did with the Puma container in Chapter 3, *Deploying a Rails Application*, on page 29. But Puma and TorqueBox are not the only Ruby web servers you can use with JRuby.

Using a Commercially Supported Server

While TorqueBox offers many features that are commonly needed in an enterprise environment, one enterprise feature it doesn't have is commercial support. Even though TorqueBox originated as a Red Hat[2] project, the company never fully productized it, which means there's no commercial support as there is for JBoss.[3]

Many organizations prefer to use technologies backed by a commercial entity to gain premium features or premium support contracts. One JRuby server offering this feature is Phusion Passenger Enterprise, which is a commercial version of the free and open source Phusion Passenger server.[4]

Passenger Enterprise includes features such as rolling restarts, error-resistant deploys, mass deployment, live IRB sessions, and advanced resource control (that is, limiting how a process consumes memory).

But the most important feature to JRuby is Passenger's support for threading. In fact, using JRuby without this feature doesn't make a lot of sense, which

2. https://www.redhat.com/
3. https://www.jboss.org/
4. https://www.phusionpassenger.com/

means Passenger Enterprise is the only form of Passenger well-suited for JRuby.

There's one more caveat: Passenger doesn't support Windows.

If you're running on Mac or Linux, the best way to get started is by running the free version of Passenger. This will ensure parity between your development environment and your production environment without having to pay for an extra license.

To begin, create a new Git branch starting from your original jruby branch by running this command:

```
$ git checkout -b passenger jruby
```

Then open the project's Gemfile and replace the puma gem with the passenger gem as shown here:

```
gem "passenger", :group => :development
```

Run Bundler to install the dependency, and run Rake to regenerate your Rails binstubs:

```
$ bundle install --binstubs
$ bin/rake rails:update:bin
```

Now launch the Passenger server by running the following command:

```
$ bin/passenger start --max-pool-size 1
```

This starts Passenger with a single application process, which is desirable for JRuby. Without the --max-pool-size option, Passenger would default to running six processes, each with its own JVM overhead. Separate processes are required on MRI to handle requests in parallel but not on JRuby.

Open a browser and point it to http://localhost:3000. You'll see the app working just as it did with the other servers. But it's not production ready yet because it's not running in threaded mode.

To run Passenger in threaded mode, you must purchase an enterprise license and install the enterprise version of the server. You can purchase a license from the Phusion website.[5] Then configure your license by setting an environment variable with this command (but replace *xxxx* with your key):

```
$ export PASSENGER_ENTERPRISE_LICENSE_DATA="xxxx"
```

Then open your Gemfile and add the following code:

5. https://www.phusionpassenger.com/download

```
group :production do
  source "https://download:#{ENV['PASSENGER_ENTERPRISE_LICENSE_DATA']}" +
         "@www.phusionpassenger.com/enterprise_gems"
  gem "passenger-enterprise-server"
end
```

This defines a new private source from which Bundler can download gems. It also adds the Passenger Enterprise Server gem as a production dependency.

Now run Bundler in production mode to install the dependency, and run Rake to re-create your Rails binstubs:

```
$ bundle install --binstubs --with=production
$ bin/rake rails:update:bin
```

When the update is finished, start Passenger with this command:

```
$ bin/passenger start --max-pool-size 1 --concurrency-model thread \
--thread-count 16
```

Passenger is running in threaded mode with sixteen threads, which is equivalent to how you ran Puma in Chapter 3, *Deploying a Rails Application*, on page 29. Put the same command in your Procfile:

```
web: passenger start --port $PORT --max-pool-size 1 --concurrency-model thread \
--thread-count 16
```

Before moving on, commit all of these changes to Git by running the following commands:

```
$ git add Gemfile Gemfile.lock Procfile
$ git commit -m "passenger"
```

Passenger is ready for production.

Deploying Passenger Enterprise

To deploy Passenger to Heroku, set your license key by running this command:

```
$ heroku config:set PASSENGER_ENTERPRISE_LICENSE_DATA="xxxx"
```

Then push your code by running this command:

```
$ git push heroku master
```

When the deployment is complete, open your app by running heroku open to confirm that it worked.

To deploy using Docker, add these lines to your Dockerfile:

```
ENV PASSENGER_ENTERPRISE_LICENSE_DATA "xxxx"

ENTRYPOINT ["bin/passenger", "start",
                             "--port",  "$PORT",
                             "--max-pool-size", "1",
                             "--concurrency-model", "thread",
                             "--thread-count",  "16"]
```

Then deploy with docker push.

Unfortunately, Passenger Enterprise does not offer all of the same features as TorqueBox. However, you can use the individual non-web TorqueBox subsystem in combination with Passenger Enterprise, which may provide the ideal solution for a risk-averse enterprise.

Wrapping Up

You made a number of changes to Twitalytics in this chapter. In addition to porting your existing components to TorqueBox, you also created some new components using the advanced features provided by your new application server. The result is a robust product that runs asynchronous jobs in the same process as the main web app.

You also learned how to deploy TorqueBox to both the public cloud and private infrastructure. You now have all the skills needed to run a complex JRuby on Rails application in production. But what happens when something goes wrong in production?

In the next chapter, you'll learn how to monitor and tune a JRuby production app to make sure you get the best uptime possible.

CHAPTER 6

Managing a JRuby Application

Deploying an application is only the first step in creating a successful production environment. Keeping it running is the real challenge.

To support any JRuby deployment, you need to understand the underlying JVM platform, how it works, and what it's doing to make your app run. In this chapter, you'll use the most common management and monitoring tools to inspect and profile a running JRuby process. They'll help you make decisions that improve both the performance and uptime of your deployment.

You'll use Java's two built-in management consoles, which provide graphical representations of resource usage over time. You'll use this same interface to control a running application by invoking management operations. Then you'll learn to use some more advanced command-line tools that provide an extreme level of detail about the platform's execution. All of these tools make it easier to tune the application for peak performance.

All of this discussion around performance and resources is moot if you never have a problem in the first place. For that reason, you'll start by adding a bug.

Creating a Memory Leak

If you don't have any bugs, you don't need any management or profiling tools. To make this chapter feel more like real life, you'll need to introduce a problem into Twitalytics so you can simulate the actual process of detecting, tracing, and resolving a real-life memory leak.

Open Twitalytics's app/controllers/post_controller.rb file and add this code to the index() method:

```
@@leaky ||= []
@@leaky << (1..1000).map{ rand(1 << 256) }
```

This creates a class variable, which is held in memory until the process is stopped. Then it adds one thousand random Bignum instances to the array. The Bignum class isn't used very often, so it will be easy to identify with the tools you'll learn about. Now when you want to simulate a problem, you only need to make a request to the PostController.

In this chapter, you'll use this leak to illustrate the capabilities of each tool you learn about. Let's begin with the most commonly used graphical JVM tool.

Inspecting the Runtime with VisualVM

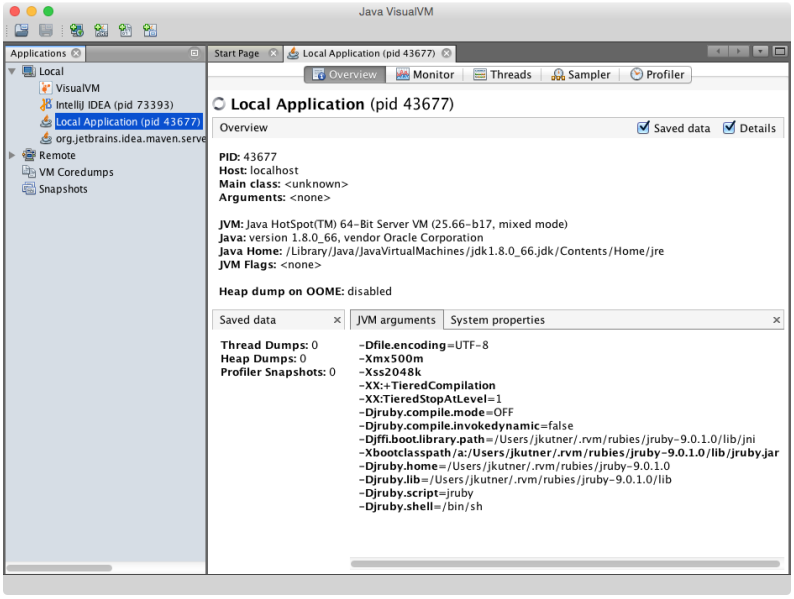
VisualVM is a graphical user interface (GUI) for monitoring a JVM, profiling a running application, and analyzing heap dumps. It comes packaged with the JDK, so there's no need to install it. As long as the java command is on your PATH, you're ready to go. Run this command to start VisualVM:

```
$ jvisualvm
```

This opens the GUI, where you'll see a list of applications in the left sidebar. There are groups for local and remote JVM applications. At a minimum, you'll see the VisualVM process listed under Local. If you have some other Java processes running, you'll also see those.

Open a new terminal window and move into the Twitalytics directory. Now start your JRuby server with Puma so VisualVM can connect to it:

```
$ bin/puma -C config/puma.rb
```

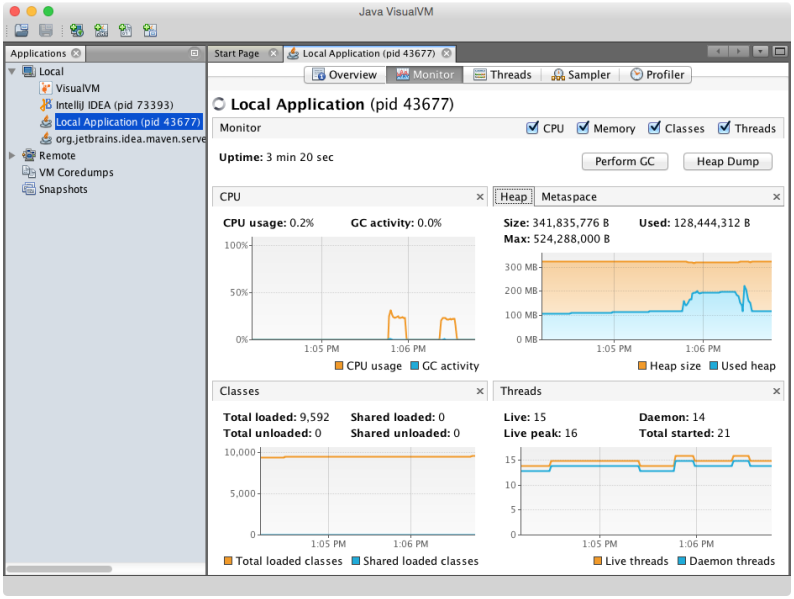


All the steps in this chapter will work for either the Puma or TorqueBox server. But the examples will use the Puma commands exclusively.

Allow the process to boot, and then point a browser at the app to ensure it's working. If all looks well, return to the VisualVM interface and you'll see a new Local Application listed in the sidebar. Double-click it and a new tab will open on the right. You'll see something like the preceding figure on page 88.

The Overview interface shows some high-level details about the process, including the command that's running, the system properties, and the location of the JVM that's running it.

Now click the Monitor tab. This view provides graphs for CPU activity, memory usage, number of classes, and number of threads for the JVM process. Because the process just recently started, there probably won't be much to see yet. Let the process run for a while, and refresh your browser a few times to force the server to process some requests; eventually you'll see something that looks like this:

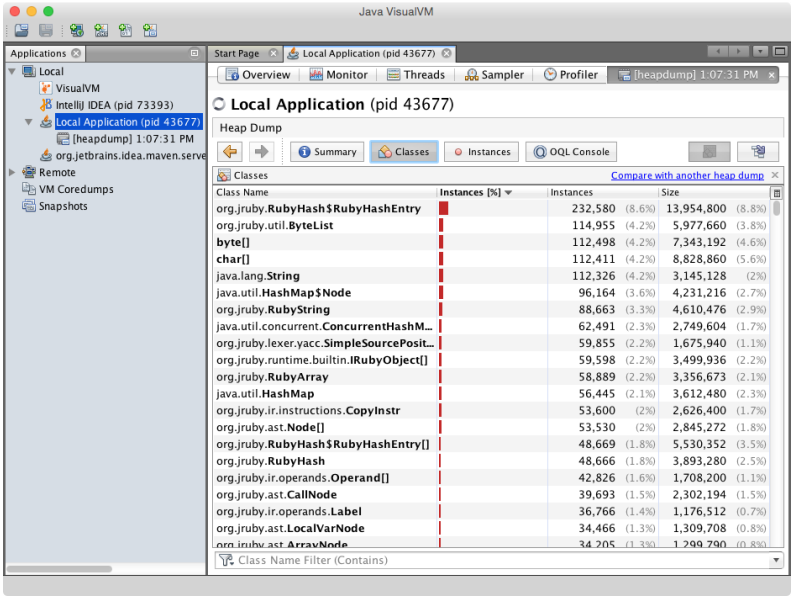


In this view, the term *Heap* refers to memory. We'll discuss this in more detail in Chapter 7, *Tuning a JRuby Application*, on page 109.

The Monitor view also has two buttons: Perform GC and Heap Dump. The Perform GC button will force the JVM to run its garbage collector. Go ahead and click it now. Next, you'll see the blue line that represents heap memory plunge as the system reclaims memory from unused objects. Heap memory

is a type of system memory that holds objects created by your app. We'll discuss this in detail in Chapter 7, *Tuning a JRuby Application*, on page 109.

The other button, Heap Dump, generates a snapshot of heap memory at a singular moment. Click it now, and you'll see a new tab appear like the one in the following figure:

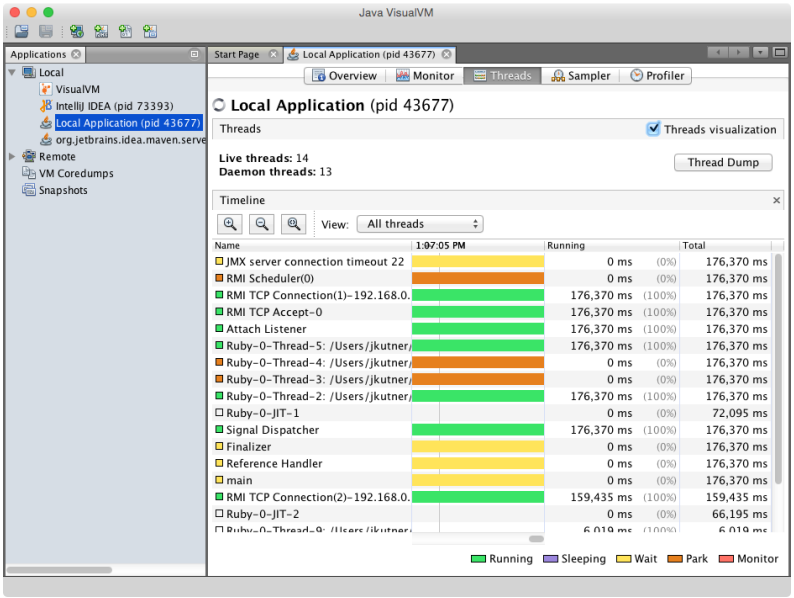


This snapshot contains data on every object your application has in memory. From this view, you can analyze classes and instances. You can also compare one dump to another to expose what has changed over time. This will become incredibly useful as we begin to tune the performance of this application and try to detect memory leaks.

Now click the Threads tab. This view illustrates all of the threads in the JVM and their current state. It looks something like the figure on page 91.

