Extracted from:

# Deploying with JRuby

Deliver Scalable Web Apps Using the JVM

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

# Deploying with JRuby

Deliver Scalable Web Apps Using the JVM

Joe Kutner

*Edited by Brian P. Hogan*

The Facets of Ruby Series

# 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 Mongrel and Apache. Having this kind of infrastructure means that 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.[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 are going to stay in business.

The problem I've just described is all too common. It has happened to everyone from small start-ups 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 are heterogeneous and decentralized in nature. Even worse, Rubyists have become comfortable with this way of doing things, and many think it is the only way of doing things. But that is 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 were not 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/defunkt/resque
2. http://daemons.rubyforge.org/
3. http://mmonit.com/monit/

behind a load balancer and putting any 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, we'll explore the most popular and well-supported methods for deploying JRuby. There is a surprising amount of flexibility in the processes and platforms that can be used, which allows Ruby and Java programmers to tailor their deployments to suit many different environments.

## What's in This Book?

Over the course of this book, we're going to rescue the application that was described at the beginning of the chapter. We'll do this by porting it to JRuby and deploying it in a way that will simplify its infrastructure and improve its ability to scale.

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

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 our systems can handle.

We'll begin rescuing Twitalytics in Chapter 1, *Getting Started with JRuby*, on page ?. We'll discuss what makes JRuby a better deployment platform and why we want to use it for our application. Then we'll port Twitalytics to JRuby and package it into an archive file with the Warbler gem. But before we can deploy it, we'll need to create an environment where it can run.

In Chapter 2, *Creating a Deployment Environment*, on page ?, we'll set up a virtual production server that will simulate a real deployment target. We'll provision it with the essential components of any production JRuby environment, which means these steps will apply not only to Twitalytics but to any JRuby deployment. You'll also learn how to automate this process to make it more reliable. We'll create a new server for each deployment strategy we use in this book, and the automated scripts will save us from having to re-create this environment each time.

Once we've completed the setup of our production server, we'll be ready to deploy. In Chapter 3, *Deploying an Archive File*, on page ?, we'll write a script that deploys the archive file we created earlier. You'll learn how this process differs from the more common practice of deploying a Ruby application as a directory of loose files. The script we'll write will be more portable than tools like Capistrano. We'll also deploy Twitalytics to the cloud with the CloudBees platform.

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

The next two chapters of the book will be dedicated to the lightweight Trinidad web server. Trinidad lets us deploy applications much like we would with MRI-based Rails applications using tools like Capistrano. But we'll find that JRuby allows us to reduce the complexity of this kind of deployment environment while increasing its reliability and portability. In Chapter 4, *Creating a Trinidad Application*, on page ?, we'll port not only the part of Twitalytics that handles web requests but also its background jobs to Trinidad. Then we'll set up our virtual server and deploy our application in Chapter 5, *Deploying a Trinidad Application*, on page ?. The resulting architecture will be friendly and familiar to Rubyists.

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

In Chapter 6, *Creating a TorqueBox Application*, on page ?, we'll introduce the concept of an application server. 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. We'll show how this eliminates the need for external processes and provides a centralized management interface. In Chapter 7, *Deploying a TorqueBox Application*, on page ?, we'll push to a production server running TorqueBox. But ultimately, we'll deploy our application to a TorqueBox cluster in Chapter 8, *Clustering a TorqueBox Application*, on page ?. This will give us the most advanced deployment environment available to any Ruby application.

An overview of each strategy covered in this book is listed in the following table:

| | Warbler | Trinidad | TorqueBox |
|---|---|---|---|
| **Built-in web server** | Winstone | Apache Tomcat | JBoss AS |
| **Archive file deployment** | WAR file | WAR file | Knob file |
| **Capistrano deployment** | No | Yes | Yes |
| **Background jobs** | No | Yes | Yes |
| **Clustering support** | 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 just part of the job. We also need to keep it running. To do that, we'll use some built-in JVM tools to inspect our new platform.

Chapter 9, *Managing a JRuby Deployment*, on page ? will present 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. Finally, Chapter 10, *Using a Continuous Integration Server*, on page ? will introduce a tool for producing reliable and consistent deployments.

