JRuby for Java Developers

Benefits and Quick-start Guide

Engine Yard, Inc. 866.518.9273 www.engineyard.com

Introduction

Chances are, you've probably heard of the Ruby programming language. Over the last few years, interest in Ruby has grown dramatically, largely thanks to the success of Ruby on Rails. Rails is a breath of fresh air in the web development world; and much of what makes Rails a success is the power and expressiveness available from the Ruby language.

Ruby is really two separate concepts: the language specification, and the platform implementation. The download available at ruby-lang.org, is the 'original' Ruby. It is is often called Ruby MRI, which stands for 'Matz's Ruby Interpreter', after the patriarch of Ruby, Yukihiro Matsumoto. You may also hear it referred to as CRuby, based on the fact that the platform is developed in C.

By contrast, JRuby is an implementation of the Ruby programming language powered by the Java platform, and today it is considered to be one of the most complete and mature implementations available.

Since JRuby is so mature, Ruby programs work just as well in JRuby as they do CRuby because the core API, standard libraries, and general execution characteristics of both languages are all compatible. At the time of this writing, JRuby is considered compliant with the 1.8.7 specification of Ruby, and is largely compatible with 1.9 (although some work remains). This paper will provide you with the following:

- A synopsis of why Java and Ruby are natural complements to each other
- 7 JRuby features that can help Java developers write better code faster
- Technical quick-start guides on getting started with both JRuby and Rails
- A summary of the JRuby ecosystem

By combining the capabilities of the Ruby programming language with the Java platform, you'll quickly see why JRuby is one of the most compelling software platforms available for Java developers today.

Synopsis: Java and JRuby

Before jumping in to JRuby, it's important to understand why the combination of Java and JRuby is so compelling: JRuby is in a unique position of benefiting from both the Java development world, and the Ruby development world at the same time. In the past decade, the Java Virtual Machine has grown to become a remarkably successful programming platform - there are a number of reasons for that success:

- The JVM is extremely portable: the Java VM has been implemented on nearly every platform in the enterprise world (from phones to mainframes and super-computers) and is known universally as being a stable and high-performance environment for software.
- The JVM has a first-class garbage collector that supports a large number of configuration options for running in nearly every conceivable condition.
- The Java virtual machine has superior management, instrumenting, and monitoring tools, all of which are built right in to the VM, and are made available through tools like VisualVM.
- Large corporations have invested significant time and money in Java support and interoperability, making it the platform of choice in a majority of situations.
- Java has a thriving open-source community.
- This success and enterprise dominance brings with it a number of reasons for an organization to continue to utilize Java technologies moving forward:
- There are Java application servers for every need, and the competition in this market does not show signs of slowing.
- There are Java libraries for nearly every task, from USB and Bluetooth, to distributed and enterprise cloud computing.

Because of the JVM's strong foundation, JRuby benefits from a top-notch platform with superior garbage collection, that is available on nearly every combination of operating system and hardware, and has enormous community and corporate backing.

While Ruby is a more recent entry, the growth in the past three years has been phenomenal. Like Java, Ruby has a vibrant and active community that emphasizes simplifying and maximizing development potential.

Importantly, a number of revolutionary technologies, like Ruby on Rails, have evolved out of the Ruby community, and have seen imitators in nearly every other arena. Plus, Ruby has an impressive array of libraries and APIs available, written with Ruby's strengths in mind.

For the most part, everything written for Ruby will just run in JRuby. There are some narrow exceptions where Ruby libraries choose to use native bindings, but even those are

becoming less common with the advent of Ruby FFI which allows for the binding of native libraries through Ruby code.

JRuby gives you the opportunity to choose the right tool for the job. You can code with JRuby to have the power and expressiveness of a dynamic language, or you can code in Java for performance-critical code sections.

7 Reasons Why JRuby = Productivity for Java Developers

To improve efficiency, all developers strive to become comfortable with the tools and libraries they use every day. So it's no surprise that considering the use of a complementary technology would raise some questions: Why would a Java developer want to consider JRuby over Java? What makes JRuby more productive? Below, you'll find an overview of seven features or JRuby that can help Java developers be more productive:

1) Ruby Syntax is Concise

Ruby syntax is brief, easy to read, and avoids nearly all repetition. Consider this simple Java example:

```
public class ExampleProgram {
  public static void main(String[] args) {
    int a = 5;
    int b = 7;
    System.out.println("a+b = " + (a+b));
    for(int i = 1; i<=10; i++) {
      System.out.println("Loop Index: " + i);
    }
  }
}
```

Listing 1: Simple Java Program

Now consider the equivalent Ruby counterpart:

```
a = 5
b = 7
puts "a+b = #{a+b}"
10.times do |i|
 puts "Loop Index: #{i}"
end
```

Listing 2: Simple Ruby Program

As you can see, the overall simplicity of Ruby's syntax makes the code easy to read,

write, and learn. Additionally, much of the required boilerplate in Java is noticeably absent, allowing developers to get up and running sooner.