The Thread Dump button on this view will generate a thread dump. Click it now. Just like the heap dump, this creates a snapshot that you can analyze and compare to other snapshots to discover trends. In practice, you'll take many thread and heap dumps over an extended period of time and compare them all. This is a common task when diagnosing problems in a JVM-based application.

The final two tabs in the GUI are Sampler and Profiler, which do two different kinds of profiling. They allow you to watch changes to the CPU and memory usage at a fine-grained level in near real time.



Sampling Profiler

Click the Sampler tab and explore the options for CPU and Memory. You'll see how the different threads, methods, and objects affect these resources.

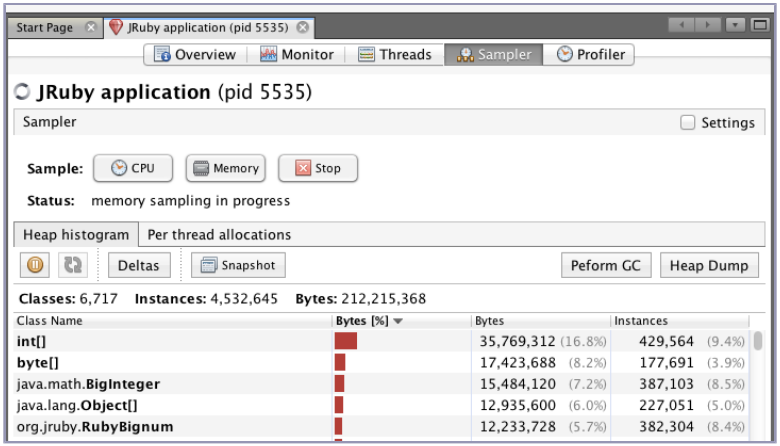
Sampling is the most basic mode of profiling and carries the least amount of overhead. It's an important mode because a common pitfall of profiling is altering the performance characteristics of an app by introducing code that measures it. Sampling reduces this risk by limiting the impact of the profiler.

To test the sampler, you'll need to exercise the synthetic memory leak you created earlier. Keep Twitalytics running, and hit the /posts route in an infinite loop by running this command:

```
$ ruby -r "net/http" \
    -e "while true; Net::HTTP.get(URI('http://localhost:3000/posts')); end"
```

After a few moments, you'll see RubyBignum and java.lang.BigInteger (which is the underlying implementation of RubyBignum in JRuby) rise toward the top of the list. You'll also see int[] and byte[], which hold internal representations of those big numbers. It will look like the figure on page 92.

Unfortunately, sampling is prone to many possible errors. The sampler takes measurements when a timer periodically fires. The profiler then looks at each thread to see what method is executing and tallies up only what it sees. If a thread is alternating evenly between executing two methods and the timer



fires only during the execution of one method, then you'll never see the second method in the output. The sampler is also easily confused by inlined methods (methods that have been compiled into other methods by the JVM's just-in-time compiler to improve performance).

You can mitigate potential sampling errors by running the profiler for a longer period of time. Or you can try a more invasive approach to profiling.

Instrumenting Profiler

Click the Profiler tab and explore the interface, which is similar to the Sampler tab. You'll see how the different resources are affected in near real time. But this time, keep an eye on the terminal session in which the JRuby process is running. You'll see something like this:

```
Profiler Agent: JNI OnLoad Initializing...
Profiler Agent: JNI OnLoad Initialized successfully
Profiler Agent: Waiting for connection on port 5140 (Protocol version: 15)
Profiler Agent: Established connection with the tool
Profiler Agent: Local accelerated session
...
Profiler Agent: Redefining 100 classes at idx 0, out of total 500
Profiler Agent: Redefining 100 classes at idx 100, out of total 500
Profiler Agent: Redefining 100 classes at idx 200, out of total 500
Profiler Agent: Redefining 100 classes at idx 300, out of total 500
Profiler Agent: Redefining 100 classes at idx 400, out of total 500
```

This output comes from the Java profiling agent, which runs alongside your application and instruments its code as it's running. In this way, it can capture exact measurements.

Is the instrumented profiler better than the sampling profiler? It's difficult to say. Instrumentation is more likely to contaminate the results, but sampling is more likely to miss something. There's no way to be sure in any given situation which one is more accurate. The best practice then is to collect as much information as possible, knowing that some of it may be flawed, and make the best decision you can from that data. Later in the chapter you'll learn about a profiling technique that's provided specifically by JRuby itself and may provide more Ruby-centric information.

The sampler and profiler are two extremely powerful tools that are unique to VisualVM. But many of the other operations are made possible by the Java Management Extensions (JMX) protocol, which can be consumed by a number of other tools. If you access these extensions directly, you can gain even more insight into the running process.

Inspecting the Runtime with JMX

Java Management Extensions is a set of tools that support the management and monitoring of system objects, devices, networking, and applications. All of these tools are exposed through a service interface that's controlled by scripts and even other applications. But the JDK comes packaged with a general-purpose console that provides a graphical interface for quickly inspecting a runtime through these extensions. This console is similar to VisualVM but provides a more fine-grained interface.

Before starting the JMX console, boot your application again. When doing so, provide the `--manage` option, which turns on JRuby's own management extensions. Using Puma, the command is this:

```
$ ruby --manage -S bin/puma
```

When running from a WAR created by Warbler, there's no option to pass because you're invoking Java directly. Instead, you need to add two options to the `java` command, which are the same options the `--manage` flag adds behind the scenes.

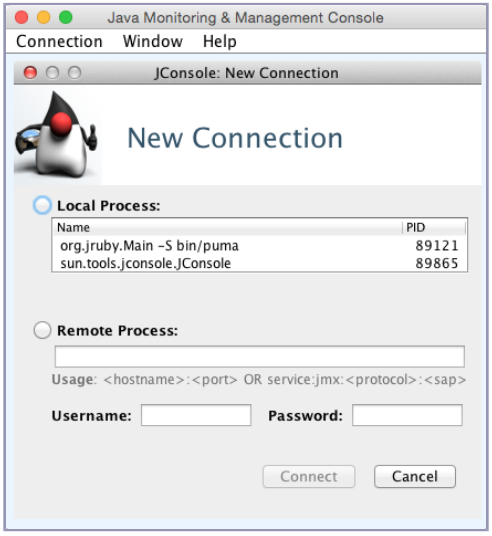
```
$ java -Dcom.sun.management.jmxremote -Djruby.management.enabled=true \
       -jar twitalytics.war
```

With TorqueBox, this option isn't necessary because the JMX services are exposed by default.

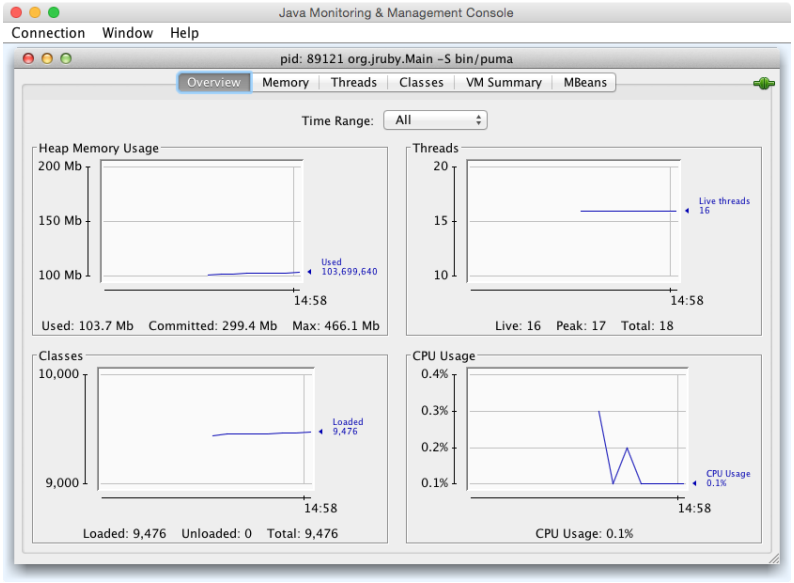
No matter which framework you choose to run Twitalytics with, open the management console by running the following command:

```
$ jconsole
```

This command is provided by the JDK you installed in *Preface*, on page xi. When the JConsole starts up, it will give you the choice of connecting to a local JVM or a remote JVM. In the list of local JVMs, you'll see the process you started earlier, as shown in the following figure:

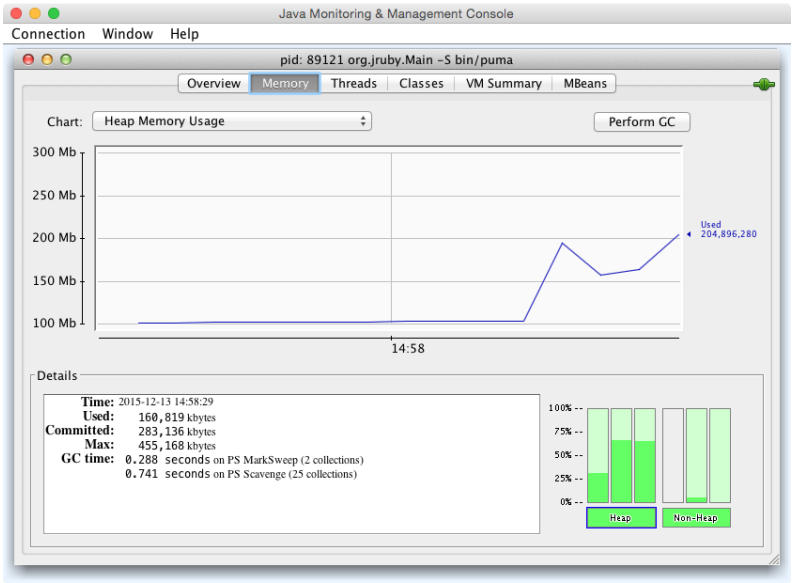


Select this process and JConsole will connect to the JVM. Select Insecure Connection when prompted (it's a local process, so this isn't much of a security concern). After it connects, you'll see an overview interface like the one pictured in the following figure:



The Overview screen provides near-real-time graphical views of heap usage, CPU usage, active threads, and the number of classes loaded in the runtime, much like the one in VisualVM. This console provides a vast amount of information—so much so that entire books are written about it. But it's not necessary to be an expert from the start.

Take a look at the Memory tab, which is shown in the following figure.

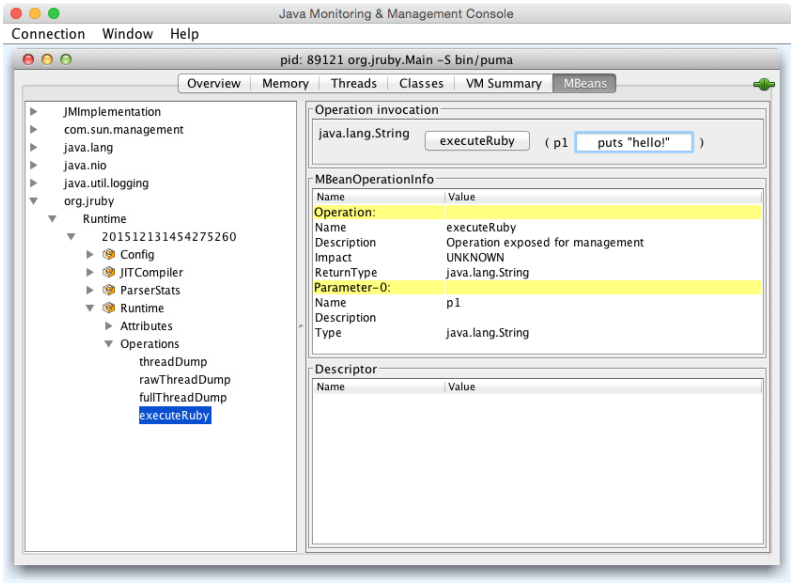


While the Overview screen showed only heap usage, the Memory screen provides insight into the different categories of JVM memory including Metaspace, Code Cache, and more. You'll learn more about these in Chapter 7, *Tuning a JRuby Application*, on page 109. The view also has a Perform GC button just like the one in VisualVM.

Now click the MBeans tab. An MBean, or Managed Bean, is an object that represents a resource in the JVM that can be managed. Each MBean will have attributes that tell you about the resource and the operations you can invoke on it.

The directory-like structure in the left panel of the screen shows the registered MBeans in the system. Each of these represents a different component of the JVM and the JRuby runtime. Browse to the org.jruby/Runtime/<guid> bean, which represents the JRuby runtime. Even though the MBean's name gives this information away, you can make certain of this by selecting the Config/Attributes node under it. Here you'll see the various properties of the JRuby instance such as the working directory, debug flag, and version string.

Next, select the Runtime/Operations node. You'll see buttons representing the management operations available for this MBean. One of these, the executeRuby() operation, allows you to execute arbitrary Ruby code in the context of the JRuby instance. Select the operation, and fill in the form with the string puts "hello!", as shown in the following figure:



Then click the executeRuby() button, which fittingly executes the given Ruby code. You'll see a dialog appear with nothing in it because the operation has no return value. But look in the terminal session that's running your JRuby server and you'll see this:

```
hello!
```

JMX provides an excellent mechanism for managing your application, but clicking buttons in a GUI may not be your preferred tool. Fortunately, there are other options.

Invoking MBeans Programmatically

JConsole is just one way to use the JMX services that are exposed by the JVM. You can also build your own clients to consume JMX services. This is a handy way to write tools you can use to manage your applications.

To create a JMX client, you'll need some libraries that can speak to the JMX interfaces. Fortunately, the jmx4r gem provides a Ruby wrapper for this Java-based protocol.

Install the jmx4r gem with the following command:

```
$ gem install jmx4r
```

Before using this gem, you'll need to make sure a JRuby server is running.

```
$ ruby --manage -S bin/puma
```

Now you can connect to the JMX services in the JRuby process from the shell. Begin by starting an IRB session and requiring the jmx4r gem:

```
$ irb
jruby-9.0.1.0 :001 > require 'rubygems'
=> true
jruby-9.0.1.0 :002 > require 'jmx4r'
=> true
```

Next, create a connection to the JRuby server process with the following command:

```
jruby-9.0.1.0 :004 > JMX::MBean.establish_connection \
                       :command => "org.jruby.Main -S bin/puma"
 => #<JMX::MBeanServerConnectionProxy:0x7e64cfe0 @connection=#<#<Class:0x5...
```

The :command argument matches the connection string you saw in the initial JConsole dialog. But the gem also supports connecting to remote JVM processes with the :host, :port, :username, and :password arguments.

Now that you've created a connection, you can get a handle to one of the MBeans. Use the Memory manager:

```
jruby-9.0.1.0 :005 > memory = JMX::MBean.find_by_name "java.lang:type=Memory"
 => #<JMX::MBean:0x1916a3de>
```

Now invoke an operation on the MBean using the gc() method, which performs the same action as when you click the Perform GC button in JConsole. Execute this statement:

```
jruby-9.0.1.0 :007 > memory.gc
 => nil
```

If you have JConsole open, you'll see another dip in the graph of heap memory usage. Running the garbage collector is a nice example, but it's not something you'll usually need in the JVM. It's possible you might want to execute some arbitrary Ruby code, as you did with JConsole. But a more useful example would invoke your own custom MBean. Let's create one and invoke it from a Rake task.

