CHAPTER 1

Introduction

JRuby on Rails is an exciting technology. If you've picked up this book, you've probably realized the same thing. You might not have much experience with either Ruby or Rails, or you've tried both of them out and want to see why the combination of JRuby on Rails is so spectacular. Regardless of the reason, I hope this book will teach you something about some of the technologies involved, introduce you to new ways to look at problems, and help you see solutions in the intersection of languages where each one isn't perfectly suited for a problem.

I've been using Java for a long time, but my heart has never been in it. I've always been a programming language nerd, trying out new languages like my girlfriend tries new shoes. I knew what was out there, and that Java wasn't the end-all solution for all the problems in the world. However, the fact remained that Java was the main language used for implementing systems during most of my employment. I compensated by continuing to have fun with other languages in my spare time. About three and a half years ago, I found Ruby. I can't exactly remember how I did that, but I started using it and liked it very much. It combined some of the more useful parts of Lisp metaprogrammability, with a Smalltalky sensibility and cleanliness, while still retaining much of Perl's pragmatism of doing whatever works.

It took me more than two years to convince my employer to start using Ruby. As much as I'd like to attribute that to the growth of my persuasion capabilities, the real reason was much more about the rise of Rails. At the time we decided to do a Rails spike, we faced a situation with resource and time limitations and needed to create a fairly simple database-backed web application. We finally convinced everyone to do this using Rails, which proved to be a clear win. Since then, more and more development is done in Rails, and right now about half the projects developed are Ruby on Rails projects instead of Java.

However, I still felt that something was wrong. As much as I liked Ruby and Rails, there were situations in which I felt it wasn't enough. In Java, I always felt constrained by the language. With Ruby, the situation was the inverse: the language was lovely, but important things were missing in the platform and ecosystem. In most cases this was caused by the relative newness of Ruby and Rails, but some of it came from features that make Java code more robust and well-performing.

That's when I started looking for a combination of the features I liked best from Ruby, while still retaining some of the good parts of the Java platform. I spent some time with different Lisp implementations on Java, ending up as a committer on the Jatha project (a Common Lisp implementation), but the Lisp implementations all shared the same problem: they didn't have Ruby's killer apps. I liked Jatha very much, but there wasn't enough community behind it, and not enough pressure to support major libraries.

So I continued my search, and I found JRuby. That was in fall 2005. At that time I really wanted to start using JRuby, and also to contribute to it, but I didn't have time. Cue three months later; Charles O. Nutter and Thomas Enebo had done some great work during that time, and it seemed obvious that JRuby would be able to run Rails sometime in the future. At that point I started helping out, contributing some smaller things, and later creating some of the more important extensions that JRuby absolutely needed to run major applications. YAML (standing for YAML Ain't Markup Language) was the key one that finally enabled us to start working on RubyGems and Rails support in earnest. Around that time I realized how powerful Rails and JRuby could be together. Now, one year later, we're successfully running almost any pure Ruby application. Rails applications usually work perfectly, and the full power of Java is also available to these applications. These applications can take full advantage of Java while retaining everything about the Ruby language.

It seems I've finally found a solution I can work with. The culmination is this book, describing what you can achieve by harnessing Ruby and Java together, creating useful Rails applications, and deploying them with tools that just aren't available when using the regular Ruby implementation.

In this first chapter, the focus will be on looking at the background behind Ruby, Rails, and JRuby; where they come from; and briefly what they are. I'll describe in more depth why JRuby on Rails is such a sweet match, and finally give a quick overview of the rest of the book, so you know what awaits you.

Ruby, JRuby, and Rails are exciting technologies. I love working with them; being involved with JRuby for the last 18 months has been the best choice I ever made. I hope I can share some of my enthusiasm—and the reasons for it—with this book, and that by the end of it you will feel some of it too. To me there's a profound difference in working in Ruby compared to Java. If you're a Java programmer, you might be skeptical about this proposition right now. However, the nice thing about JRuby is that it can act like a security blanket. You can do fun stuff with Ruby, but you'll also have Java available when you need it.

Let's get started with a quick introduction to the technologies we'll cover in this book.

Background

This book focuses on four technologies: Ruby, Rails, JRuby, and Java. The point of including Java on this list is that Java is what differentiates JRuby from Ruby. In fact, we won't look at much Java code in this book. The presence of Java should be felt, but it won't be obvious. The importance of Java is as a platform, enabling other technologies running on top of it.

