At the Forge

Getting Started with Ruby

Reuven M. Lerner

Abstract

What's behind all the Ruby hype? Reuven walks us through a couple of examples to let the code speak for itself.

About ten years ago, back when I was working in New York, friends of mine showed me something that knocked my socks off—a program that actually ran inside of the Web browser, without any need for pressing submit. It was sleek, fun to use and seemed like a major paradigm shift. We all were excited about what this new “Java” language and its applets would mean for Web development. Although we didn't quite know where or how it would end, we talked about nothing else for some time.

In the decade since then, many different technologies have been hyped as “the next best thing” or “the tool you need to make better Web sites”. Indeed, we constantly are bombarded with claims of newer, better, faster and cheaper ways to develop software. Some of these promises have panned out, but a trade-off usually is associated with them. For example, developing Web applications in Zope is indeed quite easy—one you get over the learning curve. Web services are fine, until you start to deal with complex data structures across different platforms.

You can imagine my surprise, then, when I began to see another “best new method” coming over the horizon—but this one was touted by people I respect, who normally don't give in to hype so quickly. I'm speaking, of course, about “Ruby on Rails”, an object-oriented system for creating and deploying Web applications. For several months now, I have been reading about how wonderful Rails is and how it makes Web development utterly simple.

I had been meaning to try Ruby as a language for some time, and the growth of Rails has given me an opportunity to do so. This month, we take an initial look at Ruby, examining simple ways to create Web applications with the basic Ruby language and libraries. In my next article, we will look at Rails and see how it stacks up against other, more established frameworks.

What Is Ruby?

Ruby is an open-source programming language originally developed by Japanese programmer Yukihiro Matsumoto, also known as Matz. Ruby first was released in late 1995, making it older than many people might think. It took some time for people outside of Japan to discover and work with Ruby, in part because of the lack of documentation. The first edition of Dave Thomas' book, Programming Ruby (see the on-line Resources) provided a solid introduction to the language, as well as a reference guide to its class libraries, giving it a needed PR boost. The second version of the “Pickaxe book”, as it is known, now is available.

Ruby was designed to be an “object-oriented scripting language”, and it indeed feels like a cross between Perl and Smalltalk. It assumes that you understand object-oriented programming and probably is not a good first language for someone to learn. But if you are familiar with both objects and Perl, then you quickly can
learn to do many things with Ruby.

Here is a simple “Hello, world” program in Ruby:

```ruby
#!/usr/bin/env ruby
print "Hello, world\n"
```

The first line ensures that we run the Ruby interpreter, regardless of where it might be in our path. The second line, as you might expect, prints "Hello, world" followed by a newline character. Like Python and unlike Perl, no semicolon is required at the end of a line of Ruby code.

Now that we have created a simple command-line program, it's time to create an equivalent CGI program. CGI programs are portable across all types of Web servers. Although not particularly fast or smart, they are easy to write and a good way to dip our toes into the Web development side of a language.

In the case of Ruby, the easiest CGI program would be similar to the above code. After all, the CGI specification tells us that anything written to standard output is sent to the user's Web browser. So long as we send a Content-type header before our text, we can make it a CGI program with almost no effort:

```ruby
#!/usr/bin/env ruby
# HTTP response headers, including double newline
print "Content-type: text/plain\n\n"
# Contents
print "Hello, world\n"
```

Sure enough, naming the above program hello.rb, putting it in my Web server's cgi-bin directory and pointing my Web browser to http://localhost/cgi-bin/hello.rb produces the "Hello, world" message in my browser.

**Using the Ruby Library**

The CGI object in the included Ruby library provides methods that understand Web functionality, from HTML formatting to cookies and parameters. For example, here is a new version of our “Hello, world” program written to use the built-in functionality:

```ruby
#!/usr/bin/env ruby
# -*-ruby-*-
require 'cgi'

# Create an instance of CGI, with HTML 4 output
cgi = CGI.new("html4")

# Send the following to the CGI object's output
cgi.out {
  cgi.html {
    # Produce a header
    cgi.head { cgi.title { "This is a title" } }
  } +

  # Produce a body
  cgi.body {
    cgi.h1 { "This is a headline" } +
```

```
As you can see, the code now looks substantially different, even though the output largely is the same. What we have done is switched from explicit print statements to methods invoked on our CGI object, as well as added a title and a headline.

When we create our CGI object with CGI.new, we can pass an argument indicating the level of HTML compliance we want to have. Unless you have a good reason to do otherwise, aiming for the highest level of compliance, namely HTML4, is a good idea.

Notice how the output, beginning with cgi.out, functions as a set of code blocks, each of which is expected to return a text string. Thus, cgi.h1 and cgi.p are combined—using the + operator, as in Python or Java—and are fed to cgi.body. cgi.head and cgi.body are joined as well and fed to cgi.html. The fact that this hierarchy mimics the eventual document output format makes it easy to understand and use this functionality.

CGI programs are more interesting when they handle parameters from the user. We can get parameters with the CGI.params method:

```ruby
#!/usr/bin/env ruby
# -*-ruby-*- require 'cgi'