Creating a Management Bean

An MBean's purpose is to help manage your application. Many Ruby applications provide this same kind of interface with RESTful services, but those tend to get in the way of the real application. MBeans provide a better interface because they're separated from the rest of the application, which means they have their own security, port, and graphical interface. As a result, there's less of a chance that an ordinary user will accidentally (or intentionally) gain access to the management services.

Let's create an MBean that you can use to manage the logging level of your Rails application. You'll start by adding the jmx4r gem to your Gemfile and running Bundler.

Management/twitalytics/Gemfile

```
gem 'jmx4r'
```

Next, create a lib/logging_bean.rb file and add the following code to it:

Management/twitalytics/lib/logging_bean.rb

```
class LoggingBean < JMX::DynamicMBean
  operation "Set the Rails log level"
  parameter :int, "level", "the new log level"
  returns :string
  def set_log_level(level)
    Rails.logger.level = level
    "Set log level to #{Rails.logger.level}"
  end
end
```

This class inherits from the JMX::DynamicMBean class, which hides all of the Java code that goes into creating an MBean. Then it defines a set_log_level(level) operation and declares its argument type and return value type. Unlike Ruby, Java is a strongly typed language, so it expects these things. In the body of the set_log_level(level) operation, you're setting Rails.logger.level to the value that was passed in as an argument.

Now register this MBean with the platform's MBean server, which is part of the JVM runtime, by creating an mbeans.rb initializer file in the config/initializers directory and putting the following code in it:

Management/twitalytics/config/initializers/mbeans.rb

```
java_import "javax.management.ObjectName"
java_import "java.lang.management.ManagementFactory"
```

This imports two Java classes that will give you access to the MBean server.

Next, you'll need some code that instantiates your bean and registers it with the server. Put these lines after the java_import statements in the same file:

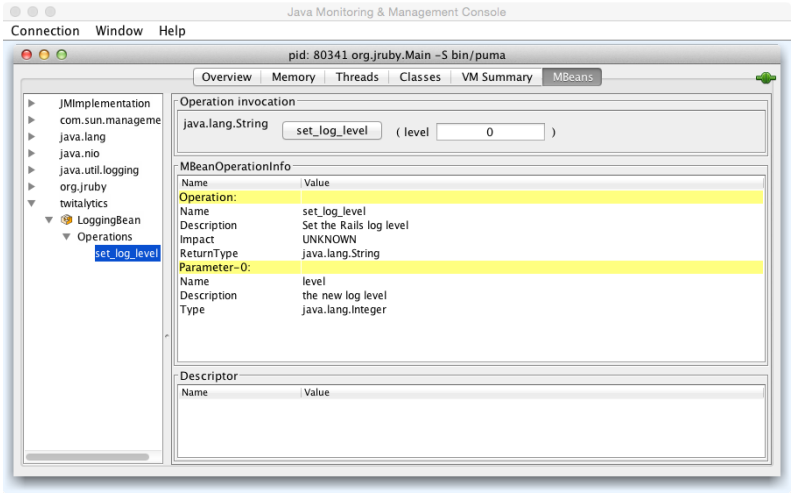
Management/twitalytics/config/initializers/mbeans.rb

```
require "logging_bean"
mbean = LoggingBean.new
object_name = ObjectName.new("twitalytics:name=LoggingBean")

mbean_server = ManagementFactory.platform_mbean_server
mbean_server.register_mbean mbean, object_name
```

Run the JRuby server again and look for your MBean in the JConsole (reboot the server if it's already running).

```
$ ruby --manage -S bin/puma
```

Run jconsole and navigate to the MBeans screen. You'll see a twitalytics MBean with the operation you defined. When you enter a value in the text field of the set_log_level(level) method and click the button, you'll see the result pictured in the following figure:



Now you can write a Rake task that invokes this MBean service. Create a lib/tasks/mbean.rake file, and add the following code to it:

Management/twitalytics/lib/tasks/mbean.rake

```
namespace :log do
  task :debug do
    JMX::MBean.establish_connection :command => "org.jruby.Main -S bin/puma"
    logging = JMX::MBean.find_by_name "twitalytics:name=LoggingBean"
    puts logging.set_log_level(0)
  end
end
```

The steps in this task are similar to the steps you executed in your IRB session earlier in the chapter. But instead of getting a handle to the Memory MBean, you're retrieving your own custom MBean.

Try it out. If the server is still running, you can execute this command:

```
$ bin/rake log:debug
Set log level to 0
```

That should feel a little more natural to a Rubyist than using the GUI. But the graphical choice is always there, and it can be useful when someone other than you is managing your application (such as an operations team).

Let's move on and use some tools that will give you even more insight into a running application.

Using the JRuby Profiler

In addition to the profiling tools you saw in VisualVM, JRuby has its own built-in profiler. You can enable this feature by adding the `--profile` option when starting a JRuby process. To test it out, run the following command:

```
$ ruby --profile -S bin/puma
```

Make a few page requests, and then kill the process by pressing Ctrl-C. This will dump some statistics to the console that look like this:

```
Total time: 19.08

total        self    children       calls  method
------------------------------------------------------------------
17.97        0.05       17.91          50  Kernel.load
16.84        0.00       16.84           1  Puma::CLI#run
16.84        0.00       16.84           1  Puma::Single#run
12.88       12.88        0.00           1  Thread#join
 5.62        0.03        5.59        6215  Kernel.require
 4.27        0.01        4.26       10565  Class#new
 4.02        0.03        3.99          74  Kernel.eval
 3.95        0.00        3.95           1  Puma::Runner#load_and_bind
 3.92        0.00        3.92          80  BasicObject#instance_eval
 3.92        0.00        3.92           1  Puma::Configuration#app
 3.92        0.00        3.92           1  Puma::Configuration#load...
 3.92        0.00        3.92           1  Puma::Rack::Builder.pars...
 3.92        0.00        3.92           1  Puma::Rack::Builder.new_...
 3.92        0.00        3.92           1  Puma::Rack::Builder#init...
 3.85        0.03        3.83        6694  Array#each
...
```

You can tell from this dump that the main program loop is controlled by the Puma::CLI#start() method. You can also see that the most expensive operation in the app right now is the Class#new() method. This kind of information can be useful when things get stuck.

But using the profiler with a complete application can leave you swimming in data. In some cases, this may be what you want—especially if you're using a tool or script to analyze the results. But it's usually better to isolate the code you're trying to profile. For example, you can profile just the standard_dev(values) method from the Twitalytics AnalyticsUtil module because it processes a big array like this:

```
$ ruby -r ./lib/analytics_util.rb --profile \
-e "AnalyticsUtil.standard_dev(Array.new(10**4) {1})"
Profiling enabled; ^C shutdown will now dump profile info

main profile results:
Total time: 0.62

total        self    children       calls  method
-----------------------------------------------------------------
0.54        0.05        0.50          27  Kernel.load
0.35        0.02        0.33          63  Kernel.require
0.08        0.02        0.07         111  Array#each
0.07        0.00        0.07           2  IO.open
0.07        0.03        0.04           2  IO#each_line
0.06        0.00        0.06           1  Gem::Specification.load_d...
0.06        0.00        0.06           1  Gem::Specification.each_s...
0.06        0.00        0.06           1  Gem::Specification.each_g...
0.05        0.00        0.05         948  Class#new
...
```

That still generates a lot of information, but it's a little more tractable. Let's break things down even more.

The built-in profiler also includes a graph mode, which separates the execution times of callers and callees. You can demonstrate this by running the previous example with the --profile.graph option:

```
$ ruby -r ./lib/analytics_util.rb --profile.graph \
-e "AnalyticsUtil.standard_dev(Array.new(10**4) {1})"
Profiling enabled; ^C shutdown will now dump profile info

Total time: 0.61
```

```
%total  %self   total     self   children            calls  name
------------------------------------------------------------------------
100%      0%    0.61     0.00      0.61            0  (top)
                0.54     0.05      0.49       19/27  Kernel.load
                0.03     0.00      0.03         1/1  AnalyticsUtil....
                0.01     0.00      0.01         1/1  JRuby.runtime
                0.01     0.00      0.01       1/948  Class#new
                0.00     0.00      0.00         2/2  Kernel.require
                0.00     0.00      0.00         1/2  Java::OrgJruby...
                0.00     0.00      0.00        1/63  Kernel.require
                0.00     0.00      0.00         1/1  Kernel.trap
                0.00     0.00      0.00         1/1  Java::OrgJruby...
                0.00     0.00      0.00         1/1  IO#puts
                0.00     0.00      0.00        4/11  JavaProxy.inhe...
                0.00     0.00      0.00      18/335  Array#eql?
                0.00     0.00      0.00     186/311  Class#inherited
...
```

Graph mode gives a better picture of why certain methods are taking up time and which callers are contributing the most to that time. But it displays a list of every method that gets called, which means you're still pretty inundated with information. Let's keep breaking things down.

The JRuby profiler also includes an API you can use to instrument code and narrow the part of your application that gets profiled. Try this in the AnalyticsUtil class. Open the lib/analytics_util.rb file and modify the standard_dev() method so it looks like this:

```
def self.standard_dev(vals)
  profile_data = JRuby::Profiler.profile do
    if vals.empty?
      0
    else
      avg = (vals.inject(0) {|sum, s| sum + s}) / vals.size
      diffs = vals.map {|s| (s-avg)**2}
      Math.sqrt((diffs.inject(0) {|sum, s| sum + s}) / vals.size)
    end
  end
  profile_printer = JRuby::Profiler::GraphProfilePrinter.new(profile_data)
  profile_printer.printProfile(STDOUT)
end
```

This wraps the body of the standard_dev() method in a block that gets passed to the JRuby::Profiler.profile() method. This returns some profiler data, which is passed to the JRuby::Profiler::GraphProfilePrinter class so it can be printed in graph mode.

Now run the example with the --profile.api option, which will turn on the API mode. You'll also need to require the jruby/profiler package, which contains the classes you added to the AnalyticsUtil. The complete command will look like this:

```
$  ruby -r ./lib/analytics_util.rb -r jruby/profiler --profile.api \
-e "AnalyticsUtil.standard_dev(Array.new(10**4) {1})"
Profiling enabled; ^C shutdown will now dump profile info
Total time: 0.04

%total   %self   total     self  children          calls  name
------------------------------------------------------------------------
100%      0%     0.04     0.00      0.04              1  (top)
                 0.03     0.00      0.03            2/2  Enumerable...
                 0.01     0.01      0.00            1/1  Array#map
                 0.00     0.00      0.00            1/1  Math.sqrt
                 0.00     0.00      0.00            2/2  Fixnum#/
                 0.00     0.00      0.00            2/4  Array#length
                 0.00     0.00      0.00            1/1  Array#empty?
------------------------------------------------------------------------
                 0.03     0.00      0.03            2/2  (top)
77%       0%     0.03     0.00      0.03              2  Enumerable...
                 0.03     0.03      0.00            2/2  Array#each
------------------------------------------------------------------------
                 0.03     0.03      0.00            2/2  Enumerable...
76%      76%     0.03     0.03      0.00              2  Array#each
------------------------------------------------------------------------
                 0.01     0.01      0.00            1/1  (top)
22%      18%     0.01     0.01      0.00              1  Array#map
                 0.00     0.00      0.00    10000/10000  Fixnum#**
------------------------------------------------------------------------
                 0.00     0.00      0.00    10000/10000  Array#map
 3%       3%     0.00     0.00      0.00          10000  Fixnum#**
```

Now you've isolated the profiling to just the relevant code, and you're getting a more concise picture of the performance metrics for the standard deviation method.

Profiling is one way of detecting problems. But not all problems are CPU bound. When you have a memory leak, you'll need a different kind of tool.

Analyzing a Heap Dump

Using VisualVM, you learned how to take heap dumps and make simple comparisons between them. That was the tip of the iceberg when it comes to JVM tooling. More advanced heap dumps can be captured with the jmap tool, and you can do an incredibly deep analysis with a tool called Eclipse Memory Analyzer (MAT).[1]

1. http://www.eclipse.org/mat/

The jmap tool comes with the JDK. If you have the java command on your PATH, then you most likely have the jmap command too. jmap is preferable in environments where VisualVM cannot be used either because of networking constraints or because you're trying to capture metrics for a process not started with the JMX services exposed.

Before giving jmap a try, start a JRuby server:

```
$ bin/puma -C config/puma.rb
```

Now open a second terminal session and find the process ID for the JRuby server by running the jps command (another tool installed with the JDK):

```
$ jps -l
48816 org/jruby/Main
48971 sun.tools.jps.Jps
```

This shows the process ID and main class of all Java processes running on your machine. At a minimum, you'll see one JRuby process, identified as org/jruby/Main, and one process for jps itself, identified as sun.tools.jps.Jps. Note the process ID (the number to the right of the class name) for the JRuby process, and use it with the jmap command like this:

```
$ jmap 48816
Attaching to process ID 48816, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 25.66-b17
```

This output means that jmap was able to connect to the process. Now capture a heap dump by running this command:

```
$ jmap -histo 48816
```

This prints a histogram of the heap to the console. It includes the number of objects, memory size in bytes, and fully qualified class names for each Java class. VM internal class names are printed with a "*" prefix. You can also use the -histo:live option to limit the output to only objects that are currently in use.

A histogram is useful for quick analysis, but it's not any better than what you saw in VisualVM. To make a deeper analysis of a heap dump, you'll need to generate a file in the Heap Profiling (HPROF) binary format by running the following command:

```
$ jmap -dump:format=b,file=heap.hprof 48816
```

This creates a file, heap.hprof, in the same directory where you ran the command. Now you can open this file in a tool that can read and analyze it. One such

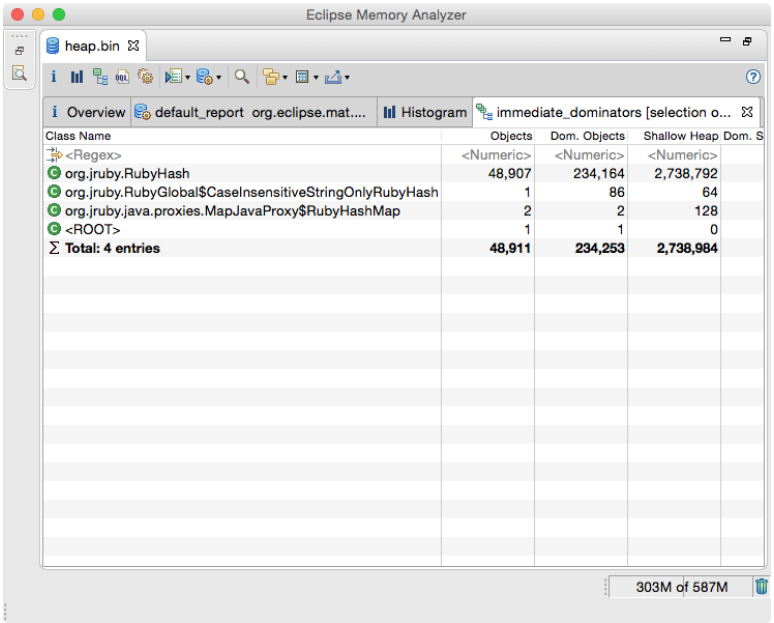
tool is jhat, which launches a web server you can use to browse the objects. But again, it's not any better than VisualVM. A better tool is Eclipse MAT.

Set up Eclipse MAT now by downloading the installer for your platform from the official website.[2] Once the app is installed, run it. From the File menu, select the Open a Heap Dump... option. Browse to the heap.hprof file you created and select it. It will take a moment to process the file, and when it's done you'll see a pie chart representing all the classes found in the heap.