2) Ruby is Dynamically-Typed

Java is a statically-typed language, this means the relationships between types are defined at compile-time, and are part of the class files. As a result, for one chunk of Java code to call a method on another chunk of Java code, the compiler must be aware of the class on which the method resides.

Unilke Java, Ruby code is dynamically-typed, meaning that the relationships between code chunks are instead uncovered during runtime as the code executes. This distinction has a major influence on the flexibility of Ruby, although it does trade some compile-time type-safety for that flexibility.

> **Note:** In fact, this addition of methods to '`String`' is exactly what Rails does to provide its naming-convention features: methods to pluralize, capitalize, camel-case, underscore and so forth are all added directly to String when you are running in the Rails environment.

Consider something as simple as the Java `Thread` class, which takes an object implementing '`java.lang.Runnable`'. This interface exists solely so the Thread class can, at compile time, be assured that the provided object implements the method '`run`'.

With dynamic typing, this sort of interface definition is no longer necessary. `Thread` would be allowed to call '`run`' on any object it was given, regardless of the object's type hierarchy, and no error would occur unless that object didn't, in fact, respond to calls to 'run'.

Another advantage of dynamic typing is that it reduces the amount of language syntax required for the average program, as variables do not require type definitions (as seen in Listing 2).

3) Ruby Supports Code Blocks

One feature that is noticeably absent from Java 6, but is popular in many other languages, is code blocks. In Java, when arbitrary code needs to be given to a particular library to execute, it typically involves constructing instances of a specific class or interface. The previous example of using '`java.lang.Runnable`' for implementing Threads illustrates this example, as do the various listener interfaces in Swing and AWT. In an effort to reduce the amount of required code and source files, often times these implementations are created using anonymous inner classes in Java, rather than creating full-fledged classes.

Unlike Java, Ruby doesn't require that the caller construct an instance of a class just to satisfy the compiler's contract. In other words, names are no longer necessary simply for the sake of invocation. This means developers who use Ruby can provide a simple code block that can be executed at the discretion of the target method.

Here is a short example, using Ruby's `Thread` class, which takes a block of code, as opposed to Java's `Thread` class, which takes a `Runnable`:

```
Thread.new do
 puts "Hello Ruby!"
end
```

Listing 3: Code blocks in Ruby

Here is the closest equivalent version in Java:

```
Thread t = new Thread(new Runnable() {
  public void run() {
    System.out.println("Hello Java!");
  }
});
t.start();
```

Listing 4: Anonymous Inner-Classes in Java

4) JRuby Integrates With Java

JRuby ships with a complete Java integration layer that allows you to work with Java APIs in your JRuby code as if they were Ruby libraries themselves. JRuby can access both the core Java libraries, as well as your own Java libraries - of which you undoubtedly have many. This provides you with the increased readability and dynamic power of Ruby all while using your existing Java libraries.

5) Ruby Supports Mixins

More and more languages today support the concept of mixins, sometimes also known as traits. Mixins are a common functionality that can be imported into multiple classes without the use of traditional object-oriented inheritance.

Having only single-inheritance, Java has no direct counterpart to mixins. The closest approximation would be interfaces, however those provide only a common API, not a common implementation.

Mixins offer the strengths of multiple-inheritance, without the weaknesses.

6) Ruby Classes Are Mutable

Ruby allows classes to be modified at runtime, providing you with an opportunity to streamline code that would otherwise be obscured by utility libraries.

As an example, many Java programs have additional functionality surrounding Strings, Dates, I/O, Collections, etc.; functionality that augments what is available in the core Java

libraries. This generally leads to lots of Java code referencing classes named `StringUtil`, `TimeUtil`, `CollectionUtil`, and so forth; each containing static methods that exist solely to enhance a core library that is frozen in time.

On the other hand, Ruby allows developers to contribute methods to any class. This meas that new methods can be called directly on objects of a given class, just as if they were part of the core library itself.

Other types of modifications that are possible with Ruby include: Removing methods, including modules, and adding and removing methods to individual instances (as opposed to the whole class).

7) Ruby Programs Have No Compile Step

Initially, the benefits of omitting the Java compilation step may seem to be fairly minor. After all, the actual cost of compilation on machines today is relatively low, especially when compared to the overall cost of building deployed resources, like JAR, WAR, and EAR files.

