Chapter 20

Ruby and the Web

Ruby is no stranger to the Internet. Not only can you write your own SMTP server, FTP daemon, or web server in Ruby, but you can also use Ruby for more usual tasks such as CGI programming or as a replacement for PHP.

Many options are available for using Ruby to implement web applications, and a single chapter can't do them all justice. Instead, we'll try to touch some of the highlights and point you toward libraries and resources that can help.

Let's start with some simple stuff: running Ruby programs as Common Gateway Interface (CGI) programs.

Writing CGI Scripts

You can use Ruby to write CGI scripts quite easily. To have a Ruby script generate HTML output, all you need is something like this:

```
#!/usr/bin/ruby
print "Content-type: text/html\r\n\r\n"
print "<html><body>Hello World! It's #{Time.now}</body></html>\r\n"
```

Put this script in a CGI directory, mark it as executable, and you'll be able to access it via your browser. (If your web server doesn't automatically add headers, you'll need to add the response header yourself, as shown in the following code.)

```
#!/usr/bin/ruby
print "HTTP/1.0 200 OK\r\n"
print "Content-type: text/html\r\n\r\n"
print "<html><body>Hello World! It's #{Time.now}</body></html>\r\n"
```

However, that's hacking around at a pretty low level. You'd need to write your own request parsing, session management, cookie manipulation, output escaping, and so on. Fortunately, options are available to make this easier.

Using cgi.rb

Class CGI provides support for writing CGI scripts. With it, you can manipulate forms, cookies, and the environment; maintain stateful sessions; and so on. It's a fairly large class, but we'll take a quick look at its capabilities here.

Quoting

When dealing with URLs and HTML code, you must be careful to quote certain characters. For instance, a slash character (/) has special meaning in a URL, so it must be "escaped" if it's not part of the path name. That is, any / in the query portion of the URL will be translated to the string %2F and must be translated back to a / for you to use it. Space and ampersand are also special characters. To handle this, CGI provides the routines CGI.escape and CGI.unescape:

Download **samples/web_3.rb**

```
require 'cgi'
puts CGI.escape("Nicholas Payton/Trumpet & Flugel Horn")
```

produces:

```
Nicholas+Payton%2FTrumpet+%26+Flugel+Horn
```

More frequently, you may want to escape HTML special characters:

Download **samples/web_4.rb**

```
require 'cgi'
puts CGI.escapeHTML("a < 100 && b > 200")
```

produces:

```
a &lt; 100 &amp;&amp; b &gt; 200
```

To get really fancy, you can decide to escape only certain HTML elements within a string:

Download **samples/web_5.rb**

```
require 'cgi'
puts CGI.escapeElement('<hr><a href="/mp3">Click Here</a><br>','A')
```

produces:

```
<hr>&lt;a href=&quot;/mp3&quot;&gt;Click Here&lt;/a&gt;<br>
```

Here only the A element is escaped; other elements are left alone. Each of these methods has an "un-" version to restore the original string:

Download **samples/web_6.rb**

```
require 'cgi'
puts CGI.unescapeHTML("a &lt; 100 &amp;&amp; b &gt; 200")
```

produces:

```
a < 100 && b > 200
```

Query Parameters

HTTP requests from the browser to your application may contain parameters, either passed as part of the URL or passed as data embedded in the body of the request.

Processing of these parameters is complicated by the fact that a value with a given name may be returned multiple times in the same request. For example, say we're writing a survey to find out why folks like Ruby. The HTML for our form looks like this:

```
<html>
  <head><title>Test Form</title></head>
  <body>
    I like Ruby because:
    <form action="cgi-bin/survey.rb">
      <input type="checkbox" name="reason" value="flexible" />
         It's flexible<br />
      <input type="checkbox" name="reason" value="transparent" />
         It's transparent<br />
      <input type="checkbox" name="reason" value="perlish" />
         It's like Perl<br />
      <input type="checkbox" name="reason" value="fun" />
         It's fun
      <p>
        Your name: <input type="text" name="name">
      </p>
      <input type="submit"/>
    </form>
  </body>
</html>
```

When someone fills in this form, they might check multiple reasons for liking Ruby (as shown in Figure 20.1 on the following page). In this case, the form data corresponding to the name reason will have three values, corresponding to the three checked boxes.