Eclipse MAT does a great deal more than display the classes and objects. It can analyze the results and even pinpoint the source of a memory leak. From the menu, select the button with an arrow. From the next set of choices select Leak Suspects. This will open a new tab with a list of possible candidates for the source of a memory leak, as shown in the following figure:

2. http://www.eclipse.org/mat/downloads.php

Click the Details link, and then right-click one of the objects listed and select Immediate Dominators. This opens a new tab containing a list of objects that are holding references to the class you selected. For example, if you selected RubyHashEntry, you'd likely see that the objects holding a reference to this are of type org.jruby.RubyHash, as shown in the following figure:



In this way, Eclipse MAT can help you not only identify leaking objects but also identify what objects are holding the references that are preventing the leaking object from being cleaned up. Diagnosing a memory leak is never easy, but with Eclipse MAT you can make intelligent decisions as you investigate the problem.

Try exercising your synthetic memory leak again. Give the code some time to run and then take a heap dump. See if you can locate the leak using Eclipse MAT. When finished, remove the leaking code in app/controllers/post_controller.rb.

The jmap tool is just one of the many tools provided with the JDK. Other tools, such as jstack, can attach to a running process and create a thread dump. jstack can even identify a deadlock. These other tools are not notably better than VisualVM, but they're essential when you need to remotely attach to a running process in a headless production environment.

VisualVM, JMX, JRuby profiler, and Eclipse MAT are all tools you can use to solve and prevent problems. But you can always defer to command-line tools when you need to write scripts or Rake tasks. As with your deployment, using

JRuby doesn't mean you have to dramatically change the way you do things. But if you're willing to embrace some new features, you'll gain a lot of power.

Wrapping Up

Keeping an application running is difficult. But the tools and techniques you used in this chapter will help diagnose and resolve problems when your application starts misbehaving. In this chapter, you learned about Java Management Extensions, which helped you inspect and control your runtime. You may choose not to use this tool, but it still helped you gain a better understanding of the JVM's innards.

You also used some profiling tools to get a snapshot of Twitalytics's performance characteristics. Every application has its slow spots, but with a basic understanding of these tools, you'll be able to track down those pain points without much trouble.

Deploying Twitalytics on JRuby simplified its infrastructure, which allows these tools to give you a better picture of the health of the system. You no longer have to monitor dozens of processes that have their own memory footprints and CPU utilization. Instead, you can use the robust tools and services provided by the JVM to capture the entire picture of your application's performance.

In the next chapter, you'll learn how to use the information you've collected with these tools to tune your app, the JRuby runtime, and the JVM for peak performance.

CHAPTER 7

Tuning a JRuby Application

Tuning for peak performance is a bit like conducting a science experiment. You develop a hypothesis, test your hypothesis, and then change variables to test their effect on the system. The trick is knowing what variables to change and when to change them.

In the coming pages, you'll learn about some variables you can configure and knobs you can turn in the JVM to improve performance. You'll learn what heap memory is and how you can adjust it. You'll also learn about other types of memory like metaspace and direct memory. In every case, you'll learn how to detect early warning signs of problems and make corrections before they cause trouble.

After memory, you'll learn about garbage collection (GC), which relates to memory but also to CPU performance. Then you'll execute benchmarks against the GC in Twitalytics. Through this process, you'll learn about the different options you can set in JRuby when starting a process. These options control every aspect of the runtime, from the JVM up to the Ruby code itself.

Let's begin with an option that is essential to how your code performs on the JVM.

Setting the Heap Size

All of the JVM and JRuby options you'll learn about in this chapter can be used with each of the frameworks discussed in this book. Some need to be defined in a different way depending on the platform, but those cases will be specifically called out. Despite the differences in how they're defined, the underlying effects of these options will remain the same across frameworks.

When a JVM starts up, it reserves a chunk of system memory called the *heap*. Each time a new object is created in a JRuby program, the platform allocates

a piece of heap memory for it, which is reserved until the GC decides to reclaim the object. At that time, the associated piece of memory is returned to the heap. This process is called *dynamic memory allocation*, and MRI uses a similar strategy. But the JVM gives you more control over how memory is managed.

When you start a JVM, you can configure several parameters that determine how heap memory grows. You can set its initial size, maximum size, and what algorithm the runtime uses for GC. The values you choose for these options greatly affect the performance of an application.

The most commonly used JVM options control the heap size. They are described here:

-J-Xms This sets the initial size of the heap. The JVM will reserve this amount of memory at startup. The flag should be followed by a positive integer value and then by either `k`, `m`, or `g` (for KB, MB, and GB, respectively). Here's an example: `-J-Xms64m`.

-J-Xmx This sets the maximum size of the heap. It should be followed by a positive integer value and then by either `k`, `m`, or `g` (for KB, MB, or GB, respectively). Here's an example: `-J-Xmx512m`.

The default maximum heap size for JRuby is 500 megabytes, or `500m`, but most web servers typically run with at least a 1-gigabyte cap. If the maximum is set too low, it can cause the JVM to lock up or even crash. You can demonstrate this by starting up Twitalytics with a ridiculously low maximum of 16 megabytes. Using Puma, the command would look like this:

```
$ ruby -J-Xmx16m -S bin/puma
...
! Unable to load application: LoadError: load error: sass/tree/css_import_node
  -- java.lang.OutOfMemoryError: GC overhead limit exceeded
LoadError: load error: sass/tree/css_import_node
  -- java.lang.OutOfMemoryError: GC overhead limit exceeded
```

With reasonable memory settings, you might run into this error if Twitalytics started leaking memory. But with a stable application, it's not something you should ordinarily see.

The memory options used with TorqueBox or Warbler are similar to the command shown previously, but you pass them to the `java` command without the `-J` prefix. A reasonable configuration might look like this:

```
$ java -Xmx1024m -Xms256m -jar twitalytics.jar
```

You can always confirm that a JVM process is using the correct memory settings by inspecting it with JConsole or VisualVM. But you can also get the value of an individual flag with the jinfo command. Get the process ID for your JRuby server using jps as you did in Chapter 6, *Managing a JRuby Application*, on page 87. Then use that ID like this:

```
$ jps -l
46063 org/jruby/Main
46126 sun.tools.jps.Jps

$ jinfo -flag MaxHeapSize 46063
-XX:MaxHeapSize=67108864
```

The MaxHeapSize flag corresponds to the Xmx setting. You can see here that it's set to 64 megabytes.

The best setting for heap size depends on your application. You'll want it high enough that the GC doesn't have to run too often. But you'll want it low enough that the GC won't have to spend too much time collecting objects and freeing up memory when it does run. You'll also want to avoid setting the heap size higher than the available physical memory. Modern operating systems handle excessive memory allocation by swapping, but this behavior is particularly detrimental to Java applications.

A good rule of thumb is to size the heap so it is 30% consumed after a full GC run. You can determine this value by running your application until it reaches a steady state and running a full GC from JConsole or VisualVM as you did in Chapter 6, *Managing a JRuby Application*, on page 87.

The JVM's cap on memory consumption may seem like a frustrating antifeature, but it protects against the process reserving every last bit of system memory at runtime. Having a JVM crash is much more pleasant than having an entire system crash or start using swap memory, which degrades performance. Once you know the amount of memory an application needs, the cap becomes a safety net instead of a roadblock.

Determining the best maximum and minimum sizes for the heap is an iterative process. After running Twitalytics in a staging environment, you'll learn where its memory consumption tops out, and you can set the boundaries accordingly. The JVM provides some excellent tools to help with this by displaying memory consumption over time.

Setting Metaspace Size

In addition to heap space, the JVM allocates multiple chunks of memory for things that need to be stored off heap. This includes I/O buffers, thread

stacks, compiled code, and metaspace. The total memory footprint of a JVM process is the sum of the memory footprint for each of these components.

Metaspace contains metadata about the application the JVM is running. It contains class definitions, method definitions, and other things like that. The more classes you load into your app, the larger metaspace will be. Most JVM processes require only 40 to 50 megabytes for metaspace, but JRuby must load the entire Ruby standard library into memory at runtime, which moves its grand total a bit higher. A typical JRuby on Rails application will use 80 to 100 megabytes of metaspace memory from the outset. The more classes you create, and the more gems you include, the higher this will go.

Other than the different types of objects that go into metaspace, it's similar to heap. Its initial and maximum sizes can be configured, and it's even eligible for GC in some cases. Here are the relevant options:

-J-XX:MetaspaceSize This sets the initial size of metaspace. The default value for a 64-bit JVM is 20.75 megabytes.

-J-XX:MaxMetaspaceSize This sets the maximum size of metaspace. It's unlimited by default.

As an example, you might start a JRuby server with these options:

```
$ ruby -J-XX:MaxMetaspaceSize=100m -S bin/puma
```

If metaspace is overallocated, your application will crash with a `java.lang.OutOfMemoryError: Metaspace` error, so it's important not to set this too low.

Metaspace is an extremely common source of off-heap memory leaks. If you find your overall memory footprint is growing well beyond the maximum heap size, the first place to look is metaspace.

Configuring Heap Generations

Within the JVM heap are segments called generations. Each generation represents a set of objects that have be kept in memory for about the same amount of time, which makes it easier for the garbage collector to find them. The first generation is called the *young generation* or sometimes the *new generation*. This is where memory is initially allocated for most objects. When objects survive GC (that is, they're still in use) they're moved to the *old generation*, which is also called the *tenured generation*.

The most important of these is the young generation. It determines the size of both generations because it's reserved first, and the remaining heap space

is used for the old generation. The young generation will grow in tandem with the overall heap, but it also fluctuates as a percentage of the total heap.

Determining the appropriate size for the overall heap is still the most important part of memory tuning. But it's followed by sizing the young generation. It's important to point out, however, that you may not need to change the defaults. The JVM will start with a balanced configuration, and you should make changes only if you experience problems, such as long pauses for GC, which you'll learn about in the next section.

Before configuring the size of the young generation, you must understand the performance implications associated with it. If the young generation is too large, the GC will run less often and fewer objects will be promoted to the old generation. On the other hand, a smaller old generation will result in more full runs of the GC, which may be desirable depending on the algorithm. Different GC algorithms balance this in different ways, but they all use the same set of flags for configuring the size of these generations.

The following options can be set in the same way as the `-J-Xmx` and `-J-Xms` options:

-J-XX:NewRatio This sets the ratio of the young generation to the old generation. As the overall heap grows, this ratio will be maintained. For example: `-J-XX:NewRatio=1`

-J-XX:NewSize This sets the initial size of the young generation. For example: `-J-XX:NewSize=256m`

-J-XX:MaxNewSize This sets the maximum size of the young generation. For example: `-J-XX:MaxNewSize=1g`

-J-Xmn Shorthand for setting initial and maximum sizes of the young generation to the same value. For example: `-J-Xmn256m`

The best way to configure the size of the young generation is by defining it as a ratio of overall heap. This ensures better scalability because no matter how much the overall heap size changes, the young generation will remain at a reasonable level. The formula used to calculate the size of the young generation from the `NewRatio` is

```
Young Gen Size = Heap Size / (1 + NewRatio)
```

The default value for the `NewRatio` option is 2, which means the default young generation size for a JRuby process with an initial heap size of 500 megabytes is about 166 megabytes. If `NewRatio` is set to 1, then the default size will be 250 megabytes. Test this by starting a JRuby server with the following options:

```
$ ruby -J-XX:NewRatio=1 -S bin/puma
```

Once the process is running, capture its PID with jps and then run jinfo to inspect it, as shown here:

```
$ jps -l
73063 org/jruby/Main
73126 sun.tools.jps.Jps

$ jinfo -flag NewSize 73063
-XX:NewSize=262144000

$ jinfo -flag InitialHeapSize 73063
-XX:InitialHeapSize=524288000
```

In this example, the NewSize is roughly half the size of the initial heap. The actual size of the memory allocations always ends up being bigger than you specify. This is because the operating system allocates blocks of it at a time.

Figuring out the best value for the young generation ratio is difficult. But you will almost always want it to be a ratio and not an predefined value. You'll probably need to conduct benchmark tests against your application with different settings to find the best value. But even if you never change the ratio, an understanding of the heap generations and how they work is important because it informs your decisions when choosing a GC algorithm.

Choosing a Garbage Collector

You might wonder why the JVM has all these different types of memory. Why not just put all objects in one place?

The JVM segments memory to make garbage collection faster, which improves overall performance. When it comes down to it, this is the same reason you might choose a Hash instead of an Array. Having different sections of memory allows the GC to find things faster and run less often. Optimizing the GC in this way is important because when the GC runs, your application pauses. Garbage collection must happen quickly and as infrequently as possible.

In general, a garbage collector is form of automatic memory management. It finds objects that are no longer in use and releases the memory associated with those objects. To do this, the GC must periodically search the heap for orphaned objects. The collection process consumes resources and can hurt the performance of an application. But a good GC does this very quickly and even knows how to coalesce the chunks of memory it frees up to avoid fragmentation.

The JVM doesn't have just one GC, though. It offers four GC algorithms to choose from. But only one can run at a time, and the algorithm must be chosen when the process is started. It's an important decision because the GC affects the performance of your application more than any other component in the JVM. If you're having performance problems that cannot be solved by increasing the heap size, then the next place to look is the GC.

The four GC algorithms are described next.

Serial Collector

The serial garbage collector is the simplest of the four GC algorithms. It's also the least powerful. It was the default GC in older 32-bit JVMs because it was well suited to client machines and development environments. But it isn't used very often with Java 8.

The serial collector uses a single thread to process the heap. It pauses all application threads while doing this, even if it isn't doing a full GC (that is, it's processing only the young generation). You can enable it by adding the `-J-XX:+UseSerialGC` flag to your JRuby command.

Throughput Collector

The throughput collector, also known as the parallel collector, is the default GC algorithm for most JVMs. As its name implies, it uses multiple threads to process the heap. But it also requires a complete pause as it does this.

One intangible benefit of the throughput collector is that it's the oldest of the collection algorithms. The JVM core engineers have had more time to optimize, debug, and otherwise improve it. It's the most mature and dependable of the algorithms available.

You usually don't need to enable the throughput collector because it's the default. But if for some reason you need to, you can turn it on by adding the `-J-XX:+UseParallelGC -J-XX:+UseParallelOldGC` flags to your JRuby command. There are two flags because these options configure the algorithms to use for the young and old generations, respectively.

CMS Collector

The Concurrent Mark Sweep (CMS) collector is designed to eliminate some of the pauses associated with the serial and throughput collectors. The CMS algorithm still pauses all application threads when processing the young generation but doesn't require a pause for a full GC. Like the throughput collector, it uses multiple threads to process the heap.

The drawback of the CMS collector is that it increases CPU usage. As you can imagine, if the collector doesn't require a pause, then it's running at the same time as your application, which means the collector is competing for resources with your application threads. In addition, the background threads don't perform any heap compaction, which can lead to memory fragmentation.

If the heap becomes too fragmented to allocate new objects, or if the CMS collector doesn't get enough CPU time to complete its tasks, then the algorithm reverts to the behavior of the serial collector. It stops all application threads until it catches up and then returns to concurrent background processing.

You can enable the CMS collector by adding the `-J-XX:+UseConcMarkSweepGC -J-XX:+UseParNewGC` flags to your JRuby command.