However, by omitting the compilation step, JRuby programs can easily be organized in a way that even when you're working under a heavy development pressure, there is no deploy step either. This is one of the many great features about Rails: The feedback cycle for Rails web developers is extremely short, because you can make changes and test without even restarting the your application, let alone redeploying - just refresh your page and try again.

JRuby and Runtime

When considering alternate languages, Java developers have a lot of important issues that need to be weighed. While productivity is always important, it's worth noting that even the most 'productive' language may not merit consideration if the constraints of your existing environment prevent its usage.

One of the predominant issues for most development shops is the runtime environment for a language, and of course, each programming language will have different requirements. As previously mentioned, CRuby is developed in C, therefore it must be compiled and installed for your platform before you can use it. Conversely, JRuby will work right away with your current Java infrastructure. JRuby is, after all, just Java - so for a Java business this means no 'secret sauce'; no new, unproven configuration or platform that requires your time and effort to install.

JRuby also integrates seamlessly with Java at runtime, allowing you to embed it in your application as a JAR file. This means that for most development, it can simply be treated

like any other dependency, and there is no additional platform to install - it just works. This malleability allows you to begin realizing the benefits of JRuby by introducing it into your application a piece at a time, without forcibly replacing entire sections of code.

Finally, JRuby is high-performance. The JRuby developers have put a lot of effort in trying to deliver the fastest Ruby implementation available. This performance means that you can be comfortable in moving highly-trafficked code into JRuby without the risk of impacting application performance.

JRuby Quick-start

If you haven't tried it yet, getting started with JRuby is both simple and straightforward. Packages are available for Windows as an executable installer, as well as all Unix variants (Mac, Linux, BSD) as a universal TAR or ZIP file. For most Java developers, the structure of the archive should be familiar, as it is similar to the download for many Java projects.

Like many Java programs, JRuby also has a very minimal install process: Simply place JRuby's 'bin' folder on the system PATH.

At this point, the JRuby command-line tools should be available. This can be verified by opening a new terminal and typing: 'jruby -v', which prints the current JRuby version. Assuming all went well, the output should be something similar to Listing 5 (at the time of this writing, JRuby 1.4.0 is the current build):

```
$jruby -v
jruby 1.4.0 (ruby 1.8.7 patchlevel 174) (2009-11-02 69fbfa3)
(Java HotSpot(TM) 64-Bit Server VM 1.6.0_15) [x86_64-java]
```

Listing 5: JRuby Version Output

The output of this command will vary depending on the build of JRuby, and the version of the JVM on which it is being executed.

Once JRuby is installed, you can start to tackle real development challenges faster than you would typically be expected to using only Java. For example, one typical problem in the enterprise world is traversing a file line by line while performing some form of transformation on the contents. To illustrate this, we will capitalize every line of a file, and output it to the console. Below you'll find a Java implementation of this program:

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
public class Capitalizer {
  public static void main(String[] args) throws IOException {
```

```
    BufferedReader r = null;
    try {
      r = new BufferedReader(new FileReader(args[0]));
      while(r.ready()) {
        System.out.println(r.readLine().toUpperCase());
      }
    }
    finally {
      if(r != null) {
        r.close();
      }
    }
  }
}
```

Listing 6: Capitalizer.java

This example simply reads the file denoted by the command line argument, and traverses the file using Java's I/O Reader libraries. For simplicity's sake, there is no explicit exception handling being done.

To implement this in Ruby, start by creating a new file: '`capitalizer.rb`' - this file can be put anywhere on your file-system because Ruby doesn't enforce the same the rigid package structure as Java. Also unlike Java, Ruby doesn't require a main method to be declared, so you can begin scripting the solution directly in the file, without any ceremony.

This example works with the Ruby '`File`' class to deal with file input and output.

```
File.open(ARGV[0], "r") do |input|
 input.each_line do |line|
  puts line.upcase
 end
end
```

Listing 7: capitalizer.rb

This program first opens the file, via a call to `File.open`, passing in the first argument to the script as the name of the file (denoted by `ARGV[0]`) and '`r`', (for 'read') which indicates an input stream is desired. A code block is then provided via the use of the `do` keyword. The code block assumes it will be given the open file stream parameter, which it has titled `input`.

The Ruby `File` class (like all Ruby I/O classes) supports resource-management, meaning that the stream parameter given to the provided code block is already configured and

available for use in the block. Additionally, once the code block completes, the file will automatically be closed and all resources will automatically be released.

The code block traverses the file stream by calling the 'each_line' method against it. Like the call to 'open', this method also accepts a code block. Also like the first code block, this one accepts a single parameter; this time called line. However, unlike the first example, this code block will be invoked multiple times - once for each line in the file.

For each line the code block receives, it simply prints the upper-case version of the line to the console, by calling upcase on the string, which is similar to Java's String#toUpperCase method.

Running this program is straightforward. Simply invoke the script with JRuby from the command line:

```
$jruby capitalizer.rb capitalizer.rb
FILE.OPEN(ARGV[0], "R") DO |INPUT|
 INPUT.EACH_LINE DO |LINE|
  PUTS LINE.UPCASE
 END
END
```

Listing 8: Output of running "capitalizer.rb" on itself

In this example, the capitalizer program is being run on the file 'capitalizer.rb' (on the source file itself), however any text file may be used.

Note that, as mentioned previously, there is no separate compilation step to perform first - JRuby simply interprets the program on first invocation.

Java Integration

The previous example utilized Ruby's built-in libraries to perform the actual file I/O, however, in some cases Ruby may not have a functional equivalent for something you want to do, or you may want to work with your existing Java code. No problem - JRuby scripts don't have to be limited to working with Ruby libraries, every Java library on the classpath is readily available as well.

A particularly popular use case for JRuby is developing GUIs using either Swing or SWT: Two very powerful Java widget toolkits. Using these libraries is simple, thanks to the Java integration features of JRuby. Let's start another Ruby script called 'gui.rb' to capture this work. To access the Swing libraries, the Ruby program must first import the Java integration support. This is done through the use of the standard Ruby 'require' keyword, providing 'java' as the library name:

```
require 'java'
```

Listing 9: gui.rb - Requiring Java support

The 'require' keyword is similar to Java's 'import' command; however, unlike 'import' in Java, 'require' is simply a function, which may be invoked by Ruby code at any time. This means that Ruby scripts can (and often do) import dependencies dynamically. Once Java support has been enabled, there are a number of ways to bring the relevant classes into the script's namespace; this example will use the simple 'java_import' function:

```
java_import 'javax.swing.JFrame'
java_import 'javax.swing.JButton'
java_import 'java.awt.event.ActionListener'
```

Listing 10: gui.rb - Importing Java classes

These imports tell JRuby to look for a Java class with the provided name, and make it available in the program's namespace through the integration layer. After importing the Java classes, they may be treated like any other Ruby class:

```
f = JFrame.new "My Application"
f.default_close_operation = JFrame::EXIT_ON_CLOSE
b = JButton.new "Test Button"
f.add b
f.pack
f.visible = true
```

Listing 11: gui.rb - Setting up a Swing GUI

This section of code creates a new `JFrame`, sets the default close operation, packs the frame, and requests it to become visible. Listing 11 shows how JRuby auto-converts Java methods to the Ruby coding-style. Ruby favors the use of underscores over camel-case, so JRuby automatically aliases fields and methods to use that format, although you can still use the original camel-case format if you prefer.

Additionally, because of the flexibility allowed developers in Ruby, traditional getters and setters as known in Java are unnecessary. As a result, JRuby aliases the getters and setters to work like traditional field reference and assignment.

These two auto-conversions are what enable `JFrame#setDefaultCloseOperation(...)` to become `JFrame#default_close_operation=...`, and `fr.setVisible(true)` to become `fr.visible=true`.

Here is some other Ruby-specific syntax differences to notice:

- The different constant syntax - The double-colon (::) denotes a class-level member in Ruby.

- The lack of parentheses - Simple, unambiguous method invocations (like the `JFrame` and `JButton` constructors) do not require parentheses in Ruby (although they can be provided if you prefer).
- The lack of variable types - Note that neither `f` nor `b` have any type definition. Ruby assumes you, the developer, will manage typing.

Now that you have a fully constructed frame with a button, it would be nice if the button could do something when it is clicked or pressed. This is typically achieved in Swing by calling `JButton#addActionListener(ActionListener)`, providing a listener implementation that does the actual work. In Java you would likely provide an anonymous inner-class as the implementation.

In JRuby you can achieve the same effect by using arbitrary code blocks, and the built-in proxy support for Java interfaces. To create a proxy of a Java interface from a code block, simply invoke the 'impl' method on the Java class, providing a block that accepts two parameters: the name of the method being invoked, and, for our example, the '`ActionEvent`' object normally provided to '`actionPerformed`' on the ActionListener:

```
b.add_action_listener do |evt|
 puts “The button was pushed!”
end
```

Listing 12: gui.rb - Adding functionality to the button

This code was added prior to making the frame visible. Running this from the command line should now result in an interactive Swing GUI, and pressing the button should invoke our Ruby code block, printing to the console.

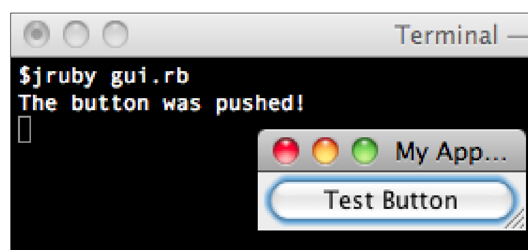

Figure 1: gui.rb in Action

JRuby's Java integration isn't limited to what is provided by JRuby for the existing library. At first it may seem unusual, but JRuby is able to add and modify methods on Java objects within the Ruby program, as if they were regular Ruby objects.

For example, perhaps there is something all buttons should be able to do in your application; something that is not part of the core Java API; such as, in our previous example, capitalize their text.

Using the previously discussed ability to change classes at runtime, you can simply add a method to `JButton`, as if it were a normal Ruby class. This block of code can be added after importing `JButton`, at the top of the file:

```
class JButton
 def swap_case
  if @original_text
   self.text = @original_text
   @original_text = nil
  else
   @original_text = self.text
   self.text = @original_text.upcase
  end
 end
end
```

Listing 13: gui.rb - Adding the method 'swap_case' to JButton

Note what appears to be a re-declaration of the `JButton` class. This simply tells Ruby that you desire to make additional changes to the `JButton` class already in the current namespace.

This method will be accessible to all subsequent Ruby code in the application, and the button will, as we asked, change case when called. Note the usage of the `@` symbol for the variable `@original_text`. This indicates that `original_text` is an instance variable of the class. Unlike Java, where instance variables must be explicitly declared at the top of the class, in JRuby instance variables are "declared" simply by being assigned the first time.

We can now modify our `JButton` action listener to toggle itself between upper-case and normal case:

```
b.add_action_listener do |me, evt|
 b.swap_case
end
```

Listing 14: gui.rb - The button calls 'swap_case' on itself.

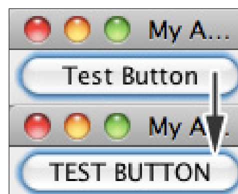

Figure 2: gui.rb Swapping Case

While this example utilizes Swing, and makes the code much more readable in the process, several members of the open-source community have already begun exploring the possibilities for JRuby to improve GUI development with Java. There are a number of solutions already available for both Swing and SWT development that take advantage of JRuby's best features, offering a lot of opportunities to leverage Ruby-like APIs. Here is a short list to consider:

- MonkeyBars - http://monkeybars.rubyforge.org (Swing)
- Profligacy - http://ihate.rubyforge.org/profligacy/ (Swing)
- Rubeus (http://code.google.com/p/rubeus/) (Swing)
- Glimmer - http://eclipse.org/glimmer/ (SWT)

Embedding Ruby Scripts in Java

Thus far, all of the JRuby code examples have been executed via a shell command. While this is simple, it may not offer the integration opportunities needed for a true production grade application, particularly if you intend to mix Java and Ruby.

To expand you integration options, it is possible to embed JRuby into your Java application. As of version 1.4, JRuby ships with a feature called Red Bridge, which is a comprehensive embedding API that provides several options for invoking and interacting with Ruby scripts from your Java program.

> **Note:** RubyGems is currently the preferred way to manage Ruby libraries. Gems have full support for version ranges, and can install gems from remote repositories (like RubyForge and Gemcutter). It is also possible to package gems for embedding in a Java application, although the details are outside the scope of this paper.

While there are a number of ways that you can invoke JRuby scripts, example in this document will demonstrate the JSR-223 scripting APIs, which are available in Java 6:

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.io.Reader;
import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;
import javax.script.ScriptException;
public class ScriptInvoker {
  public static void main(String[] args)
    throws IOException, ScriptException {
    ScriptEngineManager manager = new ScriptEngineManager();
```

```
      ScriptEngine engine = manager.getEngineByName("ruby");
      Reader r = null;
      try {
        r = new BufferedReader(new FileReader("gui.rb"));
        engine.eval(r);
      }
      finally {
        if(r != null) {
          r.close();
        }
      }
    }
  }
```

Listing 15: ScriptInvoker.java - Invoking Ruby Scripts from Java Programs

This code creates a new `ScriptEngineManager`, which can look up `ScriptEngine` implementations - in this case, we're looking for the implementation capable of parsing '`ruby`' scripts. The code then asks the `ScriptEngine` implementation to parse and execute the provided file. The only requirement for this to work is to have JRuby on the classpath.

