Table of Contents

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

14. Internet Services

Network programming is hard. The C socket library is the standard way of writing Internet clients and servers. It's like the file API descibed in Chapter 6, with its special flags and meager abstraction, only much more complicated. It's a shame because networked applications are the coolest kind of application. Only computer nerds like you and me care about XML or the best way to sort a list, but everyone uses Internet applications.

Fortunately, network programming is easy. Ruby provides bindings to the C socket library (in `socket`), but you'll probably never need to use them. Existing Ruby libraries (some in the standard distribution) can speak every popular high-level Internet protocol.

The most popular Internet service is, of course, the Web, and Ruby's most popular Internet library (or any kind of library, actually) is the Rails framework. We've devoted the entire next chapter to Rails (Chapter 15) so that we can cover other technologies here.

Apart from Rails, most of the interesting stuff you can do with Ruby happens on the client end. We start with a set of recipes for requesting web pages (Recipes 14.1, 14.2, and 14.3), which are brought together at the end of the chapter with Recipe 14.20. Combine these recipes with one from Chapter 11 (probably Recipe 11.5), and you can make your own spider or web browser.

Then we present Ruby clients for the most popular Internet protocols. Ruby can do just about everything you do online: send and receive email, perform nameserver queries, even transfer files with FTP, SCP, or BitTorrent. With the Ruby interfaces, you can write custom clients for these protocols, or integrate them into larger programs.

It's less likely that you'll be writing your own server in Ruby. A server only exists to service clients, so there's not much you can do but faithfully implement the appropriate protocol. If you do write a server, it'll probably be for a custom protocol, one for which no other server exists.

Ruby provides two basic servers that you can use as a starting point. The `gserver` library described in Recipe 14.14 provides a generic framework for almost any kind of Internet server. Here you do have to do some socket programming, but only the easy parts. `gserver` takes care of all the socket-specific details, and you can just treat the sockets like read-write `IO` objects. You can use the techniques described in Chapter 6 to communicate with your clients.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

The other basic server is WEBrick, a simple but powerful web server that's used as the basis for Rails and the Ruby SOAP server. If you've built a protocol on top of HTTP, WEBrick makes a good starting point for a server. Recipe 14.19 shows how to use WEBrick to hook pieces of Ruby code up to the Web.

Apart from Rails, web services are the major network-related topic not covered in this chapter. As with Rails, this is because they have their own chapter: Chapter 16.

Recipe 14.1. Grabbing the Contents of a Web Page

Problem

You want to display or process a specific web page.

Solution

The simplest solution is to use the `open-uri` library. It lets you open a web page as though it were a file. This code fetches the `oreilly.com` homepage and prints out the first part of it:

```
require 'open-uri'
puts open('http://www.oreilly.com/').read(200)
# <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
#         "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
# <html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
```

For more complex applications, you'll need to use the `net/http` library. Use `Net::HTTP.get_response` to make an HTTP request and get the response as a `Net::HTTPResponse` object containing the response code, headers, and body.

```
require 'net/http'
response = Net::HTTP.get_response('www.oreilly.com', '/about/')
response.code                                          # => "200"
response.body.size                                     # => 21835
response['Content-type']
# => "text/html; charset=ISO-8859-1"
puts response.body[0,200]
# <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
#     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
#
#
# <html>
# <head>
# <meta http-equiv="content-type" content="text/html; c
```

Rather than passing in the hostname, port, and path as separate arguments, it's usually easier to create `URI` objects from URL strings and pass those into the `Net::HTTP` methods.

```
require 'uri'
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
Net::HTTP.get(URI.parse("http://www.oreilly.com"))
Net::HTTP.get_response(URI.parse("http://www.oreilly.com/about/"))
```

Discussion

If you just want the text of the page, use `get`. If you also want the response code or the values of the HTTP response headers, use `get_reponse`.