G1 Collector

The Garbage First (G1) collector is designed to process large heaps with minimal pauses. A heap is considered to be large if it's greater than 4 gigabytes. The algorithm works by dividing the heap into a number of regions (around 2,048 by default). The collector uses concurrent background threads to watch these regions and then collects only those regions with the most garbage. A full pause is still required when collecting a region in the young generation, but a region in the old generation can usually be collected without any pause.

Like CMS, the drawback with G1 is increased CPU usage. The risk of fragmentation, however, is lower because the algorithm can partially compact the heap by moving objects from one region to another. For this reason, the G1 collector will most likely be the default collector in JDK 9.

You can enable the G1 collector by adding the `-J-XX:+UseG1GC` flag to your JRuby command.

Understanding how these garbage collectors work is just the first part of choosing one. The real test comes when you benchmark them against your application.

Benchmarking the Garbage Collector

Choosing a garbage collector comes down three characteristics: throughput, latency, and memory footprint. Because you don't have infinite resources, you'll have to choose one of them.

It's best to start by using the default collector. If you don't experience any performance problems, then stick with it. If you begin to encounter problems,

adjust the heap first. Use the 30% rule to find the optimal heap size. But remember, a heap that's too small *or* too big can hurt performance.

If you've optimized your heap size and still have performance problems, only then should you consider changing the garbage collector. Depending on your overall heap size, try either the CMS or G1 collector. For large heaps, try G1. For smaller heaps (less than 4 gigabytes), you may prefer the CMS collector.

In any case, make sure you benchmark the application and compare results. For a real app, you'll want to execute real behaviors. But for Twitalytics, you can use a synthetic benchmark to test different types of resource-bound operations.

To replicate these tests, you'll need to download the Faban HTTP Bench (fhb) command-line tool from the official website.[1] It's Java based and works on all platforms. Once you've downloaded the TAR file, unpack it and put the bin directory on your PATH environment variable. You're ready to run some tests.

You'll begin by measuring throughput—the rate at which a server can process requests. From the Twitalytics root directory, run the following command to create a benchmark controller:

```
$ bin/rails generate controller bench index \
    --assets=false --helper=false
```

Then open the app/controllers/bench_controller.rb file and edit the index() method so it looks like this:

Benchmark/twitalytics/app/controllers/bench_controller.rb

```
def index
  history = session[:history] || []
  history.shift if history.size > params[:save].to_i
  history << (1..100).map { rand(1 << 256) + rand(1 << 256) }
  session[:history] = history

  render :text => "done"
end
```

This code creates an Array of random values and puts it in the user's session (up to a limit defined by the save parameter). This will ensure that some load is put on the GC.

However, the default Rails session store is cookie based, which means the values your code creates won't be stored in memory. You'll need to change this session store in config/initializers/session_store.rb. Open the file and add this:

1. http://faban.org/download.html

Benchmark/twitalytics/config/initializers/session_store.rb

```
Rails.application.config.
  session_store :cache_store, key: '_twitalytics_session'
```

This configures Rails to use an in-memory session store. Note that in practice this isn't a great way of doing things, but it's necessary to perform this benchmark.

For the first test, run the app with the default GC, the throughput collector, and a 2-gigabyte heap size:

```
$ ruby -J-Xmx2g -S bin/puma
```

When the server is ready to receive requests, run fhb with the following options for output directory, -D, and number of clients, -c:

```
$ fhb -D tmp/fhb -c 1 http://localhost:3000/bench/index?save=25
```

The test will take about 10 minutes to run. When it completes, leave the JRuby server running and capture its process ID with jps and then run jstat, another JDK tool, to collect GC data:

```
$ jps -l
73063 org/jruby/Main
73126 sun.tools.jps.Jps
```

```
$ jstat -gcutil 73063
 S0     S1     E      O      M     CCS    YGC   YGCT   FGC   FGCT    GCT
62.50   0.00  78.16  80.87  98.32  98.19  1058  4.386   2   0.399  4.785
```

The GCT column displays how much time the process spent pausing to collect garbage. The other columns display details for young generation passes, old generation passes, and some other things.

The fhb tool creates a summary of its results in the tmp/fhb/1/summary.xml file. Each time you run the tool, a new directory will be created under tmp/fhb. The summary.xml file contains information on response time, number of operations, and much more.

Now you can repeat this process with different garbage collectors and different numbers of clients. Kill the Puma server and run the app again with this command:

```
$ ruby -J-XX:+UseG1GC -J-Xmx2g -S bin/puma
```

Then follow the same process of running fhb, capturing the PID and GC metrics, and saving the summary.xml file. Repeat both tests again with twenty fhb clients instead of one.

Let's examine the results for these four different tests on a server with two CPUs (an Amazon *c3.large* instance).

Clients	Throughput TPS	G1 TPS
1	19.0	18.1
20	109.4	88.9

During the first pass, with only one client (that is, one session), the throughput collector performed slightly more transactions per second (TPS) than the G1 collector. But as the number of clients increased (that is, the CPU became more saturated), the performance for the G1 dropped and the difference in TPS with twenty clients was much greater. Now let's look at how much time each collector spent pausing for GC.

Clients	Par GC Pause (seconds)	G1 GC Pause (seconds)
1	4.785	2.464
20	12.382	11.727

The G1 collector spent less time pausing to collect garbage. But it was also working concurrently and competing with the server for CPU time, which reduced the number of requests the server could process. It may come as no surprise then that the throughput collector excels at achieving higher throughput.

However, the latency associated with each run tells a different story. The Perc90 and Perc99 are much closer together, and the throughput collector falls behind for the max response time:

GC (w/ 20 clients)	Perc90 (seconds)	Perc99 (seconds)	Max Response Time (seconds)
Throughput	0.275	0.345	1.191
G1	0.275	0.355	0.895

This is due to the throughput collector pausing for a full GC pass. Even though it occurs infrequently, it can affect a small percentage of requests.

As the heap size grows and the throughput collector is required to pause for more full GC passes, the Perc99 of the G1 may become better relative to the throughput collector if sufficient CPU is available. This is a difficult scenario to reproduce in a synthetic benchmark, but it's something you'll want to watch for in the real world.

It's important not to draw too many conclusions from the results shown here. The lesson in this experiment is not the results themselves but the knowledge of how to run this test against a real app. You've learned that the throughput

collector is the best default and the G1 collector is the most predictable. You've also learned how to test your app and gather data about its performance. You can use this to collect information that will inform your decisions.

Joe asks:

What Is Perc99?

One measurement of web application performance is response time—the time between a client's request and the app's response. Faster response times are better.

It's common to track response time as an average, but averages tell only part of the story. Imagine an app with two endpoints: one with a response time of 10 ms and the other with a response time of 1,000 ms. If the fast method is called 90% of the time, the average response time for the app will be a respectable 109 ms. But this average disguises the fact that one part of the app is fast while another is very slow.

Measuring just the slowest response times can skew the results, too. If this app had a third method with a 30,000 ms response time that was called less than 1% of the time, the maximum response time would be 30,000 ms. This is also not representative of how most users experience the app.

The "Goldilocks" solution is measuring the 99th percentile of response times or the Perc99. Perc99 is the time for which 99% of requests are faster.[a] By definition, Perc99 accounts for the majority of an app's performance, without being susceptible to extreme outliers. Using the example app, the Perc99 would be 1,000 ms, which is a measure of the upper end of the response time most users would experience.

a. http://research.google.com/pubs/pub40801.html

Using invokedynamic

If you've been hanging around the JRuby community at all in the last couple of years, you've probably heard about invokedynamic. It's a new bytecode instruction that was added in Java 7 but didn't get the kinks worked out of it until Java 8. The invokedynamic instruction promised great performance improvements for JRuby, but it turned out to improve only a few specific cases. Synthetic micro-benchmarks show significant gains, but larger tests and macro-benchmarks haven't lived up to expectations. In the future, the JRuby core team will learn how to use this JVM feature to its full potential and probably turn it on selectively for you. But for now, JRuby keeps this feature off by default.

You may find it worthwhile to test your application with invokedynamic turned on. It's possible you'll see some performance gains depending on the kind of

logic you have in your app. You can enable invokedynamic by running your server with the -Xcompile.invokedynamic=true option, as shown here:

```
$ ruby -J-Xcompile.invokedynamic=true -S bin/puma
```

In order to determine if the option improved performance, you'll need to run some more benchmarks against your app and watch performance metrics. In this case, it's probably best to do that organically in production. And that brings us to the next chapter.

Wrapping Up

It's often said that magic is indistinguishable from advanced technology. And anyone who has seen the work produced by a professional performance engineer can attest that the results often seem mystical. But they're not. Performance tuning is the process of applying deep knowledge, experience, and intuition.

Experience and intuition can be achieved only through years of practice. But you've already learned the essential knowledge you need to tune a JRuby application. You've learned how to test your app in a controlled environment, and you've learned about many of the knobs you can turn as you're testing.

Running these tests takes time, though. Don't expect to start tuning your app before it gets real traffic, and don't expect tuning to be a one-time activity. You'll iterate on tuning just as you do with anything else in software.

The best way to inform your decisions as you tune is to collect as much data as possible. That includes not only the controlled setting of a benchmark but also an instrumented production runtime. In the next chapter you'll learn how to capture real performance data.

CHAPTER 8

Monitoring JRuby in Production

You've already learned how to get performance data from your app, the JRuby runtime, and the JVM. But profiling tools like VisualVM and JConsole are not well suited for use on a regular basis. They don't capture historical data well, and some of their features even degrade the performance of your app. Besides, when a problem has occurred it's usually too late to collect information with a profiler. A better solution for production monitoring is a background agent that reports to an external service.

In this chapter, you'll attach some monitoring services to your production server to track its performance, uptime, and fault tolerance. You'll learn how to create alerts for performance thresholds, capture detailed information when an error occurs, and keep a historical record of performance. A historical record can help you establish a performance baseline you can use to determine if the behavior you see is normal or an outlier. All of this will help you identify the root cause of errors, fix them faster, and even prevent them.

Let's begin with a service that's widely used in the Ruby on Rails community and works well with JRuby.

Installing the New Relic Gem

New Relic[1] is a popular application performance monitoring (APM) service that captures near real-time data about your web application's performance. You can use this data to monitor, troubleshoot, and tune a production web server.

New Relic's advantage over the other tools you've learned about is its ability to work with complete production systems, which may include more than one application. The JVM tools you used in Chapter 6, *Managing a JRuby*

1. http://newrelic.com/

Application, on page 87 work specifically with a single process, but a production app is usually composed of multiple processes for handling web requests, running background jobs, and more. With New Relic, you'll passively monitor your entire system. You'll hope that nothing goes wrong, but when an error does occur, you'll have all the data you need to debug it.

If you've ever used New Relic with a Ruby application, then you'll be pleased to learn that it works exactly the same way with JRuby as it does with MRI. You'll add an agent to a host server, which reports information back to the New Relic servers. Then you can sign in to the New Relic dashboard to view an analysis of all the data that was collected.

To use New Relic with Twitalytics, add the New Relic agent as a dependency. The agent is really just a gem, so open the app's Gemfile and add this line to the bottom of it:

```
gem "newrelic_rpm"
```

Now run bundle to download and install the gem locally. After it's installed, add the changes to Git by running these commands:

```
$ git add Gemfile Gemfile.lock
$ git commit -m "New Relic"
```

The agent will run in every process your application uses and report back to the New Relic servers. But it needs a place to put these reports. The next step is creating a New Relic app to collect the data on the receiving end. The quickest and easiest way to do this is by attaching the New Relic add-on to your Heroku app. Run this command:

```
$ heroku addons:create newrelic
```

You can even connect to this instance from your Docker containers. But if you'd prefer not to use Heroku, browse to the New Relic website and create an account.[2] Then follow the instructions for creating a new APM app and configuring your New Relic API key.

Whether you're using Heroku or not, you'll need to set an environment variable with the name of your New Relic app. The name doesn't have to be the same as your Heroku app name, but that's what we'll use here. To configure the name on Heroku, run the following command:

```
$ heroku config:set NEW_RELIC_APP_NAME="obscure-fjord-4138"
```

2. http://newrelic.com/

If you're using Docker with your private infrastructure, you can add this line to your Dockerfile:

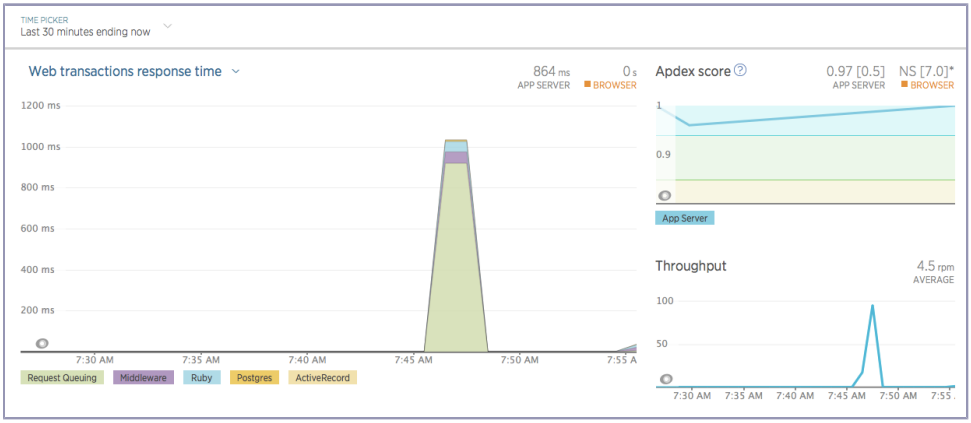
```
ENV NEW_RELIC_APP_NAME obscure-fjord-4138
```

Now redeploy your app by running either `git push heroku master` or `docker push`.

After Twitalytics restarts, make a few requests and wait about five minutes to give the app time to report back to the New Relic servers. When finished, run this command to open the instance attached to your Heroku app:

```
$ heroku addons:open newrelic
```

If you're not using Heroku, browse to the New Relic APM dashboard.[3] In either case, you'll see something like this:



The large graph is an overview of response times. The smaller graph on the bottom right is an overview of throughput. The Application Performance Index (Apdex) score in the upper right is a measurement of user satisfaction. The Apdex method converts many measurements into one number on a uniform scale.[4] A high Apdex score is good—it will decrease as response time increases.

Scroll down to see the rest of the overview page. You'll find graphs for error rates and common transactions. When you start to experience problems with an app, you can use the dashboard to drill into specific transactions or limit the time frame you're inspecting.

How will you know when you're having a problem? That's where alerting helps.

3. https://docs.newrelic.com/docs/apm/applications-menu/monitoring/viewing-your-applications-list
4. https://docs.newrelic.com/docs/apm/new-relic-apm/apdex/apdex-measuring-user-satisfaction

Joe asks:
What's Apdex?

Apdex is an industry standard for measuring user satisfaction with the response time of an application or service. The response time of the app is based on a set threshold, which is defined by the application's owner. All responses handled within this threshold or in less time are considered as satisfying the user.

For example, if the threshold (T) is 1.2 seconds and a response completes in 0.5 seconds, then the user is satisfied. All responses greater than 1.2 seconds dissatisfy the user. These measurements are then tracked in three categories:

Satisfied The response time is less than or equal to T.