The standard JRuby distribution ships with `jruby.jar`, and this can be used for embedding, but it only contains the core Ruby libraries. If you intend to use code that is in the standard libraries, you will need the `jruby-complete.jar` instead, which is available as an alternate download from JRuby.org.

Note: While Rails provides several scripts that help to generate code following the Rails conventions, they are entirely optional, and can be completely avoided in favor of manually creating the necessary resources.

When either of these JAR files are on the classpath, the `ScriptEngineManager` will automatically have a registered script engine for Ruby scripts, and the above code will work just as if you had invoked the script from the command line.

There are many other options available for sending parameters and receiving results from scripts, as well as caching scripts for additional performance. For more information on embedding, visit the Red Bridge documentation on the JRuby Wiki.

Enterprise JRuby Development

Ruby has gained much of its popularity thanks to the success of Ruby on Rails. Rails is a web application framework that takes advantage of Ruby's most powerful features. In Rails, you'll be excited to see how little code you actually need to implement a fully

functional web application. In the most general terms, this is because the combination of Ruby on Rails fundamentally supports your development process, as opposed to fighting it (for more information on this idea, check out some of the additional resources at the end of this paper in the 'Going Further' section).

Rails development, like all Ruby development, has a very low barrier to entry. Getting started is as simple as installing the Rails '`gem`',and running a few scripts to generate a rails project.

Installing the rails gem only takes a single command from the console:

```
$jruby -S gem install rails
Successfully installed activesupport-2.3.5
Successfully installed activerecord-2.3.5
Successfully installed actionpack-2.3.5
Successfully installed actionmailer-2.3.5
Successfully installed activeresource-2.3.5
Successfully installed rails-2.3.5
6 gems installed
```

Listing 16: Installing the Rails Gem

The '`-S`' parameter tells JRuby to look for the program to run on the `PATH` - in this case we're running '`gem`' - which is the RubyGems packaging application - and asking it to install the '`rails`' package. This proceeds to install Ruby on Rails, and all dependencies on the local development environment.

The other key step specific to JRuby that must be taken is to install an appropriate ActiveRecord JDBC adapter. ActiveRecord is the database mapping layer built in to Rails, and for JRuby, a JDBC adapter is required to route SQL through Java, rather than attempting to use native connectors. For this example, we will install the MySQL JDBC adapter:

```
$jruby -S gem install activerecord-jdbcmysql-adapter
Successfully installed activerecord-jdbcmysql-adapter-0.9.2
1 gem installed
```

Listing 17: Installing the ActiveRecord-JDBC MySQL Adapter

The JRuby team provides a number of ActiveRecord adapters that connect through JDBC to interact with the database. As of version 0.92 of ActiveRecord-JDBC, to get them working in Rails all you need to do is install the gem, then run a special script that installs the appropriate configuration code in the Rails application.

Before installing the adapter, we need to create our application. This is done by running

the `rails` application, giving it the application name you wish to use. These next few steps will show you how to use Rails to quickly build a web-based contact manager:

```
$jruby -S rails contact-manager -d mysql
create
create app/controllers
create app/helpers
create app/models
create app/views/layouts
(...)
```

Listing 18: Creating the 'contact-manager' Application

This creates an application in a new folder with the name `contact-manager`, and tells it to use a MySQL database configuration for the back-end. Now you can run the previously mentioned script to enable JDBC support for this project. First, move to the project folder (`cd contact-manager`), and then run the JDBC initializer script:

```
$ jruby script/generate jdbc
exists config/initializers
create config/initializers/jdbc.rb
exists lib/tasks
create lib/tasks/jdbc.rake
```

Listing 19: Initializing JDBC Support

From here on out, the instructions for development are not specific to JRuby - the same steps are equally relevant to CRuby as well.

To function fully, your Rails application will need to know how to access the local MySQL database. Connection information is stored in the config/database.yml file. This file will have username and password properties that must be changed to the appropriate values for your MySQL install.

Once configured - it is simply a matter of creating the database schema via this command: '`jruby -S rake db:create`'.

Assuming no errors, Rails should have created a new MySQL database named `contact-manager_development`, as well as `_test` and `_production` schemas to match.

Because Rails is a model-view-controller framework, it is designed with those distinct components in mind. By default, these components are connected using naming conventions that allow Rails to find them automatically, without any configuration. One of the most common ways to quickly get up and running with Rails is to use the scripts which auto-generate code following these conventions.

There are Rails scripts for generating everything from individual components such as models, views, and controllers to scripts that generate complete, working functionality called scaffolding. For this example, the scaffolding generator will be used, as it provides the most functionality, the fastest:

```
$jruby script/generate scaffold Contact first_name:string last_name:string
exists app/models/
exists app/controllers/
exists app/helpers/
   (...)
```

Listing 20: Generating Scaffolding

In the example above, we are asking the Rails generate script to create scaffolding for the model 'Contact', with two properties initially: first_name and last_name.