I assume that you know enough about Java already. I'll quickly introduce Ruby, Rails, and JRuby, though, mostly from a historical perspective. The descriptions won't contain any direct language or API information; for that, refer to Appendix A for Ruby and Chapter 3 for Rails.

Both Ruby and JRuby have been around for much longer than you would expect. Ruby matches Java in age, and JRuby is more than six years old. In both cases their importance is fairly new, particularly because Ruby recently got a killer application in Rails, and JRuby hasn't been able to serve as a general interpreter for that long.

A Brief History of Ruby

Ruby was created in 1993 by Yukihiro "Matz" Matsumoto. It was first released to the public in 1995. The main implementation is an interpreter written in C, usually called Matz Ruby Implementation (MRI) when there's a need to distinguish between the Ruby language and the Ruby implementation.

Matz has repeatedly said that Ruby is designed for programmer productivity and fun. In many cases this is obvious from the Ruby code. Matz has also emphasized that Ruby tries hard to follow the principle of least surprise, meaning that the language should minimize confusion for experienced users. A nice side effect of this is that the basics of Ruby are easy to pick up. It's important to realize that the language design is focused on the human, not the machine. This means that features that don't perform well are a part of the language, just because it's worth more to give this capability to the programmer, rather than excluding it because it's hard to implement.

If you're interested in computer language terminology, it might interest you to know that Ruby is a reflective, dynamically and strongly typed, object-oriented, garbage collected language with support for many interesting language features such as continuations, green threads, coroutines, iterators, generators, closures, and metaprogramming. It draws primarily on features from Perl, Smalltalk, Python, Lisp, Dylan, and CLU.

A Brief History of Rails

Ruby on Rails is a web application framework that was first released in 2004. It was originally extracted from the application Basecamp, created by the company 37signals. The main creator of the language is David Heinemeier Hansson (usually called DHH). After its initial release it started to gain traction, and in time attracted a large following. In retrospect, Rails can be seen as the killer application for Ruby, spreading knowledge about Ruby and making it more popular to the masses.

Rails as a framework doesn't really contain anything new; what makes it special is that it combines several usage patterns and implementations of libraries in a productive way guided by some core philosophies. One of these is "Don't Repeat Yourself" (DRY)—meaning that information should be located in a single, obvious place. Even more important is "Convention over Configuration," which means that you need to do extremely small amounts of configuration and coding if your application follows the conventions of Rails. It also helps that the implementation extensively uses many of Ruby's metaprogramming features in a way that makes web development with Rails a pleasant thing.

Rails offers scaffolding and skeleton code created by code generators to speed up application development. That means that you're usually up and running with simple create, read, update, delete (CRUD) applications within minutes of first installing Rails. This gives you an opportunity to use a different and more agile structure for developing a system, because the feedback loop is short enough that the customer usually can take part from the beginning.

You can find more information about Rails in Chapter 3, which aims to tell you all you need to know about Rails before starting to develop with it.

A Brief History of JRuby

JRuby was originally a direct port of the Ruby 1.6 C code. It was created in 2001 by Jan Arne Petersen, and for a long time it only supported 1.6 semantics. After 1.8 was released, the maintainers introduced 1.8 features piecemeal over the course of two years. The final turning point came in the beginning of 2006, when the goal by project leads Thomas Enebo and Charles O. Nutter was set to be full compliance with Ruby; the acid test of this compliance would be running Rails unmodified. To set out on this project, many hours were devoted to creating better test suites and reworking large parts of the system.

It was obvious from early on in this endeavor that it wouldn't be a good solution to just port the C code straight off. There are large differences in the execution model of Java and C programs, which means that it would be hard to duplicate the C structures when better solutions were available in Java, and also that performance would be bad if trying to use an execution model that looked like MRIs.

After porting several important extensions to Java (YAML, ZLib, and other important parts of a Ruby system), both RubyGems and Rails started working. There were problems, but the support improved by leaps and bounds. The story got even better when Tom and Charles got hired by Sun Microsystems in September 2006 to work full time on JRuby.

Due to the great amount of tests that JRuby had, it was possible to do massive refactoring of the code base, change much of the internals, and be confident that if the test suite ran, the interpreter was good. The JRuby team has also been including test suites from other projects into JRuby. The more notable of these tests are many of the regular Ruby implementation tests, most of the tests from rubinius (another alternative implementation), and several application suites. JRuby runs a continuous integration server where the full Rails test suite is run, as well as RubyGems and Rake tests.

