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

1.1 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. This has led to some complex workarounds to achieve concurrency.

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 that run in separate processes. An example of this using Mongrel is illustrated in Figure 2, *Traditional MRI web application architecture*, on page 2. 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 is a lot of overhead in starting a new process. Several instances may end up fighting each other for resources if they are 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 we'll also need more memory for the server.

Several frameworks, such as Unicorn, Passenger, and Thin, have been created that try to improve upon this model. But they all suffer from the same underlying constraint. MRI cannot handle multiple requests in the same runtime concurrently. If you want to handle ten requests at the same time, then you need to have ten instances of your application running. No matter how you do it, deploying with MRI means managing lots of processes.

JRuby allows us to use a very similar model but with only one JVM process. Inside this JVM process is a single application instance that handles all of our website's traffic. This works by allowing the platform to create many threads that run against the same application instance in parallel. We can

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

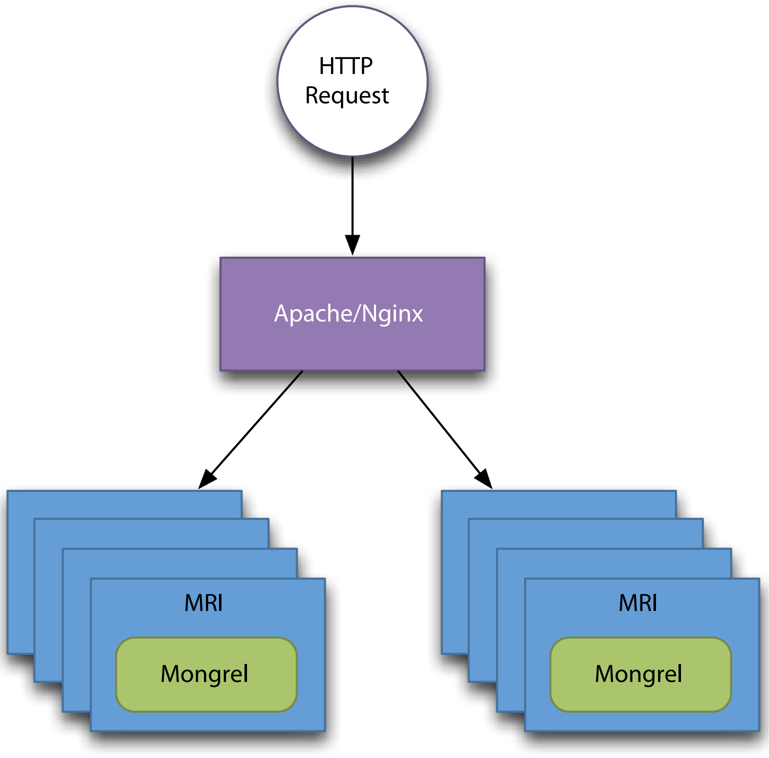

Figure 2— Traditional MRI web application architecture

create far more JVM threads than we could MRI processes because they are much lighter weight. This model is illustrated in Figure 3, *Architecture of a JRuby web application*, on page 3, and we can use it to serve many more concurrent requests than an MRI-based system.

We've included Apache in the architecture, but its role on a single instance is greatly reduced. We'll use it 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, we'll build an architecture like the one we've just described with each of the technologies we use. We'll start with Warbler, which will get us up and running quickly. Let's begin by using Warbler to package a simple Rack application.

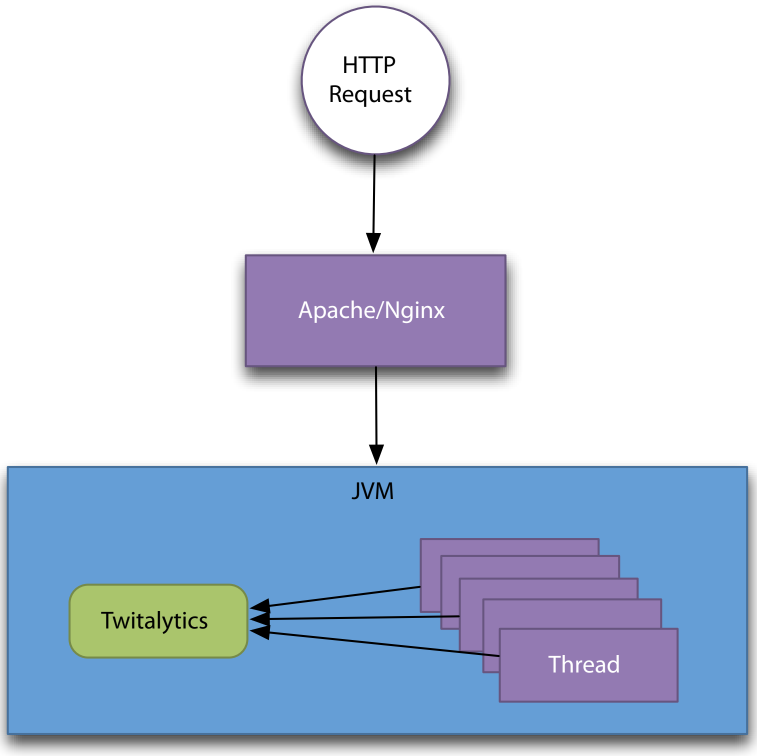

Figure 3—Architecture of a JRuby web application