Extracted from:

Using JRuby

Bringing Ruby to Java

The Pragmatic Programmers

Using JRuby

Bringing Ruby to Java

Charles O Nutter,
Nick Sieger,
Thomas Enebo,
Ola Bini, and
Ian Dees

Forewords by
Yukihiro Matsumoto
and Bruce Tate

Edited by Jacquelyn Carter

The Facets of Ruby Series

```
$ ./bin/historian7 PASTA NOODLES
ruby_from_java/historian/lib/git/lib.rb:700:in `command':
git diff "-p" "PASTA" "NOODLES"  2>&1:fatal: ambiguous argument 'PASTA':
unknown revision or path not in the working tree. (Git::GitExecuteError)
Use '--' to separate paths from revisions
        from ruby_from_java/historian/lib/git/lib.rb:249:in `diff_full'
        from ruby_from_java/historian/lib/git/diff.rb:100:in `cache_full'
        from ruby_from_java/historian/lib/git/diff.rb:106:in `process_full'
        from ruby_from_java/historian/lib/git/diff.rb:64:in `each'
        from ruby_from_java/historian/lib/archive7.rb:10:in `history'
        from <script>:1
Couldn't generate diff; please see the log file.
```

So, there you have it: a program written in Java that calls a Ruby method to inspect the source code of…the program itself. We will be covering some more details for the rest of this chapter, but you largely have all the skills you need now. Go forth and make some simple embedded Ruby applications, or read on for the nitty-gritty details.

3.2 The Nitty-Gritty

There are always special circumstances and strange little details that a project runs into. If you find yourself wanting more control knobs for the embedding API than we've shown you so far, then read on.

Other Embedding Frameworks

All the examples we've seen so far have used Embed Core, the main embedding API that ships with JRuby. This API offers a great deal of interoperability. You can call a Ruby method, crunch the results in Java, and hand data back into Ruby. What makes this deep integration possible is that Embed Core was created just for JRuby.

There are times, however, when a general scripting API is a better fit than a Ruby-specific one. For instance, if your Java project already includes other scripting languages, you probably don't want to use a separate API for each language.

JRuby supports the two most popular Java embedding APIs. Bean Scripting Framework, the older of the two, began at IBM and is now hosted by the Apache Jakarta project. javax.scripting, also known as JSR 223, is part of the official JDK. Both have a similar flavor: you connect a general-purpose script manager to a language-specific scripting engine.

In case you're curious, here's how the final Historian example from earlier would look in JSR 223, minus the exception code. First, the imports at the top need to change a little:

Download ruby_from_java/historian/src/book/embed/Historian8.java

```
package book.embed;

import java.lang.NoSuchMethodException;

import java.util.List;

import javax.script.Invocable;
import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;
import javax.script.ScriptException;
```

Now for the Ruby embedding code:

Download ruby_from_java/historian/src/book/embed/Historian8.java

```
public static void main(String[] args)
    throws ScriptException, NoSuchMethodException {

    ScriptEngineManager manager = new ScriptEngineManager();
    ScriptEngine engine = manager.getEngineByName("jruby");
    Invocable invocable = (Invocable)engine;

    engine.eval("$LOAD_PATH << 'lib'");
    engine.eval("require 'archive8'");

    Object archive = engine.eval("Archive.new");

    List<GitDiff> diffs = (List<GitDiff>)
        invocable.invokeMethod(archive,
                               "history",
                               new Revisions(args[0], args[1]));

    for (GitDiff diff : diffs) {
        System.out.println("FILE: " + diff.getPath());
        System.out.println(diff.getPatch());
    }
}
```

JSR 223 is able to perform the same tasks for Historian that Embed Core does, in a slightly less expressive notation. BSF has a similar feel to what you saw previously, so we won't show a detailed example for it. Instead, we recommend you use JSR 223 for non-Ruby-specific embedding projects, because of its official position as part of the JDK.