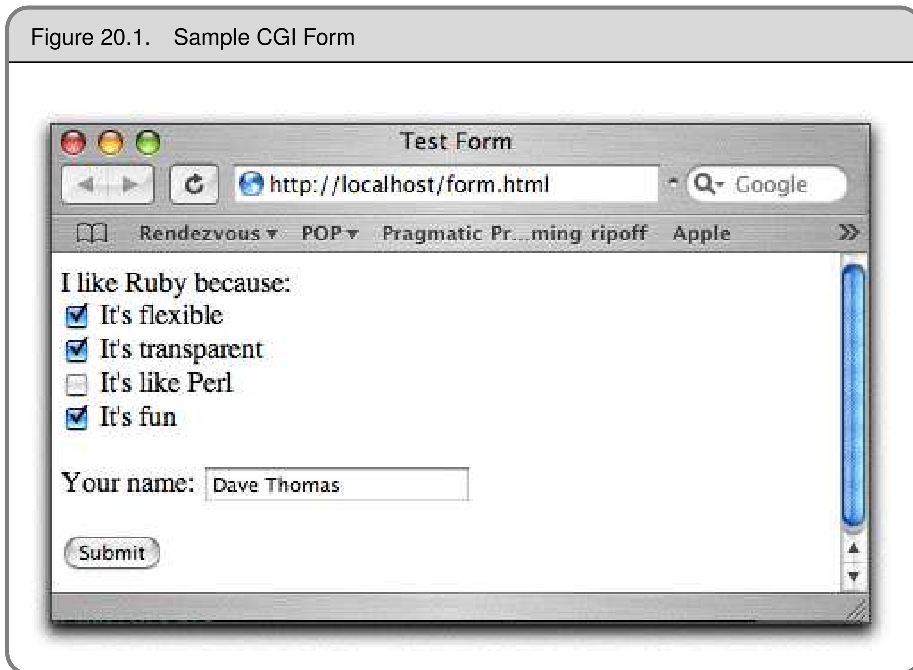
Class CGI gives you access to form data in a couple of ways. First, we can just treat the CGI object as a hash, indexing it with field names and getting back field values.

Download **samples/web_8.rb**

```
require 'cgi'
cgi = CGI.new
cgi['name']     # =>   "Dave Thomas"
cgi['reason']   # =>   "flexible"
```

1.9 However, this doesn't work well with the reason field, because we see only one of the three values. We can ask to see them all by using the CGI#params method. The value returned by params acts like a hash containing the request parameters. You can both read and write this hash (the latter allows you to modify the data associated with a request). Note that each of the values in the hash is actually an array.

Figure 20.1. Sample CGI Form



Download samples/web_9.rb

```
require 'cgi'
cgi = CGI.new
cgi.params              # =>   {"name"=>["Dave Thomas"],
                               "reason"=>["flexible", "transparent",
                               "fun"]}
cgi.params['name']      # =>   ["Dave Thomas"]
cgi.params['reason']    # =>   ["flexible", "transparent", "fun"]
cgi.params['name'] = [ cgi['name'].upcase ]
cgi.params              # =>   {"name"=>["DAVE THOMAS"],
                               "reason"=>["flexible", "transparent",
                               "fun"]}
```

You can determine whether a particular parameter is present in a request by using CGI#has_key?:

Download samples/web_10.rb

```
require 'cgi'
cgi = CGI.new
cgi.has_key?('name')    # =>   true
cgi.has_key?('age')     # =>   false
```

Generating HTML

CGI contains a huge number of methods that can be used to create HTML—one method per element. To enable these methods, you must create a CGI object by calling CGI.new, passing in the required version of HTML. In these examples, we'll use html3.