During much of this time, the core developers spent time looking at ways to compile Ruby code to Java bytecode. When 1.0 was released in June 2007, the runtime system by default ran in mixed mode, doing Just In Time (JIT) compilation of methods. The compiler wasn't completed at that point, handling about half of the syntactic construct of Ruby, but it gave a perceivable boost. At the time of the 1.0 release, JRuby performed close to MRI, being much closer in some cases, and slower in others.

Several things separate JRuby from MRI. The threading model is different, because JRuby uses real operating system threads, where MRI uses green threads (implemented by the Ruby interpreter and running within the same process).

JRuby doesn't support continuations. Continuations are one of those features that are incredibly hard to implement on a system running on a virtual machine, such as Java. Another reason for this decision is that it would be impossible to mix Java integration features with continuations. If this debate interests you, there are many posts in the archives of both the Jruby-dev and Ruby-core mailing lists.

There are also incompatibilities with how file system operations work, but in most cases these parts don't work well on Windows systems with MRI either.

There are currently plans to support an execution model that mimics the next big version of Ruby, called YARV (Yet Another Ruby VM). There is also talk of supporting rubinius execution.

Why JRuby on Rails?

Now you know why Ruby and Rails are interesting and exciting technologies. What's left to tell is why JRuby on Rails is different enough to warrant a book about it. I didn't mention in the introduction to Ruby that there are several problems with MRI. Many of these problems are caused by the flexibility of the language, and the fact that Matz has always focused on the language itself, not on its implementation.

The first problem is performance. In many cases, Ruby is fast enough and it works very well for many of its tasks. On the other hand, Ruby routinely finishes last in all language performance benchmarks. There's a common attitude that if you have performance problems in Ruby, you can always drop down to C and implement the critical parts there. Now, I love the Ruby language. That's why I want to use it. I don't want to drop down to C, and I especially don't want to drop down to C for the critical areas of my application. In fact, it should be the case that the critical parts are where I'll gain the most by using Ruby. However, that's not always possible today.

JRuby aims to fix that by focusing heavily on performance. The 1.0 release didn't have much performance work in it, and that shows. It's hard to measure general performance, but in most cases JRuby 1.0 seems to be about 1.5 to 2 times slower than Ruby 1.8.6. The raison d'être for 1.0 was compatibility. However, while working on the interpreter and compiler, the core team laid down the foundations to build on and improve performance. So, there seems to be no reason why JRuby can't be much faster than it currently is, and also much faster than the C implementation.

The second problem with the current Ruby implementation is that the support for Unicode and UTF-8 is spotty at best. To create applications connected to the Internet in 2007, you need to have fast, reliable, omnipresent Unicode support at the language level. Without it, you're lost. MRI does have some support for it, through something called KCode. However, this isn't at all pervasive; many string methods aren't KCode aware, and there are many problems with it. Application developers have resorted to creating libraries to handle these deficiencies; Rails has it in something called `Multibyte`.

Because JRuby runs on the Java platform, you can technically have access to all support Java offers for Unicode. At the moment you need to work with real Java `String`s to do this, but adding better language-level support for JRuby is one of the major priorities. It's also something that will be easy to put there, because it's already available natively. The first step towards this will be to implement a native back end to `Multibyte` and other similar libraries. When you read this, that should already have happened.

The fact that a Ruby program will never be able to take advantage of more than one core in your processor, due to it using green threads, is unacceptable in certain applications and merely inconvenient in others. JRuby solves this by having Ruby threads be based on real operating system threads instead. This causes some incompatibilities with MRI, but the general consensus is that it's worth it.

These are the major poster children for using JRuby instead of Ruby, and they apply equally well for running Rails on the platform. However, with regard to Rails there are a few more interesting opportunities and capabilities that the Java platform provides. First of all, Rails development is generally considered pleasant; Rails deployment isn't. There are many tools to help with this, but what it comes down to is that Rails development doesn't have the maturity that Java has. So, deploying Rails applications in JRuby is one selling point. (You'll see how to do so in Chapter 11.)

In many situations, an application needs to use several libraries for functionality. Ruby has been around for a long time, but the maturity of its libraries can't be compared to that of the Java platform. In some cases you'll have to write your own libraries for Ruby because no one has done what you've tried to do yet. That never happens with Java anymore. With Java, you usually have a good amount of libraries to use, and usually also commercial offerings. So, when developing a new application it can be highly useful to do it with JRuby on Rails, but it's also helpful to be able to fall back and use Java libraries from inside the application for certain functionality. JRuby makes it easy to do so.