Containers and Contexts

Each ScriptingContainer object that you create for embedding Ruby code has an associated *context* object, which JRuby uses for internal bookkeeping. By "bookkeeping," we mean things like the Ruby interpreter instance, I/O streams, a special variable store, and configuration options.

The simplest ScriptingContainer constructor creates a context implicitly for you. In case you want a little more control, you can specify the kind of context you want:

```
new ScriptingContainer(); // defaults to SINGLETON

new ScriptingContainer(LocalContextScope.SINGLETON);
new ScriptingContainer(LocalContextScope.THREADSAFE);
new ScriptingContainer(LocalContextScope.SINGLETHREAD);
```

Singleton

SINGLETON, the default choice, creates one Ruby runtime shared by the entire JVM. No matter how many ScriptingContainers you create, they'll all share the same context if you use this option. You can either specify this type explicitly or use the no-argument form of the constructor.

Singleton contexts are simple to use, because you don't have to pass ScriptingContainer references all around your program. But they also have a big drawback: they're not thread-safe. Try to run two chunks of Ruby code in different Java threads, and...kaboom!

Thread-Safe

If you know multiple threads will be accessing the same ScriptingContainer (or if you're just feeling paranoid), then you should use a THREADSAFE context. This type synchronizes all access to the Ruby runtime so that multiple threads can safely call into it without crashing.

This mode is certainly safer than SINGLETON, but it doesn't automatically make your concurrency problems go away. Under a heavy load, you may end up with a lot of waiting threads. It's even possible run into a deadlock situation. For instance, if an embedded script returns a Ruby object that, in turn, calls back into the embedding API, you can end up with a call that never returns. Fortunately, this is a bit of an extreme case. Just keep in mind the hazards of multithreaded programs as you're writing your code.[8]

8. For more information on what some of these hazards are, see Ousterhout's "Why Threads Are a Bad Idea (for most purposes)" at http://home.pacbell.net/ouster/threads.pdf.

Tom Says. . .

What Type of Context Should You Use?

Even though it's a bit of extra work up front, I recommend starting your project off with THREADSAFE containers. This keeps you in the habit of passing around the ScriptingContainer reference, in case you later decide to switch to using to one of the other two modes. It also makes it harder to accidentally kill your Ruby runtime.

Single-Threaded

So, the first mode guaranteed a single Ruby runtime, and the second introduced some thread safety. The third mode does...none of the above. Each time you create a ScriptingContainer with the SINGLETHREAD option, you actually create a brand new context. This new context is completely unconcerned with concurrent access. Everything rides on you, the programmer, to access the container from one thread at a time.

In truth, this kind of context is not such a dangerous beast if used in a controlled environment. For example, if you are running a servlet that spins up multiple threads, you can safely spawn one SINGLETHREAD-ed ScriptingContainer per servlet thread in Servlet.init(). Some configurations of the jruby-rack project use this strategy.

Ruby Version

JRuby supports both Ruby 1.8 and Ruby 1.9 syntax and semantics. By default, a new ScriptingContainer uses Ruby 1.8 mode, but it's quite easy to use 1.9 instead:

```
container.setCompatVersion(org.jruby.CompatVersion.RUBY1_9);
```

Compile Mode

We hesitate even to bring up this option but have decided to give it a passing mention, in case you encounter it in the wild or in documentation. In practice, we strongly recommend leaving it at the default setting.

The compile mode determines when, if ever, your ScriptingContainer object compiles individual Ruby methods down to JVM bytecode. It's tempting to set this option to force, meaning "always compile." After all, compiling just *sounds* faster, doesn't it?

Of course, real life is never so simple. The act of compilation takes time, so it only makes sense to compile a Ruby method if it's going to be called often enough for the time savings (if any!) to outweigh the initial delay. That's exactly what the default option, jit, tries to do.[9] There are times when compiling Ruby code makes sense but not when you're embedding a JRuby runtime in a Java project.