To make element nesting easier, these methods take their content as code blocks. The code blocks should return a String, which will be used as the content for the element. For this example, we've added some gratuitous newlines to make the output fit on the page:

Download **samples/web_11.rb**

```
require 'cgi'
cgi = CGI.new("html3")  # add HTML generation methods
cgi.out do
  cgi.html do
    cgi.head { "\n"+cgi.title { "This Is a Test"} } +
    cgi.body do "\n"+
      cgi.form do"\n"+
        cgi.hr +
        cgi.h1 { "A Form: " } + "\n"+
        cgi.textarea("get_text") +"\n"+
        cgi.br +
        cgi.submit
      end
    end
  end
end
```

produces:

```
Content-Type: text/html
Content-Length: 302

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN"><HTML><HEAD>
<TITLE>This Is a Test</TITLE></HEAD><BODY>
<FORM METHOD="post" ENCTYPE="application/x-www-form-urlencoded">
<HR><H1>A Form: </H1>
<TEXTAREA NAME="get_text" COLS="70" ROWS="10"></TEXTAREA>
<BR><INPUT TYPE="submit"></FORM></BODY></HTML>
```

Although vaguely interesting, this method of generating HTML is fairly laborious and probably isn't used much in practice. Most people seem to write the HTML directly, use a templating system, or use an application framework, such as Rails. Unfortunately, we don't have space here to discuss Rails—take a look at the online documentation at http:/rubyonrails.com—but we can look at templating (including erb, the templating engine used by Rails).

Templating Systems

Templating systems let you separate the presentation and logic of your application. It seems that just about everyone who writes a web application using Ruby at some point also writes a

templating system; the RubyGarden wiki lists quite a few,[1] and even this list isn't complete. For now, let's just look at two: Haml and erb/eruby. Also, remember to look at Builder if you need to generate XHTML or XML. (We saw Builder briefly starting on page 240.)

Haml

Haml[2] is a library that generates HTML documents from a template. Unlike many other templating systems, Haml uses indentation to indicate nesting (yup, just like Python). For example, you can represent a <ul> in Haml using this:

```
%ul
  %li item one
  %li item two
```

Install Haml using this:

```
% gem install haml
```

The Haml input syntax is rich and powerful, and the example that follows touches on only a subset of the features. Lines starting % get converted to HTML tags, nested in the output according to their indentation in the input. An equals sign means *substitute in the value of the Ruby code that follows*. A minus sign executes Ruby code but doesn't substitute the value in—our example uses that to look over the reasons when constructing the table.

There are many ways of getting values passed in to the template. In this example, we chose to pass in a hash as the second parameter to render. This results in local variables getting set as the template is expanded, one variable for each key in the hash:

Download **samples/web_13.rb**

```
require 'haml'
engine = Haml::Engine.new(%{
%body
  #welcome-box
    %p= greeting
  %p
    As of
    = Time.now
    the reasons you gave were:
  %table
    %tr
      %th Reason
      %th Rank
    - for reason in reasons
      %tr
        %td= reason[:reason_name]
        %td= reason[:rank]
})
```

1. http://www.rubygarden.org/ruby?HtmlTemplates

2. http://haml.hamptoncatlin.com//

```
data = {
  :greeting => 'Hello, Dave Thomas',
  :reasons  => [
    { :reason_name => 'flexible',    :rank => '87' },
    { :reason_name => 'transparent', :rank => '76' },
    { :reason_name => 'fun',         :rank => '94' },
  ]
}
puts engine.render(nil, data)
```

produces:

```
<body>
  <div id='welcome-box'>
    <p>Hello, Dave Thomas</p>
  </div>
  <p>
    As of
    2009-04-13 13:26:10 -0500
    the reasons you gave were:
  </p>
  <table>
    <tr>
      <th>Reason</th>
      <th>Rank</th>
    </tr>
    <tr>
      <td>flexible</td>
      <td>87</td>
    </tr>
    <tr>
      <td>transparent</td>
      <td>76</td>
    </tr>
    <tr>
      <td>fun</td>
      <td>94</td>
    </tr>
  </table>
</body>
```

erb and eruby

So far we've looked at using Ruby to create HTML output, but we can turn the problem inside out; we can actually embed Ruby in an HTML document.

A number of packages allow you to embed Ruby statements in some other sort of a document, especially in an HTML page. Generically, this markup is known as "eRuby." Specifically, several different implementations of eRuby exist, including erubis and erb. erubis is available as a gem, while erb is written in pure Ruby and is included with the standard distribution. We'll look at erb here.