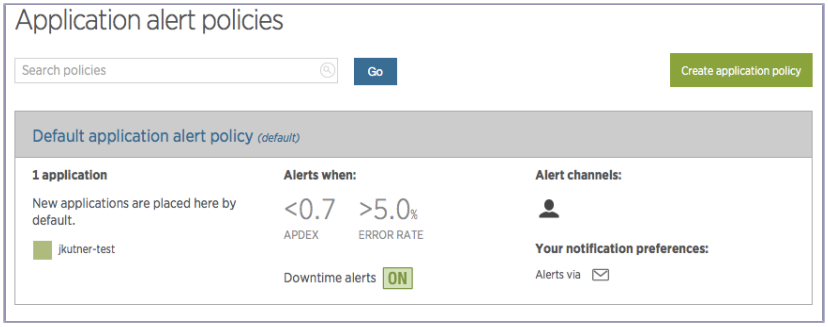
Tolerating The response time is greater than T and less than or equal to 4T. In this example, 4 x 1.2 = 4.8 seconds as the maximum tolerable response time.

Frustrated The response time is greater than 4T.

The overall Apdex score is a ratio of the number of satisfied and tolerating requests to the total requests made. Each satisfied request counts as one request, while each tolerating request counts as half a satisfied request.

Creating a New Relic Alert

From the New Relic dashboard, select the Alerts options from the menu. Then click the Edit Alert Policies button, and you'll see a default policy for your app, as shown here:



This policy is designed to alert you if the application shuts down for any reason. But it needs to have a few options set before it can work. The first thing you'll configure is a Ping URL. This is the URL New Relic uses to determine if your application is running or not.

Click the policy, and then click the Edit button. In the Applications menu on the right side, select your app. A dialog will open with an empty field for the

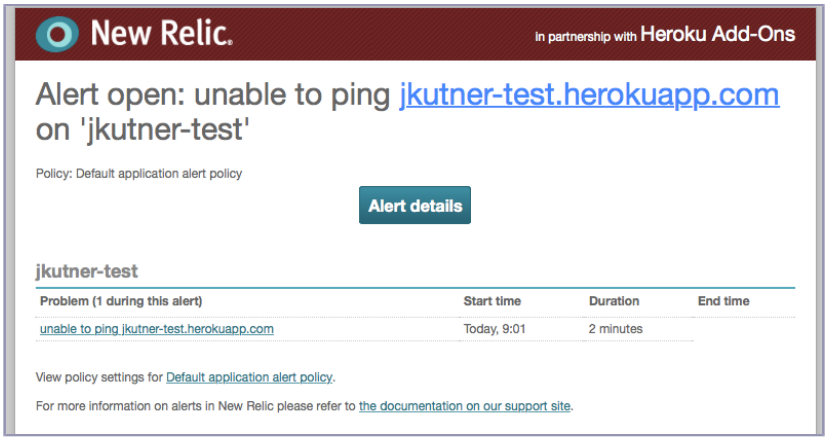
Ping URL. Put the URL for your app in the field and click Save Your Changes. Now the alert can trigger, but it needs somewhere to send the message.

Click the Alert Channels tab in the policy form and then select Create Channels. In the dialog that appears, select Email and then Create Channel. Enter your email address in the text box, and click Save My Changes.

Now New Relic is ready to alert you. Test it out by shutting down your app. On Heroku, you can run this command:

```
$ heroku ps:scale web=0
```

After a few moments, you'll receive an email that looks like this:



New Relic can send alerts when an application crashes or performance degrades or for just about any parameters you're interested in. That's an important feature because early detection is the key to solving problems.

Degraded performance is a bit of a nebulous and subjective problem, though. Some problems are more explicit and bubble up as exceptions. New Relic can help with uncaught exceptions, but it's often better to have a dedicated error-tracking service.

Handling Errors with Rollbar

Rollbar is a service for tracking and reproducing exceptions and errors. It can collect, de-duplicate, and alert on error conditions. It also provides a dashboard and analysis tools you can use to investigate specific incidents.

Having an error tracker helps you monitor the unexpected. When you're monitoring performance, you know to look at heap and CPU usage. But unexpected errors often come from the places you never thought to look at. Rollbar watches for exceptions globally within the runtime, which means it

can catch any error—even those that originate outside your code. Let's add Rollbar to Twitalytics.

If you're using Heroku, attach the Rollbar add-on to your app by running this command:

```
$ heroku addons:create rollbar
```

If you're not using Heroku, browse to the Rollbar website and create an account.[5] Then follow the instructions for trying Rollbar for free.

In either case, the next step is adding the Rollbar gem to Twitalytics. Much like New Relic, Rollbar uses a client gem that communicates information about the app back to the Rollbar servers. Add this line to your Gemfile:

```
gem "rollbar"
```

Then run Bundler to install the gem, and run Rake to regenerate your Rails binstubs:

```
$ bundle install --binstubs
$ rake rails:update:bin
```

Rollbar provides a helpful Rails generator that creates the necessary configuration. Use this command to run it:

```
$ bin/rails generate rollbar
```

A config/initializers/rollbar.rb file is generated in your app. The initializer contains everything needed to connect your app to Rollbar except for the API key, which it retrieves from an environment variable. To test Rollbar locally, you'll need to set the variable in your development environment.

To get the value of your API key on Heroku, run this command:

```
$ heroku config:get ROLLBAR_ACCESS_TOKEN
```

If you're not using the Heroku Rollbar instance, you can get your Rollbar API key from the Rollbar website.[6] In either case, set the key as the value for the ROLLBAR_ACCESS_TOKEN environment variable locally by running this command on Mac and Linux, with the value of the key replacing *xxxx*:

```
$ export ROLLBAR_ACCESS_TOKEN="xxxx"
```

Or run this command on Windows with the value of the key replacing *xxxx*:

```
C:\> set ROLLBAR_ACCESS_TOKEN="xxxx"
```

5. https://rollbar.com/signup
6. https://rollbar.com/

From the same terminal session, test Rollbar by running this command:

```
$ bin/rake rollbar:test
...
RollbarTestingException (Testing rollbar with "rake rollbar:test"....
...
```

The test must generate an exception—because that's what Rollbar captures—so it may look like the test failed. But as long as you see the line containing RollbarTestingException and "If you can see this, it works," then the test was successful.

You're ready to deploy. Add your changes to Git by running these commands:

```
$ git add config/initializers/rollbar.rb
$ git commit -m "rollbar"
```

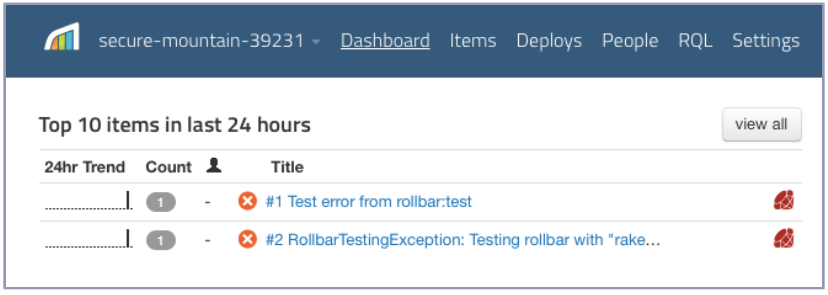
Then deploy by running either git push heroku master or docker push depending on your choice of platform.

When your deployment is complete, run this command to open the dashboard for the Heroku Rollbar instance:

```
$ heroku addons:open rollbar
```

If you're not using Heroku, browse to your app's dashboard on the Rollbar website.

Either way, you'll see something like this:



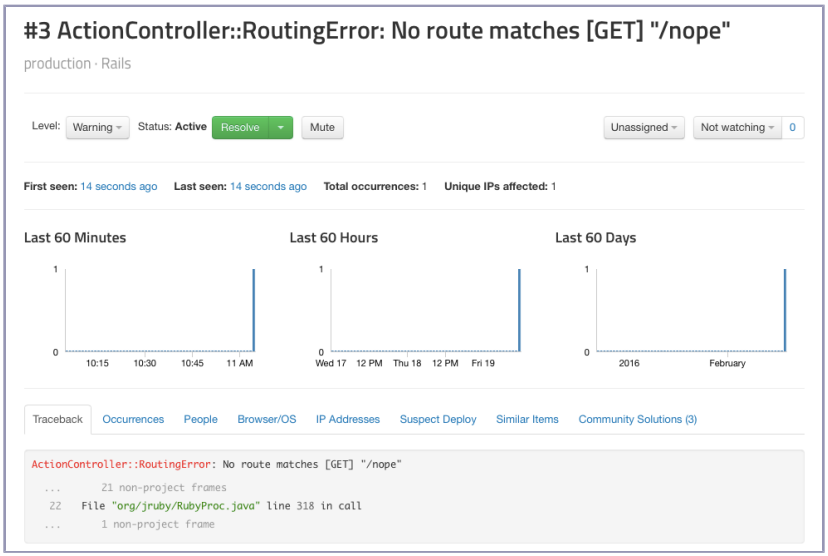
The dashboard contains two *items*, which represent the errors generated by the test you ran from the terminal a moment ago.

To create an error occurrence from the real running app, simply browse to a URL that doesn't exist. For example, run this command on Heroku:

```
$ heroku open nope
```

Or browse to the /nope route on your private infrastructure. For both, you'll see an error page with the text "The page you were looking for doesn't exist."

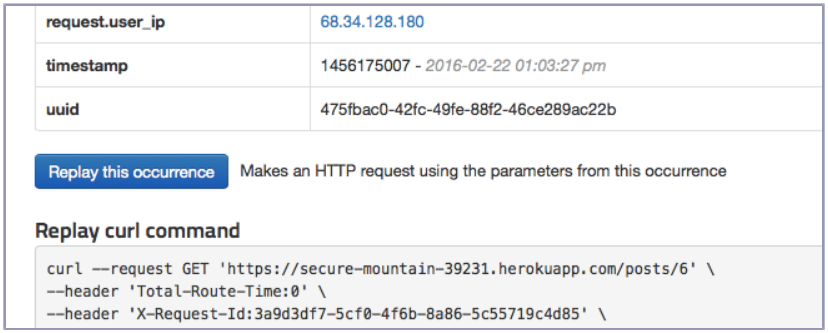
Return to the Rollbar dashboard and refresh the page. You'll see a new error of type ActionController::RoutingError. Click it to drill into the details and you'll see something like this figure:



Rollbar tracks how many times an error occurred, how often it occurred, and how many users it affected.

Refresh the browser page containing the 404 error a few times, and then return to the Rollbar page. You'll see an increase in the graph and the count of occurrences. Click the Occurrences tab, and you'll see a list of each error instance. Click one of them to see the detail view.

Now scroll down past the details about the user agent, session ID, timestamp, and other information until you find the Replay This Occurrence button. It will look like this:



If you're using Heroku or a publicly accessible Docker instance, click the button. It will make a new request to your server, mimicking the original request associated with the error.

For most applications, this is all you need to get detailed information about unexpected errors in your app. But in some cases you'll want to customize how Rollbar reports errors.

Customizing Rollbar Reporting

By default, the Rollbar client will report any uncaught exception. But you can also report exceptions and other types of events manually.

For example, you may want to trap an exception, report it, and then proceed with some default logic. In Twitalytics, this is necessary when a nonexistent post is requested. Open the app/controllers/posts_controller.rb file, and find the set_post method. Modify it to look like this:

```
def set_post
  @post = Post.find(params[:id])
rescue ActiveRecord::RecordNotFound => e
  Rollbar.error(e, "Requested an unknown Post")
  Rollbar.log("Defaulting to first Post")
  @post = Post.first
end
```

The rescue clause calls Rollbar.error if an ActiveRecord::RecordNotFound is raised. This will happen if no record is found in the database for the ID. It then sets @post to the first post in the database as a default. In this way, you can capture some details about the error but avoid showing an error to the user.

Commit this change to Git by running

```
$ git add app/controllers/posts_controller.rb
$ git commit -m "rollbar error"
```

Then deploy with either git push heroku master or docker push. When the deployment completes, open a browser to a URL for a post that doesn't exist. You can run this command for Heroku:

```
$ heroku open posts/9999
```

Instead of an error, you'll see the details for the first post in the database (the post with ID 1). When you return to your Rollbar dashboard, you'll see a Warning item for the ActiveRecord::RecordNotFound error as well as a log entry for the default message.

Rollbar also has the ability to track deployments, which help determine the version of the app that was running when an error occurred. It can integrate with tools such as GitHub, Slack, HipChat, JIRA, Pivotal Tracker, Trello, Campfire, and more. As it collects more and more errors over time, you'll be able to discover trends in how people are using your app and what specific problems they're encountering.

Wrapping Up

The moral of this chapter is *don't wait until it's too late*. Performance and error monitoring are the kind of services you realize you need only after you're up the proverbial creek without them.

Because these services are so critical, it's also a good idea to use them in staging and test environments. You can use them to collaborate with a quality assurance team or ensure the monitoring system is working before you go to production. The minute you receive your first requests from real customers is the minute you need real application performance monitoring.

In the next chapter, we'll move away from our production runtime and take a look at the bigger picture of how you deliver code to your customers. You'll implement a modern deployment technique alluded to in some of the earlier chapters.

CHAPTER 9

Using a Continuous Integration Server

Continuous integration (CI) is the process of applying quality control validations to a code base every time it changes. In the case of Twitalytics and most Ruby applications, this means running unit tests after each commit. But it is not enough to rely on developers to run these tests because their local environments may differ from your production environment. Developers do lots of stuff on their computers that can affect a test run, such as installing software and setting environment variables. To ensure the reliability of your tests, you must run them the same way every time. This principle also applies to deployment.

When you deploy from your development machine to production, you run the risk of the local configuration affecting the artifacts you publish. But a CI server provides a static environment resembling the production server. The result is a more consistent and reliable process for publishing releases of your software.

In this chapter, you'll introduce continuous integration into your process by using the Jenkins CI server[1] to run your tests and deploy Twitalytics to a production server. This will result in not only continuous integration but also continuous deployment. Let's begin by getting to know Jenkins.

Installing Jenkins

Jenkins is an open source application for continuously building and testing software. An excellent publicly available example of a running Jenkins instance is the TorqueBox CI service.[2] This instance runs on the cloud-based CloudBees service, but you can also run a self-hosted Jenkins instance.

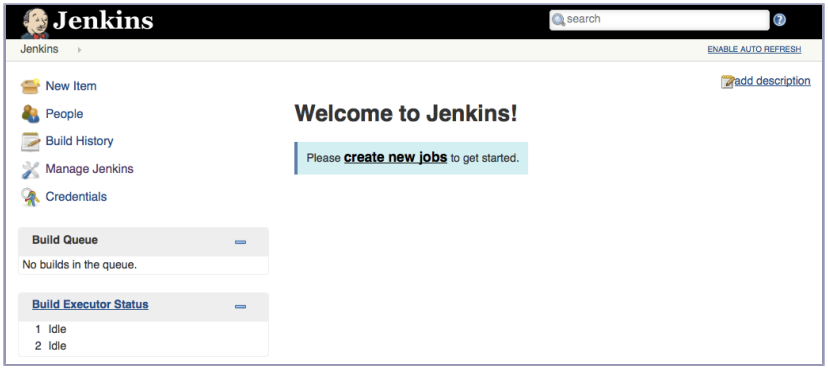
1. http://jenkins-ci.org/
2. https://projectodd.ci.cloudbees.com/

You'll use Jenkins to test your application and deploy it each time changes are pushed to your repository. But rather than setting up a cloud-based or virtual CI server, you'll run Jenkins on your local machine. There are several binary distributions of Jenkins for specific platforms including an executable WAR file distribution, which is similar to the executable WAR file you created for the Twitalytics stock-service in Chapter 1, *Getting Started with JRuby*, on page 1. This will ensure that the steps in the chapter are the same on all platforms.