The `get_response` method returns some `HTTPResponse` subclass of `Net:HTTPResponse`, which contains all information about an HTTP response. There's one subclass for every response code defined in the HTTP standard; for instance, `HTTPOK` for the 200 response code, `HTTPMovedPermanently` for the 301 response code, and `HTTPNotFound` for the 404 response code. There's also an `HTTPUnknown` subclass for any response codes not defined in HTTP.

The only difference between these subclasses is the class name and the `code` member. You can check the response code of an HTTP response by comparing specific classes with `is_a?`, or by checking the result of `HTTPResponse#code`, which returns a `String`:

```
puts "Success!" if response.is_a? Net::HTTPOK
# Success!

puts case response.code[0] # Check the first byte of the response code.
  when ?1 then "Status code indicates an HTTP informational response."
  when ?2 then "Status code indicates success."
  when ?3 then "Status code indicates redirection."
  when ?4 then "Status code indicates client error."
  when ?5 then "Status code indicates server error."
  else "Non-standard status code."
end
# Status code indicates success.
```

You can get the value of an HTTP response header by treating `HTTPResponse` as a hash, passing the header name into `HTTPResponse#[]`. The only difference from a real `Hash` is that the names of the headers are case-insensitive. Like a hash, `HTTPResponse` supports the iteration methods `#each`, `#each_key`, and `#each_value`:

```
response['Server']
# => "Apache/1.3.34 (Unix) PHP/4.3.11 mod_perl/1.29"
response['SERVER']
# => "Apache/1.3.34 (Unix) PHP/4.3.11 mod_perl/1.29"

response.each_key { |key| puts key }
# x-cache
# p3p
# content-type
# date
# server
# transfer-encoding
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

If you do a request by calling `NET::HTTP.get_response` with no code block, Ruby will read the body of the web page into a string, which you can fetch with the `HTTPResponse::body` method. If you like, you can process the body as you read it, one segment at a time, by passing a code block to `HTTPResponse::read_body`:

```
Net::HTTP.get_response('www.oreilly.com', '/about/') do |response|
  response.read_body do |segment|
    puts "Received segment of #{segment.size} byte(s)!"
  end
end
# Received segment of 614 byte(s)!
# Received segment of 1024 byte(s)!
# Received segment of 848 byte(s)!
# Received segment of 1024 byte(s)!
# …
```

Note that you can only call `read_body` once per request. Also, there are no guarantees that a segment won't end in the middle of an HTML tag name or some other inconvenient place, so this is best for applications where you're not handing the web page as structured data: for instance, when you're simply piping it to some other source.

See Also

- Recipe 14.2, "Making an HTTPS Web Request"
- Recipe 14.3, "Customizing HTTP Request Headers"
- Recipe 14.20, "A Real-World HTTP Client," covers a lot of edge cases you'll need to handle if you want to write a general-purpose client
- Most HTML you'll find on the web is invalid, so to parse it you'll need the tricks described in Recipe 11.5, "Parsing Invalid Markup"

Recipe 14.2. Making an HTTPS Web Request

Problem

You want to connect to an HTTPS web site, one whose traffic is encrypted using SSL.

Solution

You need the OpenSSL extension to Ruby. You'll know if it's installed if you can `require` the `net/https`library without getting a `LoadError`.

```
require 'net/https'                    # => true
```

You can't make HTTPS requests with the convenience methods described in Recipe 14.1, but you can use the `Net::HTTP::Get` and `Net::HTTP::Post` class described in Recipe

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

14.3. To make an HTTPS request, just instantiate a `Net::HTTP` object and set its `use_ssl` member to `true`.

In this example, I try to download a page from a web server that only accepts HTTPS connections. Instead of listening on port 80 like a normal web server, this server listens on port 443 and expects an encrypted request. I can only connect with a `Net::HTTP` instance that has the `use_ssl` flag set.