This process creates a number of files automatically - of particular note in this example are the controller (app/controllers/contacts_controller.rb), the model (app/models/contact.rb), and the various view files (app/views/contacts/).

If you look at the model contact.rb file, you'll see that a very basic class was created:

```
class Contact < ActiveRecord::Base
end
```

Listing 21: The Contact model class

While it was mentioned earlier that Ruby is dynamically-typed, in the examples above you are required to specify that these properties are string. This may seem counter-intuitive, but the reason is simple - MySQL is not dynamically typed, therefore, Rails needs to know the types of fields to create the correct database columns.

New syntax in this example includes the < character, which denotes inheritence - in this case, Contact is inheriting from the Rails ActiveRecord::Base class. The properties (first_name, and last_name) are dynamically created by the super-class, and as such are not listed here.

One additional block of code generated by the scaffolding is a special database migration - this will create the table in MySQL for storing the new Contact object.

The database migrations can be run using the following command:

```
$jruby -S rake db:migrate
== CreateContacts: migrating ===================================
-- create_table(:contacts)
  -> 0.2870s
```

```
  -> 0 rows
== CreateContacts: migrated (0.2880s) =========================
```

Listing 22: Migrating the database

Everything is now in place to try this `Contact` class out, and it isn't even necessary to start the web application. To see the model in action, you only have to open the Rails console and write live code against the application model, creating and saving a new instance.

```
$jruby script/console
Loading development environment (Rails 2.3.5)
>> c = Contact.new
=> #<Contact id: nil, first_name: nil, last_name: nil, created_at: nil,
updated_at: nil>
>> c.first_name = "Charles"
=> "Charles"
>> c.last_name = "Nutter"
=> "Nutter"
>> c.save
=> true
>> c.id
=> 1
```

Listing 23: Working with Contacts in the Rails Console

In the above, note the standard use of the 'new' constructor method, and the ability to work with the properties as simple fields. You should also note that once save is called, the object is given an ID. At this point, your model is now saved to the MySQL database, and has been assigned a primary key value of 1.

Now that your Rails application is configured and the model and database is working correctly, it's time to try the application by starting the server. Doing so simply requires the execution of another script:

```
$jruby script/server
=> Booting WEBrick
=> Rails 2.3.5 application starting on http://0.0.0.0:3000
=> Call with -d to detach
=> Ctrl-C to shutdown server
[2009-12-07 23:23:41] INFO WEBrick 1.3.1
[2009-12-07 23:23:41] INFO ruby 1.8.7 (2009-11-02) [java]
[2009-12-07 23:23:41] INFO WEBrick::HTTPServer#start: pid=22144 port=3000
```

Listing 24: The WEBrick Server Startup

Once this script reports that the server has been started, the application can be reached by going to `http://localhost:3000/contacts`; the 'contacts' portion of the URL is automatically assigned by the Rails routing code. The application should have a fully functioning UI for maintaining contact objects.

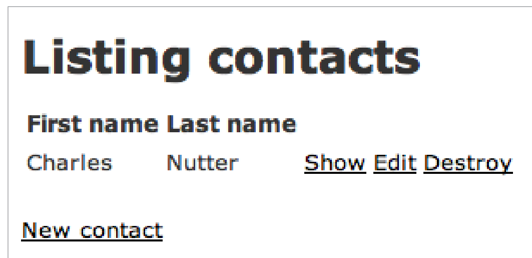

Figure 3: The Rails contact-manager Application

Note that the contact you previously added through the console is visible here, and that the Rails `contact-manager` application is now up and running.

There are several pieces of functionality automatically created by the Rails generator, including the ability to add, edit, and delete contacts. Further information regarding how all of the Rails components inter-connect is available in the Rails Guides.

This basic server works fine for development purposes, however at some point, you'll probably want to switch to a production-ready server. With JRuby it is preferred to use a server like GlassFish, which is a highly-scalable Java application server.

All that is required to get started with GlassFish and Rails is to install the 'glassfish' gem via: `jruby -S gem install glassfish`.

Once installed, just like with starting the server above, it is possible to run the gem in the root of the application folder:

```
$jruby -S glassfish
Starting GlassFish server at: 10.0.1.2:3000 in development environment...
Writing log messages to: contact-manager/log/development.log.
Press Ctrl+C to stop.
```

Listing 25: The GlassFish Server Startup

Just as with the basic server script, this should make the application available, ready to accept requests (still on port 3000), and `Ctrl+C` can be used to stop it when complete. Also just like the WEBrick server, files can be changed freely and the changes will become live immediately.