Download the WAR file from the official Jenkins website.[3] Put the downloaded file into your home directory and run it with the following command (but make sure you don't already have Warbler or TorqueBox running because they use the same ports):

```
$ java -jar jenkins.war
```

Jenkins is running. Browse to http://localhost:8080, and you'll see the Jenkins dashboard, as pictured here:



Jenkins isn't able to run tests for a JRuby app out of the box, though. You'll need to add a few extensions.

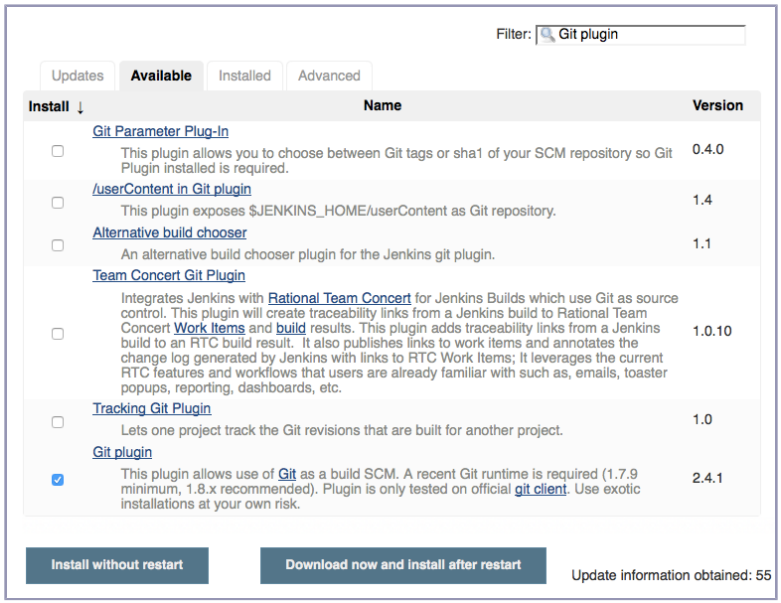
Installing Jenkins Plugins

The base Jenkins server is missing Git and RVM dependencies, which are needed to run JRuby tests. You can install these features as easy-to-use Jenkins plugins.

From the Jenkins dashboard, click the Manage Jenkins link in the left navigation pane of the page. Then click the Manage Plugins link on the next page. On the plug-ins page, select the Available tab. This will bring up a list of plug-ins you can install to the Jenkins server.

3. http://mirrors.jenkins-ci.org/war/latest/jenkins.war

In the filter box near the top of the page, enter the value “Git plugin.” In the filtered list, check the box next to the plugin with that name, as shown here:



Then click the Install Without Restart button at the bottom of the page. It will take a few moments to download the plugin and install it.

If you’re not running Jenkins on a Windows machine, return to the Manage Plugins page. Filter for the value RVM and select the corresponding plugin. Then click the Install Without Restart button at the bottom of the page. When the installation completes, return to the Jenkins dashboard at http://localhost:8080.

Your CI server is ready do some work. But before you can add a job that runs your tests, you’ll need to tell Jenkins how to access your code. To do this, you’ll create a depot for your Git repository.

Creating a Git Depot

A Git depot is a bare clone of a Git repository, which means it’s a repository that doesn’t have a staging area where edits can be made and committed. Instead, it can only be pushed to and pulled from. A depot is usually used to share changes between distributed copies of the repository. The best example of this is a GitHub project.

Create a depot for the Twitalytics repository so that Jenkins can check out the code and run the tests against it. You could do this by pushing the code to GitHub or a similar service, but you’ll use a local depot in this example.

Run the following command with the --bare option to create a copy of your Twitalytics repository in the ~/depot/twitalytics.git directory:

```
$ git clone --bare ~/code/twitalytics ~/depot/twitalytics.git
```

Next, add the clone as the remote origin in your Twitalytics repository:

```
$ cd ~/code/twitalytics
$ git remote add origin ~/depot/twitalytics.git/
```

Now you can push to the depot like this:

```
$ git push depot
Everything up-to-date
```

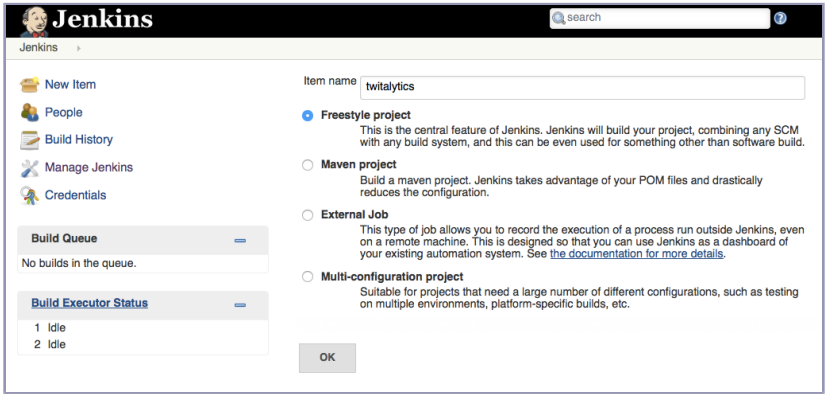
Everything is already up to date because you haven't made any changes since you cloned the repository.

Your Git depot is ready. Now you'll set up Jenkins to use it.

Creating a Jenkins Job

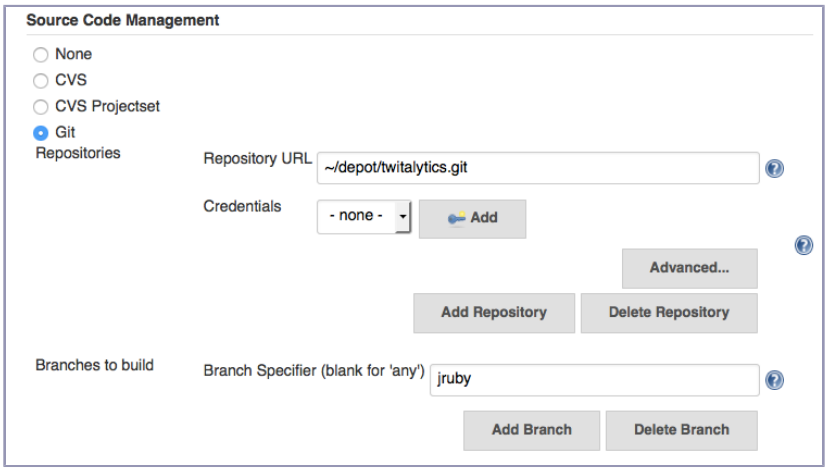
Jenkins uses the concept of a job to represent any kind of user-defined task. A job can run your tests, migrate a database, push a WAR file out to a server, or run static analysis tools like Brakeman[4] to provide reports on code correctness. A job can even do things unrelated to your application, such as install software packages on the host. The possibilities are endless.

Let's create a job that runs the tests for Twitalytics. This will automate your build process and make it more consistent. Browse to the Jenkins dashboard at http://localhost:8080 and follow the Create New Jobs link on the front page. On the next page, enter "twitalytics" for the name of the job, select Freestyle Project, and then click the OK button, as shown in the following figure.



4. http://brakemanscanner.org/

This opens a page containing a form you can use to configure the job. Scroll down to the Source Code Management section and fill in the Git Repository URL field with the location of your Twitalytics depot, as pictured here (note that you'll need to replace the ~ with the full path to your depot):



Then fill in the branch specifier with `jruby` because that's the version of Twitalytics you're going to build and deploy. Each time this job runs, it will check out a fresh copy of your `jruby` branch to ensure it's testing and deploying the latest code.

If you're not using Windows, scroll down to the Build Environment section. Check the box next to the option Run the Build in an RVM-Managed Environment and enter "jruby-9.0.5.0" in the text box that follows it.

Next, scroll down to the Build section. Select the Add Build Step drop-down and choose Execute Shell or Execute Windows Batch Command, depending on your platform. This reveals a Command text box you'll need to fill in with the steps for running your tests, like this:

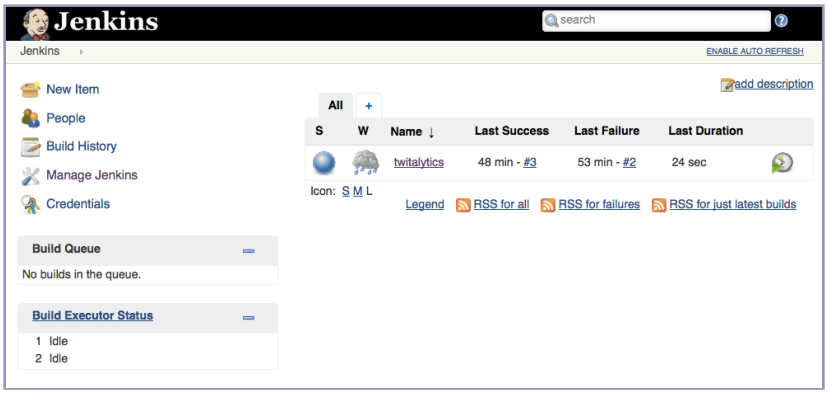
```
bundle install --jobs=3 --retry=3 --without production --binstubs
RAILS_ENV=test bin/rake test
```

These commands will install your dependencies, create a fresh test database, and finally run your tests.

Scroll to the bottom of the page and click Save. Your job is ready to run.

For the first run, you'll execute the job manually. You'll automate it later. Browse to the Jenkins dashboard, and click the twitalytics link for your job. Then click the Build Now link on the page that follows. Shortly after clicking it, you'll see the job show up in the build queue on the bottom left of the page.

When the job finishes, the gray dot next to its entry in the queue will turn blue, as shown in the primary pane here:



The blue dot means the job was successful.

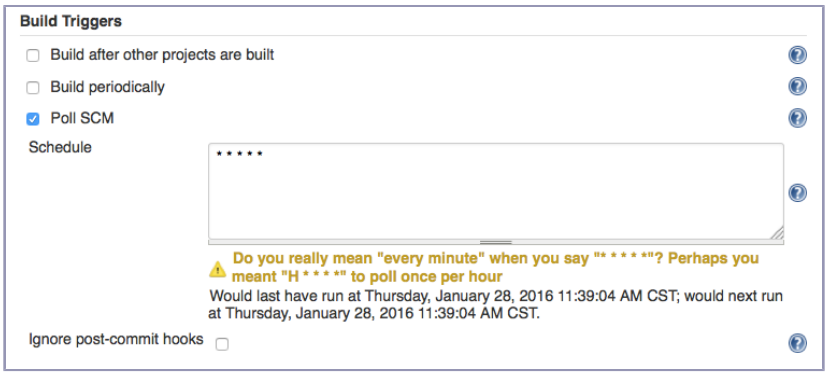
Now drill down to take a closer look at what happened. Click the entry for the most recent build in the queue on the twitalytics job page. Then follow the Console Output link, which will take you to a page with the full output of the job. Scroll down to the bottom and you'll see some text like this:

```
+ RAILS_ENV=test
+ bin/rake test
Run options: --seed 2214

# Running:

.......

Finished in 3.399273s, 2.0593 runs/s, 3.8243 assertions/s.

7 runs, 13 assertions, 0 failures, 0 errors, 0 skips
Finished: SUCCESS
```

Your CI job is working! Let's set it up to run automatically so you won't need to click the Build Now link every time you want it to run. On the Configuration page for the job, scroll down to the Build Triggers section, and select the Poll SCM box. In the text field, enter the cron string `* * * * *`, as shown in the figure on page 139.

This will schedule the server to poll the Git depot for changes every minute. If it finds that new changes have been checked in since the last build, it will run the job again. Click the Save button to make sure your change is remembered.

Now that Jenkins is watching the repo for updates, you can use it to automatically deploy your app.



Enabling Continuous Delivery

Continuous integration is a great step toward improving your build and deployment processes. But it also opens the door to continuous delivery, which is the cornerstone of a DevOps deployment strategy.

Continuous delivery is the process of releasing an application to production anytime a change is made to its code. In many cases, the entire process is automated, and there's no human intervention between committing the change and releasing it to production. In that case, the process is called continuous deployment. It may seem like a scary proposition, but its success has been proven by companies like Etsy, Netflix, and others.[5] [6]

You can enable continuous delivery for Twitalytics right from Jenkins. Open the configuration for the job you created earlier. In the Build section, where you wrote the commands for the rake task, add the commands to push the code to Heroku. The complete script should look like this:

```
bundle install --jobs=3 --retry=3 --without production --binstubs
RAILS_ENV=test bin/rake test

heroku git:remote obscure-fjord-4138
git checkout jruby
git push -f heroku master
```

As before, replace "obscure-fjord-4138" with the name of your Heroku app. These commands will add the Heroku app as a Git remote, check out the Git jruby branch (because Jenkins normally works from a detached commit even though the code is the same), and push the repo to Heroku just as you would from your development command line.

5. http://techblog.netflix.com/2013/08/deploying-netflix-api.html
6. https://www.thoughtworks.com/insights/blog/case-continuous-delivery

Click Save to ensure your changes are made permanent. Then return to the dashboard and click Build Now to force the job to run. Browse to the build output page for the job, and watch as the Git output for the Heroku deployment is logged. Once the job is complete, you can check that the deployment was successful by running `heroku open` from your app's repo.

If you're deploying to private infrastructure or some platform other than Heroku, you can always replace the `git push` with a `docker push`. No matter how you deploy your app, you'll be able to do it from Jenkins.

The benefit of continuous delivery is agility. The more often you deploy, the easier it is to release features and fix bugs. That, in turn, lowers the cost of accidentally introducing bugs into production because they can easily be rolled back or fixed by another deployment.

Wrapping Up

You've turned your development environment into a CI server. But setting up Jenkins on a dedicated CI server or cloud-based server will use the same process and configuration. Once you've moved CI into its own environment, you can begin to change the way you manage your infrastructure.

If all deployments are run from CI, you no longer need to give developers access to the staging or production servers. You can lock them down to ensure deployments come from a single source. This will improve the consistency and reliability of your application.

Building and deploying from a CI server is an essential part of an effective deployment process. It ensures the reliability of your code by testing it in an isolated and consistent environment before sending it out to the world.

But adopting continuous integration is more than just using new tools. It also changes the end-to-end process you use to deliver software to your customers. It's the first step on the path to continuous deployment, which can greatly improve a development team's ability to respond to bugs and failures. This can solve many but not all of your deployment problems.

The most difficult part of deployment is that every environment is different. Technologies, processes, and team expertise all play a role in determining how an application is delivered to customers. This makes it hard to create reusable deployment tools. Thus, the individuals responsible for a deployment have to be intimately familiar with the technologies they're working with.

This book has provided a survey of the frameworks and tools you can use to support a JRuby deployment. But as you go forth and build more advanced

and complex applications, you'll need to dig deeper into the particular technologies you've chosen for your product. Nothing is more helpful to this task than the support of the community.

The communities that surround the JRuby technologies are some of the most helpful and supportive in the industry. The JRuby core team actively addresses issues on the mailing list and on IRC. The TorqueBox and Warbler teams are equally helpful, and the rest of the ecosystem is just the good-old Ruby community. The technologies covered in this book are mature, and the number of users adopting them has grown to a level where you can easily find answers to your questions on Stackoverflow.com and mailing lists. Many developers have already worked through the problems you'll encounter, and they'll often share their wisdom with you.