```
require 'net/http'
uri = URI.parse("https://www.donotcall.gov/")

request = Net::HTTP.new(uri.host, uri.port)
response = request.get("/")
# Errno::ECONNRESET: Connection reset by peer

require 'net/https'
request.use_ssl = true
request.verify_mode = OpenSSL::SSL::VERIFY_NONE
response = request.get("/")
# => #<Net::HTTPOK 200 OK readbody=true>
response.body.size                              # => 6537
```

Discussion

The default Ruby installation for Windows includes the OpenSSL extension, but if you're on a Unix system, you might have to install it yourself. On Debian GNU/Linux, the package name is `libopenssl-ruby[Ruby version]`: for instance, `libopenssl-ruby1.8`. You might need to download the extension from the Ruby PKI homepage (see below), and compile and install it with Make.

Setting `verify_mode` to `OpenSSL::SSL::VERIFY_NONE` suppresses some warnings, but the warnings are kind of serious: they mean that OpenSSL won't verify the server's certificate or proof of identity. Your conversation with the server will be confidential, but you won't be able to definitively authenticate the server: it might be an imposter.

You can have OpenSSL verify server certificates if you keep a few trusted certificates on your computer. You don't need a certificate for every server you might possibly access. You just need certificates for a few "certificate authorities:" the organizations that actually sign most other certificates. Since web browsers need these certificates too, you probably already have a bunch of them installed, although maybe not in a format that Ruby can use (if you don't have them, see below).

On Debian GNU/Linux, the `ca-certificates` package installs a set of trusted server certificates into the directory `/etc/ssl/certs`. I can set my request object's `ca_path` to that directory, and set its `verify_mode` to `OpenSSL::SSL::VERIFY_PEER`. Now OpenSSL can verify that I'm actually talking to the web server at donotcall.gov, and not an imposter.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
request = Net::HTTP.new(uri.host, uri.port)
request.use_ssl = true
request.ca_path = "/etc/ssl/certs/"
request.verify_mode = OpenSSL::SSL::VERIFY_PEER
response = request.get("/")
# => #<Net::HTTPOK 200 OK readbody=true>
```

The SSL certificate for www.donotcall.gov (http://www.donotcall.gov) happens to be signed by Network Solutions. I already have Network Solutions' certificate installed on my computer, so I can verify the signature. If I trust Network Solutions, I can trust donotcall.gov.

See Also

- Recipe 14.1, "Grabbing the Contents of a Web Page"
- HTTPS is just one more thing a robust web client needs to support; Recipe 14.20, "A Real-World HTTP Client," shows how to integrate it into a general framework
- The Ruby OpenSSL project homepage (http://www.nongnu.org/rubypki/)
- The (unofficial) Mozilla Certificate FAQ provides a good introduction to SSL certificates (http://www.hecker.org/mozilla/ca-certificate-faq/background-info)
- If you don't have any certs on your system or they're not in a format you can give to Ruby, you can download a bundle of all the certs recognized by the Mozilla web browser; instead of setting `ca_path` to a directory, you'll set `ca_file` to the location of the file you download (http://curl.haxx.se/docs/caextract.html)
- You can create your own server certificates with the QuickCert program; your certificates won't be recognized by any certificate authority, but if you control the clients as well as the server, you can manually install the server certificate on every client (http://segment7.net/projects/ruby/QuickCert/)

Recipe 14.3. Customizing HTTP Request Headers

Problem

When you make an HTTP request, you want to specify custom HTTP headers like "User-Agent" or "Accept-Language".

Solution

Pass in a `Hash` of header values to `Net::HTTP#get` or `Net::HTTP#post`:

```
require 'net/http'
require 'uri'

#A simple wrapper method that accepts either strings or URI objects
#and performs an HTTP GET.
module Net
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
  class HTTP
    def HTTP.get_with_headers(uri, headers=nil)
      uri = URI.parse(uri) if uri.respond_to? :to_str
      start(uri.host, uri.port) do |http|
        return http.get(uri.path, headers)
      end
    end
  end
end

#Let's get a web page in German.
res = Net::HTTP.get_with_headers('http://www.google.com/',
                                 {'Accept-Language' => 'de'})

#Check a bit of the body to make sure it's really in German.
s = res.body.size
res.body[s-200..s-140]
# => "ngebote</a> - <a href=/intl/de/about.html>Alles \374ber Google</"
```

Discussion

Usually you can retrieve the web pages you want without specifying any custom HTTP headers. As you start performing more complicated interactions with web servers, you'll find yourself customizing the headers more.