Embedding Ruby in HTML is a very powerful concept—it basically gives us the equivalent of a tool such as ASP, JSP, or PHP, but with the full power of Ruby.

Using erb

erb is normally used as a filter. Input text is passed through untouched, with the following exceptions:

Expression	**Description**
<% *ruby code* %>	This executes the Ruby code between the delimiters.
<%= *ruby expression* %>	This evaluates the Ruby expression and replaces the sequence with the expression's value.
<%# *ruby code* %>	The Ruby code between the delimiters is ignored (useful for testing).
% *line of ruby code*	A line that starts with a percent is assumed to contain just Ruby code.

You can run erb from the command line:

erb *[options] [document]*

If the *document* is omitted, erb will read from standard input. The command-line options for erb are shown in Table 20.1 on the following page.

Let's look at some simple examples. We'll run the erb executable on the following input:

```
% 99.downto(96) do |number|
<%= number %> bottles of beer...
% end
```

The lines starting with the percent sign simply execute the given Ruby. In this case, it's a loop that iterates the line between them. This middle line contains the sequence <%= number %>, which substitutes in the value of number into the output.

```
% erb f1.erb
```

produces:

```
99 bottles of beer...
98 bottles of beer...
97 bottles of beer...
96 bottles of beer...
```

erb works by rewriting its input as a Ruby script and then executing that script. You can see the Ruby that erb generates using the -n or -x option:

```
% erb -x f1.erb
```

produces:

```
_erbout = '';  99.downto(96) do |number|
_erbout.concat(( number ).to_s); _erbout.concat " bottles of beer...\n"
 end
_erbout
```

Table 20.1. Command-Line Options for erb

Option	Description
-d	Sets $DEBUG to true
-E *ext[:int]*	Sets the default external/internal encodings
-n	Displays resulting Ruby script (with line numbers)
-r *library*	Loads the named *library*
-P	Doesn't do erb processing on lines starting %
-S *level*	Sets the *safe level*
-T *mode*	Sets the *trim mode*
-U	Sets default encoding to UTF-8
-v	Enables verbose mode
-x	Displays resulting Ruby script

Notice how erb builds a string, _erbout, containing both the static strings from the template and the results of executing expressions (in this case the value of number).

Embedding erb in Your Code

So far we've shown erb running as a command-line filter. However, the most common use is to use it as a library in your own code. (This is what Rails does with its .erb templates.)

Download **samples/web_17.rb**

```
require 'erb'
SOURCE =
%{<% for number in min..max %>
The number is <%= number %>
<% end %>
}
erb = ERB.new(SOURCE)
min = 4
max = 6
puts erb.result(binding)
```

produces:

```

The number is 4

The number is 5

The number is 6
```

Notice how we can use local variables within the erb template. This works because we pass the current *binding* to the result method. erb can use this binding to make it look as if the template is being evaluated in the context of the calling code.

erb comes with excellent documentation: use ri to read it. One thing that Rails users should know is that in the standard version of erb, you can't use the -%> trick to suppress blank lines. (In the previous example, that's why we have the extra blank lines in the output.) Take a look at the description of trim modes in the documentation of ERB.new for alternatives.

Cookies

Cookies are a way of letting web applications store their state on the user's machine. Frowned upon by some, cookies are still a convenient (if unreliable) way of remembering session information.

The Ruby CGI class handles the loading and saving of cookies for you. You can access the cookies associated with the current request using the CGI#cookies method, and you can set cookies back into the browser by setting the cookie parameter of CGI#out to reference either a single cookie or an array of cookies:

```
#!/usr/bin/ruby
COOKIE_NAME = 'chocolate chip'
require 'cgi'
cgi = CGI.new
values = cgi.cookies[COOKIE_NAME]
if values.empty?
  msg = "It looks as if you haven't visited recently"
else
  msg = "You last visited #{values[0]}"
end
cookie = CGI::Cookie.new(COOKIE_NAME, Time.now.to_s)
cookie.expires = Time.now + 30*24*3600 # 30 days
cgi.out("cookie" => cookie ) { msg }
```

Sessions

Cookies by themselves still need a bit of work to be useful. We really want *sessions:* information that persists between requests from a particular web browser. Sessions are handled by class CGI::Session, which uses cookies but provides a higher-level abstraction.