Twitalytics is a Rails application, and we'll use this to our advantage as we deploy it. But all of the server technologies we'll use work equally well with any Rack-compliant framework (such as Sinatra[4] or Merb[5]). In fact, the steps we'll use to package and deploy Twitalytics would be identical for these other frameworks. Warbler, Trinidad, 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 are pigeonholed to work with Rails. In fact, Rails works with these servers because it is Rack-based.

## 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 are not familiar with how this new platform will change their infrastructure.

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

It is not required that you 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 is not required that you have any experience with Java or its associated technologies.

## The No-Java-Code Promise

You will not have to write any Java code as you work your way through this book. That's not what this book is about. It is about deploying Ruby applications on the JVM. The technologies and tools that you will be introduced to in this book hide the XML and Java code from you. As the TorqueBox developers like to say, "[They] write Java so you don't have to."[7]

You may want to include some Java code in your application. Or you may want to make calls to some Java libraries. That is entirely your choice. If you want to write your programs exclusively in Ruby and deploy them on the Java Virtual Machine—like 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.

## 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 be using notation from bash, the default shell on Mac OS X and many Linux distributions, so 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
```

The commands we'll use are mostly compatible between Windows and bash systems (such as cd and mkdir). In the cases where they are not compatible,

7. http://vimeo.com/27494052

the appropriate commands for both systems will be spelled out. One 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 is 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 7 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.

## Preparing Your Environment

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

- Java Development Kit (JDK) 6 (aka 1.6)
- JRuby 1.6.7
- Git 1.7
- Bundler 1.0

Java 7 was released in July 2011 and is supported by JRuby 1.6.7, but this newer version of the JVM is not readily available on all operating systems. To ensure the consistency of the steps in this book, we will use Java 6. However, you are encouraged to try Java 7 if your platform supports it.[8]

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.

8. http://www.engineyard.com/blog/2011/getting-started-with-jruby-and-java-7/

## Installing Java

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

```
$ sudo apt-get install openjdk-6-jdk
```

On Fedora, Oracle Linux, and Red Hat, the JVM can be install with the yum command, like this:

```
$ su -c "yum install java-1.6.0-openjdk"
```

For Mac OS X systems, Apple provides a JDK version 6, and versions of Mac OS X prior to 10.7 (Lion) ship with the JDK. If you are running Lion, you can install the JDK by opening the Java Preferences application under the /Applications/Utilities/ directory. The first time this program is opened, we'll see a dialog like the one in Figure 1, *Mac OS X prompt to install Java*, on page xii. Follow its instructions to install the Java runtime. If the dialog does not appear, then the JDK is already installed.

For Windows systems, we'll need to use the Oracle JDK. Download and run the binary installer from the Oracle website.[9] After the installation completes, we'll need to set the JAVA_HOME variable. (The exact path may vary).

```
C:\> SET JAVA_HOME="C:\Program Files\Java\jdk1.6.0_27"
```

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

```
$ java -version
java version "1.6.0_07"
Java(TM) SE Runtime Environment (build 1.6.0_07-b06-153)
Java HotSpot(TM) 64-Bit Server VM (build 1.6.0_07-b06-57, mixed mode)
```

Now that the JVM is ready, we can put JRuby on our machine.

## Installing JRuby

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 us to use the ruby and gem commands without putting the j character in front of them or prefixing every other command with the jruby -S command. RVM is compatible only with bash systems, which does not include Windows. Installing JRuby on Windows will be described in a moment, but if you are using a bash system, run this command to install RVM:

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

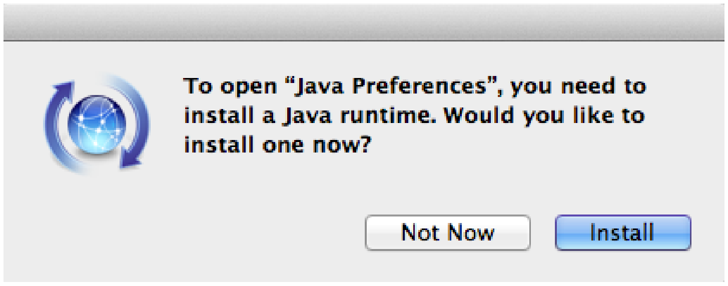
To open "Java Preferences", you need to install a Java runtime. Would you like to install one now?

Figure 1—Mac OS X prompt to install Java