For instance, if you write a web spider or client, you'll want it to send a "User-Agent" header on every request, identifying itself to the web server. Unlike the HTTP client libraries for other programming languages, the `net/http` library doesn't send a "User-Agent" header by default; it's your reponsibility to send one.

```
Net::HTTP.get_with_headers(url, {'User-Agent' => 'Ruby Web Browser v1.0'})
```

You can often save bandwidth (at the expense of computer time) by sending an "Accept-Encoding" header, requesting that a web server compress data before sending it to you. Gzip compression is the most common way a server compresses HTTP response data; you can reverse it with Ruby's `zlib` library:

```
uncompressed = Net::HTTP.get_with_headers('http://www.cnn.com/')
uncompressed.body.size
# => 65150

gzipped = Net::HTTP.get_with_headers('http://www.cnn.com/',
                                     {'Accept-Encoding' => 'gzip'})
gzipped['Content-Encoding']
# => "gzip"
gzipped.body.size
# => 14600

require 'zlib'
require 'stringio'
body_io = StringIO.new(gzipped.body)
unzipped_body = Zlib::GzipReader.new(body_io).read()
unzipped_body.size
# => 65150
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

If you want to build up a HTTP request with multiple values for the same HTTP header, you can construct a `Net::HTTP::Get` (or `Net::HTTP::Post`) object and call the `add_field` method multiple times. The example in the Solution used the "Accept-Language" header to request a document in a specific language. The following code fetches the same URL, but its "Accept-Language" header indicates that it will accept a document written in any of four different dialects:

```
uri = URI.parse('http://www.google.com/')

request = Net::HTTP::Get.new(uri.path)
['en_us', 'en', 'en_gb', 'ja'].each do |language|
  request.add_field('Accept-Language', language)
end
request['Accept-Language']
# => "en_us, en, en_gb, ja"

Net::HTTP.start(uri.host, uri.port) do |http|
  response = http.request(request)
  # … process the HTTPResponse object here
end
```

See Also

- Recipe 12.10, "Compressing and Archiving Files with Gzip and Tar," for more about the `zlib` library
- Recipe 14.1, "Grabbing the Contents of a Web Page"
- Recipe 14.20, "A Real-World HTTP Client," covers a lot of edge cases you'll need to handle if you want to write a general-purpose client
- REST web services often use the value of the "Accept" header to provide multiple representations of the same resource; Joe Gregorio's article "Should you use Content Negotiation in your Web Services?" explains why it's a better idea to provide a different URL for each representation (http://bitworking.org/news/WebServicesAndContentNegotiation)
- Recipe 16.1 for more on REST web services

Recipe 14.4. Performing DNS Queries

Problem

You want to find the IP address corresponding to a domain name, or see whether a domain provides a certain service (such as an email server).

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Solution

Use the `Resolv::DNS` class in the standard `resolv` library to perform DNS lookups. The most commonly used method is `DNS#each_address`, which iterates over the IP addresses assigned to a domain name.