There are more reasons to consider JRuby on Rails, and I'll touch on most of them in several places in the book.

Overview of the Book

This book is divided into four different parts, with some information before and after, and three appendixes. To help you get a feeling for how the book is laid out, I'll give a quick introduction to each chapter here. If you need specific information about a subject, please feel free to jump around. Keep in mind that most chapters use an overarching project for that part, which means that in some cases important context can be found in preceding chapters.

The four project parts are relatively separate from each other, but they each depend on things you learned in earlier chapters.

Chapter 1: Introduction

This is the introduction to the book, giving you information about the technologies covered, why they should interest you, and an overview of the book. You should be reading it right now.

Chapter 2: Getting Started

This chapter is aimed at getting you up to speed by helping you to install everything you need for the rest of the book, including all RubyGems you'll be using. The chapter also gives a small introduction to each of them, and tells you how to do basic tasks with the `gem` command.

Project 1: The Store (Shoplet)

The store application is the first Rails project you create, and as such won't differ much from what you would have done if you were developing the application with MRI. The big difference is that the system is backed by Java Database Connectivity (JDBC).

Chapter 3: Introduction to Rails

This chapter is a gentle, mostly non-coding introduction to Rails; it describes what parts it contains and things that are good to know when doing Rails development.

Chapter 4: Store Administration

Here you build the first half of the Shoplet application. The chapter introduces many of the more practical details of Rails in the process.

Chapter 5: A Database-Driven Shop

The Shoplet application gets finished and you take a look at the databases that JRuby on Rails supports.

Project 2: A Content Management System (CoMpoSe)

The second application isn't much larger than the first one; the difference is that it makes heavy use of some Java libraries to process Extensible Markup Language (XML) and handle content rendering.

Chapter 6: Java Integration

In this chapter we take our first detour and focus exclusively on the syntax and usage of JRuby's Java integration features.

Chapter 7: A Rails CMS

Using what we learned from the first project, we proceed to create most of the Rails code needed for the CMS application, but stub out all rendering functionality.

Chapter 8: Content Rendering

Using some of the Java integration features displayed in Chapter 6, this chapter completes the CMS application by adding all the rendering functionality and also taking a look at a few alternative approaches.

Project 3: An Administration System (BigBrother)

The BigBrother system is based on separating the Rails front end from an enterprise back end. It also has some features allowing it to be managed by Java Management Extensions (JMX).

Chapter 9: A JRuby Enterprise Bean

We look at how to use JRuby from inside a J2EE Enterprise Bean, implementing the functionality of this bean in Ruby.

Chapter 10: An EJB-Backed Rails Application

Most of the BigBrother application is completed by implementing a Rails front end that talks to an Enterprise JavaBean and also uses JMX to manage itself.

Chapter 11: Deployment

The next detour details deployment options for a JRuby on Rails application, how regular Rails deployment usually works, and how to make the situation much better with JRuby.

Project 4: A Library System (LibLib)

The final project is a distributed library system that shares a centralized data storage inside of the boundaries of a legacy system. The application uses messaging services to interact with other instances of the application, and also the legacy system.

Chapter 12: Web Services with JRuby

This chapter looks at the options available to consume web services with JRuby, and implements a library to search for books at Amazon.com.

Chapter 13: JRuby and Message-Oriented Systems

We take a deep dive into message-oriented middleware, looking at implementing both ends of such a system using JRuby and JMS. The chapter culminates in creating two different libraries for JMS interaction.

Chapter 14: The LibLib Rails Application

We create the final project application, using the libraries developed in Chapter 12 and 13 to provide some interesting library services.

Chapter 15: Coda: Next Steps

This chapter contains a few pointers as to what to do next and how to contribute to JRuby or its surrounding projects.

Appendix A: Ruby for Java Programmers

This appendix offers a short introduction to the Ruby language; it's aimed at Java programmers, but it should be digestible by anyone with programming experience.

Appendix B: JRuby Syntax

This appendix has a table detailing the Java integration features and other JRuby-specific APIs.

Appendix C: Resources

This appendix contains pointers to web pages, blogs, and posts that might be of further interest to you.

Summary

It's time to get started. I've talked in depth about what the book will cover and why these technologies are interesting, and might just transform your life. What's missing is the *how* of it; before we get into that, a short chapter will tell you how to install everything needed, and then it's time to start learning JRuby and Rails.