There are a few more options beyond these basic ones. You can control how an embedded JRuby runtime finds Ruby code, how it finds Java classes, how local variables are remembered from one invocation to the next, and more. Our goal, however, isn't to present a laundry list of every possible setting but to show you the ones you're most likely to encounter in the real world. For the rest, you may want to have a peek at the reference documentation.[10]

3.3 Embedding Strategies

In our Historian example, we saw several different ways to stitch the Java and Ruby sides together. You can pass a Java class into your Ruby script, make a Ruby class that implements/extends a Java type, or just use simple, coercible types such as strings.

There is no single best approach that applies in all situations. This section will break down some of the reasons why you may consider picking one strategy over another.

Passing Java Data Into Ruby

How do you get data into your embedded Ruby script? Passing in a Java object is the easiest approach. The embedded script can call the object's methods just as if they were written in Ruby. You can even decorate the object with additional, easier-to-use methods that actually *are* written in Ruby.

When is passing data into Ruby as plain Java objects *not* a good fit? It depends on how often the Ruby script ends up calling back into Java.

9. http://www.realjenius.com/2009/10/06/distilling-jruby-the-jit-compiler/
10. http://wiki.jruby.org/RedBridge#Configurations

Calling from Ruby to Java is a little slower than staying inside the Ruby universe. In many cases the difference is unnoticeable, but in others, the type coercion cost (for example, copying a java.lang.String to a Ruby String) makes this approach too slow.

So if your Ruby code needs to call a string-returning Java method in a tight loop, consider reshaping your solution a bit. Perhaps the Java side could assemble a Ruby object with the data preconverted and pass that in instead. Or you could move that time-sensitive loop into your Java code.

We don't mean to scare you away from the direct approach. Start out by passing a Java object into Ruby. If this doesn't meet your performance goals, *then* measure and rework.

Returning Data to Java

Getting data back into Java-land is a little more involved; Java knows less about JRuby than JRuby knows about Java. In general, there are three options:[11]

1. Return a Ruby object that implements a Java interface.
2. Return a Ruby object that extends a Java class (concrete or abstract).
3. Construct a Java object in Ruby and return it.

Options 1 and 2 are similar, in that you are returning a Ruby object that is tied to the JRuby runtime it came from. If your Java code calls methods on the object, these invocations will land back in the same JRuby runtime.

As we saw in Section 3.2, *Containers and Contexts*, on page 73, this reuse of runtimes can have interesting consequences for multithreaded Java programs. If you are passing objects between threads without using THREADSAFE mode, you can crash the Ruby runtime.

Option 3 is much less prone to threading issues than the other two choices. It can also be slightly faster, since you're not dispatching function calls from one language to another.

The obvious downside is inelegance. If you have a small, clean Ruby script, then the extra step of constructing a Java class for the sole

11. Technically, there's a fourth option: calling become_java! on a Ruby class. But we don't recommend it.

purpose of returning results will feel like makework.[12] If, on the other hand, you can build a simple Java class that doesn't look too out of place alongside your Ruby code, then go for it.

Type Coercion Pitfalls

JRuby strives to do the right thing with type coercions. As you call into Ruby code, and as that Ruby code returns data back to Java, many types will get implicitly converted to similar types in the other language.

This approach is not, however, immune to mishaps. Once an object is coerced to another type, no matter how similar, it really is a different object. Code that relies on object identity will not work right. For example, Maps may not work as you expect.

We've discussed a lot of "doom and gloom" scenarios in this section. While these are important to keep in mind, remember that, for the most part, things will just work. If you go about your project armed with the knowledge of which subtleties can bite you and what to do about them, you'll be fine.

3.4 Wrapping Up

In this chapter, we looked at the various ways to call from Java into Ruby, all in the context of a real-life example. We then highlighted a couple of specific features of JRuby embedding that may help you in your own projects. Finally, we zoomed out to discuss the general tradeoffs among embedding approaches.

We hope this discussion has whetted your appetite to introduce Ruby into your Java project. In the next chapter, we're going to take the next logical step and compile Ruby programs down to JVM bytecode.

12. Anyone remember the original EJB specification?