```
require 'resolv'
Resolv::DNS.new.each_address("oreilly.com") { |addr| puts addr }
# 208.201.239.36
# 208.201.239.37
```

Discussion

If you need to check the existence of a particular type of DNS record (such as a MX record for a mail server), use `DNS#getresources` or the iterator `DNS#each_resource`. Both methods take a domain name and a class denoting a type of DNS record. They perform a DNS lookup and, for each matching DNS record, return an instance of the given class.

These are the three most common classes:

`DNS::Resource::IN::A`

: Indicates a DNS record pointing to an IP address for the domain.

`DNS::RESOURCE::IN::NS`

: Indicates a DNS record pointing to a DNS nameserver.

`DNS::Resource::IN::MX`

: Indicates a DNS record pointing to a mail server.

This code finds the mail servers and name servers responsible for oreilly.com:

```
dns = Resolv::DNS.new
domain = "oreilly.com"
dns.each_resource(domain, Resolv::DNS::Resource::IN::MX) do |mail_server|
  puts mail_server.exchange
end
# smtp1.oreilly.com
# smtp2.oreilly.com

dns.each_resource(domain, Resolv::DNS::Resource::IN::NS) do |nameserver|
  puts nameserver.name
end
# a.auth-ns.sonic.net
# b.auth-ns.sonic.net
# c.auth-ns.sonic.net
# ns.oreilly.com
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

If your application needs to do a lot of DNS lookups, you can greatly speed things up by creating a separate thread for each lookup. Most of the time spent doing a DNS lookup is spent connecting to the network, so doing all the lookups in parallel can save a lot of time. If you do this, you should `include` the `resolv-replace` library along with `resolv`, to make sure your DNS lookups are thread-safe.

Here's some code that sees which one-letter *.com* domains (*a.com, b.com*, etc.) are mapped to IP addresses. It runs all 26DNS queries at once, in 26 threads, and summarizes the results.

```
require 'resolv-replace'
def multiple_lookup(*names)
  dns = Resolv::DNS.new
  results = {}
  threads = []
  names.each do |name|
    threads << Thread.new(name) do |name|
      begin
        dns.each_address(name) { |a| (results[name] ||= []) << a }
      rescue Resolv::ResolvError
        results[name] = nil
      end
    end
  end
  threads.each { |t| t.join }
  return results
end

domains = ("a".."z").collect { |l| l + '.com' }
multiple_lookup(*domains).each do |name, addresses|
  if addresses
    puts "#{name}: #{addresses.size} address#{addresses.size == 1 ? "" : "es"}"
  end
end
# x.com: 4 addresses
# z.com: 1 address
# q.com: 1 address
```

See Also

- Chapter 20 uses a DNS lookup of an MX record to check whether the domain of an email address is valid
- A DNS lookup is the classic example of a high-latency operation; much of Chapter 20 deals with ways of making high-latency operations run more quickly: see especially Recipe 20.3, "Doing Two Things at Once with Threads," and Recipe 20.6, "Running a Code Block on Many Objects Simultaneously"

Recipe 14.5. Sending Mail

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Problem

You want to send an email message, either an autogenerated one or one entered in by an end user.

Solution

First you need to turn the parts of the email message into a single string, representing the whole message complete with headers and/or attachments. You can construct the string manually or use a number of libraries, including RubyMail, TMail, and ActionMailer. Since ActionMailer is one of the dependencies of Rails, I'll use it throughout this recipe. ActionMailer uses TMail under the covers, and it's provided by the `actionmailer` gem.

Here, I use ActionMailer to construct a simple, single-part email message:

```
require 'rubygems'
require 'action_mailer'

class SimpleMailer < ActionMailer::Base
  def simple_message(recipient)
     from 'leonardr@example.org'
     recipients recipient
     subject 'A single-part message for you'
     body 'This message has a plain text body.'
   end
end
```

ActionMailer then makes two new methods available for generating this kind of email message: `SimpleMailer.create_simple_message`, which returns the email message as a data structure, and `SimpleMailer.deliver_simple_message`, which actually sends the message.

```
puts SimpleMailer.create_simple_message('lucas@example.com')
# From: leonardr@example.org
# To: lucas@example.com
# Subject: A single-part message for you
# Content-Type: text/plain; charset=utf-8
#
# This message has a plain text body.
```