Finally, it should be noted that Rails applications can always easily be built into a standard Java WAR, by using the Warbler gem, allowing them to be deployed against any Java application server.

Your JRuby installation is now running on Rails!

Going Further

There are a number of resources available for Ruby, Rails, and JRuby on the internet:

- Ruby-lang.org is the official Ruby programming language site, and has significant documentation about the Ruby programming language. Most of this documentation is equally applicable to all Ruby implementations.
- Ruby-Doc.org is a community-driven site that serves to document the core APIs and Ruby standard library.
- The Ruby on Rails Guides are an excellent resource for starting Rails programming, and aside from the few exceptions detailed in this document, all of the Rails development is identical in JRuby.
- The JRuby Wiki has a wealth of information about JRuby, including links to several other documentation sources. There is more elaborate documentation on the Java/Ruby bridges, the best ways to distribute JRuby applications, and a number of other JRuby-specific topics.
- The JRuby Team-Member Blogs are regularly updated by the various committers, and provide news and unique insight for JRuby.
- The Engine Yard blog is frequently updated with technical articles, often written by committers for projects central to the Ruby ecosystem, like JRuby and Ruby on Rails.
- IsItJRuby.com is a community-powered database for capturing which Ruby gems are compatible with JRuby.

The JRuby Ecosystem

Momentum

Given the success of Java and the current excitement around Ruby, it's little surprise that JRuby is seeing a rapid growth in adoption. There are currently nine committers to JRuby. Charles Nutter and Tom Enebo, the co-team-leads for JRuby, as well as Nick Sieger are all currently employed full-time to do development on JRuby by Engine Yard: a major Ruby cloud provider. For the last release of JRuby (1.4.0), the team of committers fixed over 300 bugs, moved from 1.8.6 to 1.8.7 Ruby compatibility, and shipped several large new features.

2009 also saw the first official JRuby conference, JRubyConf, which came on the tails of RubyConf, covering topics specific to running Ruby applications in Java. The conference was immensely successful, and appears to be a likely repeat in 2010.

Tool Support

All of the major Java IDEs are now shipping Ruby development support, and each has first-class support for JRuby:

- Eclipse provides Ruby support with the Eclipse Dynamic Languages Toolkit. Alternatively, Aptana offers a product called RadRails which provides comprehensive Rails development support.
- NetBeans has full Ruby and Rails support out of the box.
- JetBrains IntelliJ IDEA has full support in the ultimate edition. Alternatively, JetBrains also offers RubyMine, which is a stand-alone Ruby IDE.

Enterprise Support

The Java enterprise is embracing JRuby as well, with a number of enterprise products shipping Ruby support through JRuby. As demonstrated earlier, Glassfish support is instantly available through RubyGems, and can be used to run high-scalability Rails applications "the Ruby way". The GlassFish website details the various facets of this support.

Even if custom integration isn't available for your application server of choice, with tools like Warbler, it's easy to install JRuby applications on the same infrastructure that all enterprise Java shops already use today.

Deployment

Finally, there are a number of commercial options for deploying JRuby:

Engine Yard currently offers JRuby support on their private cloud. Engine Yard cloud services make it extremely simple to develop, deploy, and scale Ruby on Rails applications.

JRuby can now run on Google App Engine. (Ref 1, Ref 2)

Of course, nearly any hosting provider can also be used for JRuby hosting, provided you use Warbler for deploys, or are willing and able to perform a lot of sys-admin and maintenance work.

Conclusion

Ruby and Rails provides developers a number of expressive and powerful features, and JRuby extends these capabilities to work within the Java platform that so many developers

are using today. By bringing the advanced capabilities of Ruby on Rails to Java with JRuby, you can can combine the best of both languages to help you reach your development goals.

About Engine Yard

Engine Yard provides automation technologies and services for Ruby on Rails, including Engine Yard Cloud, an application services platform for web developers and web teams. It provides easy-to-use, automated Rails application deployment and management, with a design philosophy that allows easy migration of existing applications. Engine Yard employs top industry experts and sponsors or contributes to many Open Source projects and efforts such as Ruby on Rails (www.rubyonrails.com), JRuby (www.jruby.org) and Rubinius (www.rubini.us). Headquartered in San Francisco, Calif., Engine Yard is backed by Benchmark Capital, New Enterprise Associates, and Amazon.com. Visit www.engineyard.com to learn more.

Contact us:
Engine Yard
500 Third Street, Suite 510
San Francisco, CA 94107
(866) 518-YARD (9273)