As with cookies, sessions emulate a hashlike behavior, letting you associate values with keys. Unlike cookies, sessions store the majority of their data on the server, using the browser-resident cookie simply as a way of uniquely identifying the server-side data. Sessions also give you a choice of storage techniques for this data: it can be held in regular files, in a PStore (see the description on page 794), in memory, or even in your own customized store.

Sessions should be closed after use, because this ensures that their data is written out to the store. When you've permanently finished with a session, you should delete it.

```
require 'cgi'
require 'cgi/session'
cgi = CGI.new("html3")
sess = CGI::Session.new(cgi,
                        "session_key" => "rubyweb",
                        "prefix" => "web-session.")
if sess['lastaccess']
  msg = "<p>You were last here #{sess['lastaccess']}.</p>"
else
  msg = "<p>Looks like you haven't been here for a while</p>"
end
count = (sess["accesscount"] || 0).to_i
count += 1
msg << "<p>Number of visits: #{count}</p>"
sess["accesscount"] = count
sess["lastaccess"]  = Time.now.to_s
sess.close
cgi.out {
  cgi.html {
    cgi.body {
      msg
    }
  }
}
```

The code in the previous example used the default storage mechanism for sessions: persistent data was stored in files in your default temporary directory (see Dir.tmpdir). The filenames will all start with web-session. and will end with a hashed version of the session number. See ri CGI::Session for more information.

Choice of Web Servers

So far, we've been running Ruby scripts under the Apache web server. However, Ruby comes bundled with WEBrick, a flexible, pure-Ruby HTTP server toolkit. WEBrick's an extensible plug-in–based framework that lets you write servers to handle HTTP requests and responses. Here's a basic HTTP server that serves documents and directory indexes:

```
#!/usr/bin/ruby
require 'webrick'
include WEBrick
s = HTTPServer.new(
  :Port         => 2000,
  :DocumentRoot => File.join(Dir.pwd, "/html"))
trap("INT") { s.shutdown }
s.start
```

The HTTPServer constructor creates a new web server on port 2000. The code sets the document root to be the html/ subdirectory of the current directory. It then uses Kernel.trap to arrange to shut down tidily on interrupts before starting the server running. If you point your browser at http://localhost:2000, you should see a listing of your html subdirectory.

WEBrick can do far more than serve static content. You can use it just like a Java servlet container. The following code mounts a simple servlet at the location /hello. As requests arrive, the do_GET method is invoked. It uses the response object to display the user agent information and parameters from the request.

```
#!/usr/bin/ruby
require 'webrick'
include WEBrick
s = HTTPServer.new( :Port => 2000 )
class HelloServlet < HTTPServlet::AbstractServlet
  def do_GET(req, res)
    res['Content-Type'] = "text/html"
    res.body = %{
      <html><body>
        <p>Hello. You're calling from a #{req['User-Agent']}</p>
        <p>I see parameters: #{req.query.keys.join(', ')}</p>
      </body></html>
    }
  end
end
s.mount("/hello", HelloServlet)
trap("INT"){ s.shutdown }
s.start
```

More information on WEBrick is available from http://www.webrick.org. There you'll find links to a set of useful servlets, including one that lets you write SOAP servers in Ruby.

Frameworks

In reality, CGI is just the start of using Ruby on the Web. Most of the real action these days is with frameworks. Frameworks abstract away all this low-level detail and also help you structure your code into something that is both easy to write and (probably more importantly) easy to maintain.

At the time of writing, Ruby on Rails[3] is the leading web framework for Ruby. It has an incredibly active community and a vast set of plug-ins so the chances are good you'll find a lot of preexisting code to help you kick-start your application. Merb[4] is a lighter-weight alternative. Rails and Merb will merge and become Rails 3. Other alternatives include Camping, Sinatra, and Ramaze.[5] By the time you read this, the list will have grown. And, if you fancy writing your own framework, consider making it independent of the underlying web server by building it on top of Rack.[6]

3. http://www.rubyonrails.com

4. http://merbivore.com/

5. http://camping.rubyforge.org/files/README.html, http://sinatra.rubyforge.org/, and http://ramaze.net/

6. http://rack.rubyforge.org/