# Create an instance of CGI
cgi = CGI.new("html4")

# Get our first name
firstname = cgi.params['firstname']
if (firstname.empty?)
  firstname = '(No firstname)'
end

# Get our last name
lastname = cgi.params['lastname']
if (lastname.empty?)
  lastname = '(No lastname)'
end

# Send some output to the end user
cgi.out {
  cgi.html {
    # Produce a header
cgi.head { cgi.title { "This is a title" } }
+  
    # Produce a body
cgi.body {
      cgi.h1 { "This is a headline" } +
      cgi.p { "Hello, #{firstname} #{lastname}" }
    }
  }
}
```

There are two basic differences between this code and its predecessor. To begin with, we now are defining two variables, firstname and lastname, which we then print for the user. The variables are defined based on the parameter values passed to the program, either by way of the URL in a GET request or in the body of the request for POST. We use the empty? method on both firstname and lastname to check whether they are
empty and then assign a default value to them if that is the case. Finally, we use Ruby's #{expression} syntax within double-quoted strings to display the user's first and last names.

**WEBrick**

The above are what we might expect from simple CGI programs—easy to write, easy to work with and slow to execute. If our programs get any more complicated, we have to deal with new issues that we might prefer to ignore, such as personalization.

Luckily, Ruby comes with its own HTTP server, known as WEBrick, that is similar in some ways to AOLserver or mod_perl. There is also mod_ruby, if you are interested in a more direct equivalent to mod_perl, that runs under Apache. To start a basic HTTP server on port 8000, looking at the same static documents as Apache, use the following code:

```
#!/usr/bin/env ruby
#
require 'webrick'
include WEBrick

# Create an HTTP server
s = HTTPServer.new(
  :Port => 8000,
  :DocumentRoot => '/usr/local/apache/htdocs/'
)

# When the server gets a control-C, kill it
trap('INT') { s.shutdown }

# Start the server
s.start
```

There are several things to note here. First, there isn't much code. You indicate what port WEBrick should listen to, tell it where files are located and then start it up.

Before we start the server, we have to make sure it is possible to stop it easily. To do that, we invoke trap, indicating that we want to trap SIGINT (that is, Ctrl-C) and that s.shutdown should be invoked upon receiving that signal.

If you put the above program in a file named server.rb and execute it, you should have a fully functional HTTP server running on your system. Creating a Web server has never been simpler.

**Ruby Servlets**

Of course, no one runs WEBrick instead of Apache for its speed or to serve static documents. Rather, WEBrick shines when you want to create custom behaviors. In spirit and terminology, there is a fair amount of overlap between WEBrick servlets and Java servlets. The basic idea is the same: define a new class and then attach an instance of that class to a particular URL. For example, if we want to create a servlet that prints the time of day, we can create the following:

```
#!/sw/bin/ruby
require 'webrick'
include WEBrick

# Define a new class
```
class CurrentTimeServlet
  < WEBrick::HTTPServlet::AbstractServlet

    def do_GET(request, response)
      response['Content-Type'] = 'text/plain'
      response.status = 200
      response.body = Time.now.to_s + "\n"
    end

  end

# ----------------------------------------------
#
# Create an HTTP server
#
#
# ----------------------------------------------
#
# Create an HTTP server
#
s = HTTPServer.new(
  :Port            => 8000,
  :DocumentRoot    => "/usr/local/apache/htdocs/"
)

s.mount("/time", CurrentTimeServlet)

# When the server gets a control-C, kill it
# trap("INT"){ s.shutdown }

# Start the server
s.start

Our one file contains both the class definition for CurrentTimeServlet and the commands for starting
WEBrick. This is not the most elegant style for creating a servlet, and you typically want to put each servlet
in its own file. That said, Ruby makes it easy and convenient to define or redefine classes and methods
wherever it might be best to do so. This is one of those features in Ruby that reminds me of Perl: the
language gives you a great deal of flexibility when writing your code but expects you to be responsible
enough to avoid making a mess of it.

We define our servlet, CurrentTimeServlet, to be a subclass of WEBrick::HTTPServlet::AbstractServlet,
making it a simple servlet indeed. We then define the do_GET method along with the do_POST method, if
you so desire, which gets both a request and a response object. If you have written Java servlets, this should
look familiar to you. We set the content type of the response, the status code (200) for the response and even
the body of the response with a few simple lines of code. And that's it; our servlet has been defined and is
ready to go. All that is left to do is connect the servlet to a URL:

s.mount("/time", CurrentTimeServlet)

If we want, we can pass parameters to the servlet when we initialize it. Anything beyond the first two
parameters to s.mount is sent:

s.mount("/time", CurrentTimeServlet, 'a parameter')

Conclusion

Is it amazing that we can do this much in so few lines of code? Perhaps—although similar functionality
certainly exists in other languages. For example, Perl programmers can download HTTP::Server::Simple
from CPAN and do many of the same things. And if I really was interested in modifying the behavior of an
HTTP server to do interesting things, I probably would think of using mod_perl or AOLserver first, for
reasons of performance and flexibility.

That said, WEBrick is extremely easy to get running and for creating custom HTTP-based behaviors. I can
imagine using it to handle Web services, for example, because of the flexibility that Ruby brings to the table,
or to test applications written in Rails.
And, although people are using Ruby and WEBrick for plain-vanilla Web development, most of the excitement seems to be over the specific Rails framework, rather than Ruby or WEBrick themselves. In my next article, we will start to explore Rails—how to install it, how to develop applications with it and how it stacks up against other open-source application frameworks.

**Resources for this article:** [http://www.linuxjournal.com/article/8397](http://www.linuxjournal.com/article/8397).