Extracted from:

Deploying with JRuby 9k

Deliver Scalable Web Apps Using the JVM

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Deploying with JRuby 9k

Deliver Scalable Web Apps Using the JVM

Joe Kutner

Edited by Brian P. Hogan

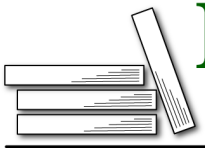
Deploying with JRuby 9k

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

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
```

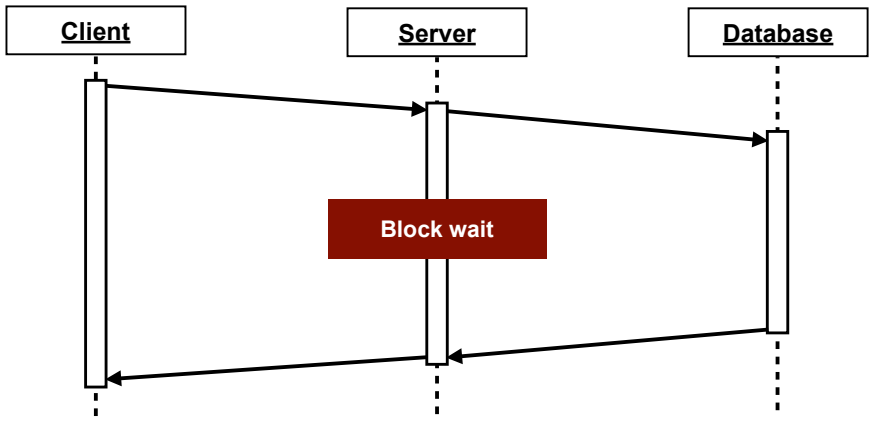
This generates a stock-service.war file, which you can execute by running this command:

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

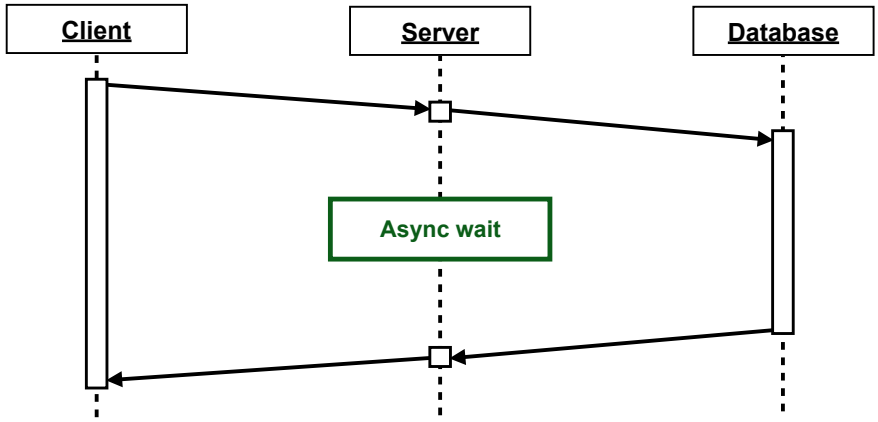
With the Java process running, test out the service by opening another terminal window and executing this command:

```
$ 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.

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/