Likewise, it's important that you share what you learn with the community. As you grow in your ability to run and manage a JRuby application, be sure to help others, because they may return the favor one day.

Index

J

U

V

W

X

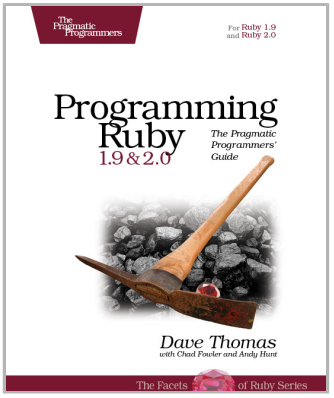
Y

Z

Level Up Your Ruby

Time to stop just “using a little Ruby.” See what you’re missing in basic Ruby and advanced metaprogramming techniques.

Programming Ruby 1.9 & 2.0 (4th edition)

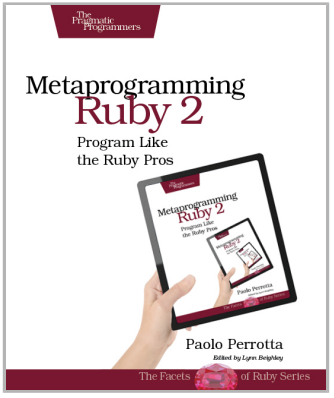


Ruby is the fastest growing and most exciting dynamic language out there. If you need to get working programs delivered fast, you should add Ruby to your toolbox.

This book is the only complete reference for both Ruby 1.9 and Ruby 2.0, the very latest version of Ruby.

Dave Thomas, with Chad Fowler and Andy Hunt
(888 pages) ISBN: 9781937785499. $50
https://pragprog.com/book/ruby4

Metaprogramming Ruby 2



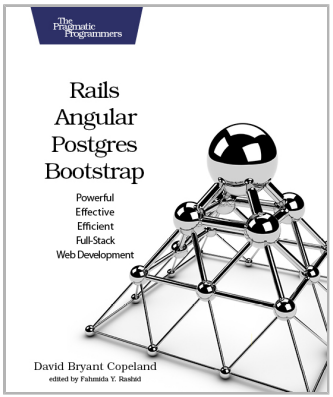
Write powerful Ruby code that is easy to maintain and change. With metaprogramming, you can produce elegant, clean, and beautiful programs. Once the domain of expert Rubyists, metaprogramming is now accessible to programmers of all levels. This thoroughly revised and updated second edition of the bestselling *Metaprogramming Ruby* explains metaprogramming in a down-to-earth style and arms you with a practical toolbox that will help you write your best Ruby code ever.

Paolo Perrotta
(278 pages) ISBN: 9781941222126. $38
https://pragprog.com/book/ppmetr2

Rails and More...

Explore the Ruby on Rails ecosystem for easier web development.

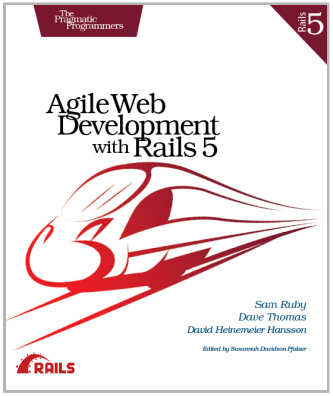
Rails, Angular, Postgres, and Bootstrap



As a Rails developer, you care about user experience and performance, but you also want simple and maintainable code. Achieve all that by embracing the full stack of web development, from styling with Bootstrap, building an interactive user interface with AngularJS, to storing data quickly and reliably in PostgreSQL. Take a holistic view of full-stack development to create usable, high-performing applications, and learn to use these technologies effectively in a Ruby on Rails environment.

David Bryant Copeland
(306 pages) ISBN: 9781680501261. $35
https://pragprog.com/book/dcbang

Agile Web Development with Rails 5



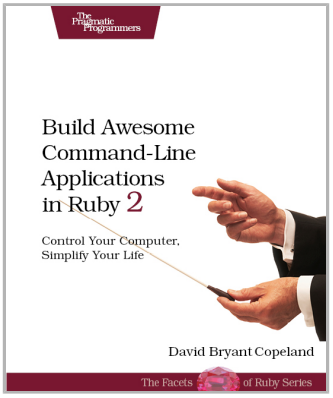
Rails 5 and Ruby 2.2 bring many improvements, including new APIs and substantial performance enhancements, and the fifth edition of this award-winning classic is now updated! If you're new to Rails, you'll get step-by-step guidance. If you're an experienced developer, this book will give you the comprehensive, insider information you need for the latest version of Ruby on Rails.

Sam Ruby
(450 pages) ISBN: 9781680501711. $46
https://pragprog.com/book/rails5

Ruby and the Command Line

Ruby is perfect for text processing and command-line scripts. See how to do it well.

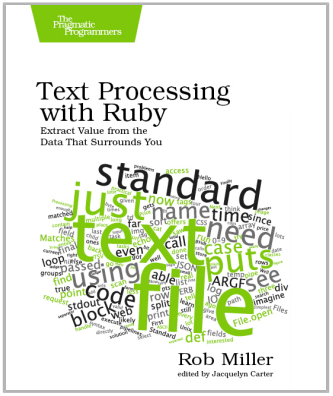
Build Awesome Command-Line Applications in Ruby 2



Speak directly to your system. With its simple commands, flags, and parameters, a well-formed command-line application is the quickest way to automate a backup, a build, or a deployment and simplify your life. With this book, you'll learn specific ways to write command-line applications that are easy to use, deploy, and maintain, using a set of clear best practices and the Ruby programming language. This book is designed to make *any* programmer or system administrator more productive in their job. This is updated for Ruby 2.

David Copeland
(222 pages) ISBN: 9781937785758. $30
https://pragprog.com/book/dccar2

Text Processing with Ruby



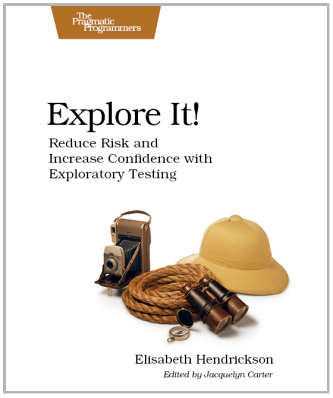
Whatever you want to do with text, Ruby is up to the job. No matter what the source – web pages, databases, the contents of files – learn how to acquire the text and get it into your program. Explore techniques to process that text and then output the transformed or extracted text. Cut even the most complex text-based tasks down to size and learn how to master regular expressions, scrape information from Web pages, develop reusable utilities to process text in pipelines, and more.

Rob Miller
(272 pages) ISBN: 9781680500707. $36
https://pragprog.com/book/rmtpruby

Explore Testing and Cucumber

Explore the uncharted waters of exploratory testing and beef up your automated testing with more Cucumber.

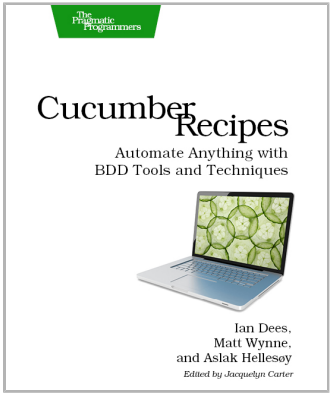
Explore It!



Uncover surprises, risks, and potentially serious bugs with exploratory testing. Rather than designing all tests in advance, explorers design and execute small, rapid experiments, using what they learned from the last little experiment to inform the next. Learn essential skills of a master explorer, including how to analyze software to discover key points of vulnerability, how to design experiments on the fly, how to hone your observation skills, and how to focus your efforts.

Elisabeth Hendrickson
(186 pages) ISBN: 9781937785024. $29
https://pragprog.com/book/ehxta

Cucumber Recipes



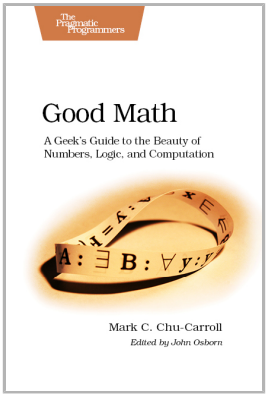
You can test just about anything with Cucumber. We certainly have, and in *Cucumber Recipes* we'll show you how to apply our hard-won field experience to your own projects. Once you've mastered the basics, this book will show you how to get the most out of Cucumber—from specific situations to advanced test-writing advice. With over forty practical recipes, you'll test desktop, web, mobile, and server applications across a variety of platforms. This book gives you tools that you can use today to automate any system that you encounter, and do it well.

Ian Dees, Matt Wynne, Aslak Hellesoy
(274 pages) ISBN: 9781937785017. $33
https://pragprog.com/book/dhwcr

The Joy of Math and Healthy Programming

Rediscover the joy and fascinating weirdness of pure mathematics, and learn how to take a healthier approach to programming.

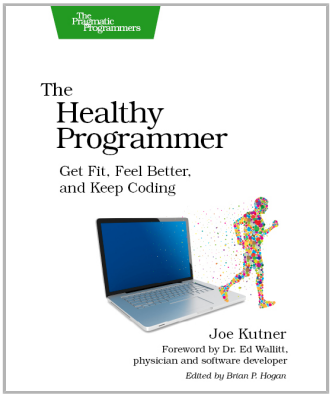
Good Math



Mathematics is beautiful—and it can be fun and exciting as well as practical. *Good Math* is your guide to some of the most intriguing topics from two thousand years of mathematics: from Egyptian fractions to Turing machines; from the real meaning of numbers to proof trees, group symmetry, and mechanical computation. If you've ever wondered what lay beyond the proofs you struggled to complete in high school geometry, or what limits the capabilities of the computer on your desk, this is the book for you.

Mark C. Chu-Carroll
(282 pages) ISBN: 9781937785338. $34
https://pragprog.com/book/mcmath

The Healthy Programmer



To keep doing what you love, you need to maintain your own systems, not just the ones you write code for. Regular exercise and proper nutrition help you learn, remember, concentrate, and be creative—skills critical to doing your job well. Learn how to change your work habits, master exercises that make working at a computer more comfortable, and develop a plan to keep fit, healthy, and sharp for years to come.

This book is intended only as an informative guide for those wishing to know more about health issues. In no way is this book intended to replace, countermand, or conflict with the advice given to you by your own healthcare provider including Physician, Nurse Practitioner, Physician Assistant, Registered Dietician, and other licensed professionals.

Joe Kutner
(254 pages) ISBN: 9781937785314. $36
https://pragprog.com/book/jkthp

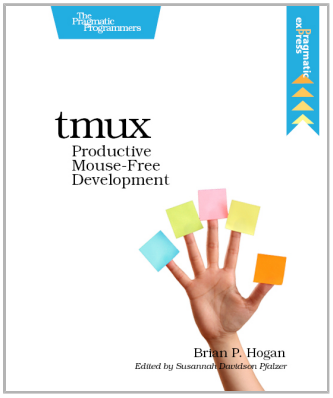
Long Live the Command Line!

Use tmux and Vim for incredible mouse-free productivity.

tmux

Your mouse is slowing you down. The time you spend context switching between your editor and your consoles eats away at your productivity. Take control of your environment with tmux, a terminal multiplexer that you can tailor to your workflow. Learn how to customize, script, and leverage tmux's unique abilities and keep your fingers on your keyboard's home row.

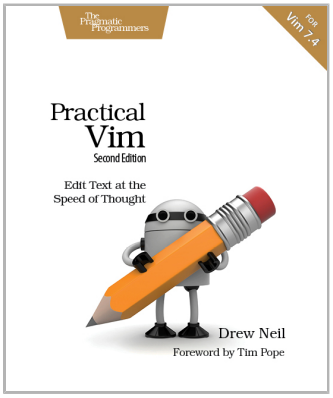
Brian P. Hogan
(88 pages) ISBN: 9781934356968. $16.25
https://pragprog.com/book/bhtmux



Practical Vim, Second Edition

Vim is a fast and efficient text editor that will make you a faster and more efficient developer. It's available on almost every OS, and if you master the techniques in this book, you'll never need another text editor. In more than 120 Vim tips, you'll quickly learn the editor's core functionality and tackle your trickiest editing and writing tasks. This beloved bestseller has been revised and updated to Vim 7.4 and includes three brand-new tips and five fully revised tips.

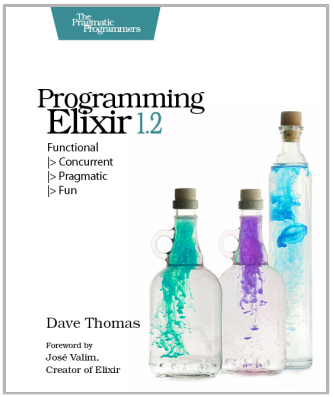
Drew Neil
(354 pages) ISBN: 9781680501278. $29
https://pragprog.com/book/dnvim2



Put the “Fun” in Functional

Elixir 1.2 puts the “fun” back into functional programming, on top of the robust, battle-tested, industrial-strength environment of Erlang. Add in the unparalleled beauty and ease of the Phoenix web framework, and enjoy the web again!

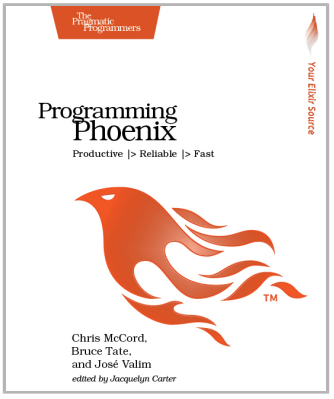
Programming Elixir 1.2



You want to explore functional programming, but are put off by the academic feel (tell me about monads just one more time). You know you need concurrent applications, but also know these are almost impossible to get right. Meet Elixir, a functional, concurrent language built on the rock-solid Erlang VM. Elixir’s pragmatic syntax and built-in support for metaprogramming will make you productive and keep you interested for the long haul. This book is *the* introduction to Elixir for experienced programmers.

Dave Thomas
(352 pages) ISBN: 9781680501667. $38
https://pragprog.com/book/elixir12

Programming Phoenix



Don’t accept the compromise between fast and beautiful: you can have it all. Phoenix creator Chris McCord, Elixir creator José Valim, and award-winning author Bruce Tate walk you through building an application that’s fast and reliable. At every step, you’ll learn from the Phoenix creators not just what to do, but why. Packed with insider insights, this definitive guide will be your constant companion in your journey from Phoenix novice to expert, as you build the next generation of web applications.

Chris McCord, Bruce Tate, and José Valim
(298 pages) ISBN: 9781680501452. $34
https://pragprog.com/book/phoenix

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

This Book's Home Page
https://pragprog.com/book/jkdepj2
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
https://pragprog.com/updates
Be notified when updates and new books become available.

Join the Community
https://pragprog.com/community
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
https://pragprog.com/news
Check out the latest pragmatic developments, new titles and other offerings.

Buy the Book

If you liked this eBook, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: *https://pragprog.com/book/jkdepj2*

Contact Us

Online Orders:	*https://pragprog.com/catalog*
Customer Service:	*support@pragprog.com*
International Rights:	*translations@pragprog.com*
Academic Use:	*academic@pragprog.com*
Write for Us:	*http://write-for-us.pragprog.com*
Or Call:	+1 800-699-7764