```
$ bash < <( curl http://rvm.beginrescueend.com/releases/rvm-install-head )
```

You'll also have to reload your shell. The most dependable way to do this is to close the current terminal and open a new one. Now we can use RVM to install JRuby with this command:

```
$ rvm install jruby
jruby-1.6.7 - #fetching
jruby-1.6.7 - #extracted to ~/.rvm/src/jruby-1.6.7 (already extracted)
Building Nailgun
jruby-1.6.7 - #installing to ~/.rvm/rubies/jruby-1.6.7
jruby-1.6.7 - #importing default gemsets (/home/vagrant/.rvm/gemsets/)
Copying across included gems
Building native extensions. This could take a while...
Successfully installed jruby-launcher-1.0.12-java
1 gem installed
```

We'll also need to set JRuby as the default Ruby.

```
$ rvm --default use jruby
Using ~/.rvm/gems/jruby-1.6.7
```

If you are using a system that does not support RVM, such as Windows, then JRuby can be installed manually with these three steps:

1. Download the JRuby binaries from the official website.[10]
2. Unpack the downloaded file, which will create a jruby-<version> directory.
3. Add jruby-<version>/bin to the PATH.

Without RVM, we'll have to modify the commands that are used in this book. RVM allows us to invoke JRuby without using the jruby or jgem command, so we'll have to change all ruby commands in this book to jruby commands. We'll also need to prefix any other commands (such as bundle, gem, and rails) with the jruby -S prefix, like this:

10. http://jruby.org/download

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

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

```
$ ruby -v
jruby 1.6.7 (ruby-1.8.7p357) (2012-02-22 3e82bc8) ...
```

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

Next, we need to install Git.

## Installing Git

Git is a source control management tool that allows us to track versions of our code. We'll be using Git to switch between different versions of Twitalytics as we deploy it to new platforms. Follow the instructions for downloading and installing Git from the official website.[11]

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. The examples will even work without version control software, but that is not recommended. The source code for each branch we'll create is available from http://pragprog.com/titles/jkdepj/source_code, so instead of switching branches, you can change to the directory that corresponds to the chapter you're reading.

## Getting the Source Code

Now we're ready to set up the Twitalytics application. We'll start by downloading the source code from http://pragprog.com/titles/jkdepj/source_code. Unpack the downloaded file and put it in your home directory. This will create a code directory and inside of that will be a twitalytics directory, which contains the baseline code for the application (in other words, the MRI-based code).

We need to change directories into this location and initialize it as a Git repository.

```
$ cd twitalytics
$ git init
$ git add .
$ git commit -m "initial commit"
```

Next, we need to install Bundler, a dependency management tool for Ruby, by running the following command:

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

```
$ gem install bundler
```

Now we can use Bundler to install Twitalytics' dependencies by running this command:

```
$ bundle install --without production
```

We've added the `--without production` option to exclude the `pg` gem, which requires that PostgreSQL be installed. We'll take care of this later in the book by switching to some JRuby database adapters that are just as fast and don't rely on native code.

Our development environment is ready, but we won't be able to run Twitalytics with JRuby yet; it works only under MRI. We'll port it to JRuby in Chapter 1, *Getting Started with JRuby*, on page ?.

## 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.net community has forums[12] and numerous Java-related articles.

For JRuby-related problems, the official JRuby website[13] has links to several community outlets. The most useful of these are the mailing list[14] and the #jruby IRC channel on FreeNode.[15]

For Trinidad-related problems, there is a mailing list[16] and a wiki.[17]

For TorqueBox-related problems, there is a mailing list,[18] extensive documentation,[19] and the #torquebox IRC channel on FreeNode.

12. http://www.java.net/forum
13. http://jruby.org/community
14. http://xircles.codehaus.org/projects/jruby/lists
15. http://freenode.net/
16. http://groups.google.com/group/rails-trinidad
17. https://github.com/trinidad/trinidad/wiki
18. http://torquebox.org/community/mailing_lists/
19. http://torquebox.org/documentation/