To deliver the message, call `deliver_simple_message` instead of `create_simple_message`. First, though, you'll need to tell ActionMailer about your SMTP server. If you're sending mail from `example.org` and you've got an SMTP server on the local machine, you might send a message this way:

```
ActionMailer::Base.server_settings = { :address => 'localhost',
                                       :port => 25, # 25 is the default
                                       :domain => 'example.org' }

SimpleMailer.deliver_simple_message('lucas@example.com')
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

If you're using your ISP's SMTP server, you'll probably need to send authentication information so the server knows you're not a spammer. Your ActionMailer setup will probably look like this:

```
ActionMailer::Base.server_settings = { :address => 'smtp.example.org',
                                       :port => 25,
                                       :domain => 'example.org',
                                       :user_name => 'leonardr@example.org',
                                       :password => 'my_password',
                                       :authentication => :login }

SimpleMailer.deliver_simple_message('lucas@example.com')
```

Discussion

Unless you're writing a general-purpose mail client, you probably won't be letting your users compose emails from scratch. More likely, you'll define a template for every *type* of email your application might send, and fill it in with custom data every time you send a message.[1]

[1] You can use ActionMailer even if you are writing a general-purpose mail client (just write a single hook method called `custom_messge` that takes a whole lot of arguments), but you might prefer to drop down a level and use TMail or RubyMail.

This is what ActionMailer is designed for. The `simple_message` method defined above is actually a hook method that makes ActionMailer respond to two other methods: `create_simple_message` and `deliver_simple_message`. The hook method defines the headers and body of a message template, the `create_` method instantiates the template with specific values, and the `deliver_` method actually delivers the email. You never call `simple_message` directly.

Within your hook method, you can set most of the standard email headers by calling a method of the same name (`subject, cc`, and so on). You can also set custom headers by modifying the `@headers` instance variable:

```
class SimpleMailer
  def headerful_message
     @headers['A custom header'] = 'Its value'
     body 'Body'
  end
end

puts SimpleMailer.create_headerful_message
# Content-Type: text/plain; charset=utf-8
# A custom header: Its value
#
# Body
```

You can create a multipart message with attachments by passing the MIME type of the attachment into the `attachment` method.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Here's a method that creates a message containing a dump of the files in a directory (perhaps a bunch of logfiles). It uses the `mime-types` gem to determine the probable MIME type of a file, based on its filename:

```
require 'mime/types'

class SimpleMailer
  def directory_dump_message(recipient, directory)
    from 'directory-dump@example.org'
    recipients recipient
    subject "Dump of #{directory}"
    body %{Here are the files currently in "#{directory}":}

    Dir.new(directory).each do |f|
      path = File.join(directory, f)
      if File.file? path
        mime_type = MIME::Types.of(f).first
        content_type = (mime_type ? mime_type.content_type :
                        'application/binary')
        attachment(content_type) do |a|
          a.body = File.read(path)
          a.filename = f
          a.transfer_encoding = 'quoted-printable' if content_type =~ /^text\//
        end
      end
    end
  end
end

SimpleMailer.create_directory_dump_message('lucas@example.com',
                                           'email_test')
```

Here it is in action:

```
Dir.mkdir('email_test')
open('email_test/image.jpg', 'wb') { |f| f << "\377\330\377\340\000\020JFIF" }
open('email_test/text.txt', 'w') { |f| f << "Here's some text." }

puts SimpleMailer.create_directory_dump_message('lucas@example.com',
                                                'email_test')
# From: directory-dump@example.org
# To: lucas@example.com
# Subject: Dump of email_test
# Mime-Version: 1.0
# Content-Type: multipart/mixed; boundary=mimepart_443d73ecc651_3ae1..fdbeb1ba4328
#
#
# --mimepart_443d73ecc651_3ae1..fdbeb1ba4328
# Content-Type: text/plain; charset=utf-8
# Content-Disposition: inline
#
# Here are the files currently in "email_test":
# --mimepart_443d73ecc651_3ae1..fdbeb1ba4328
# Content-Type: image/jpeg; name=image.jpg
# Content-Transfer-Encoding: Base64
# Content-Disposition: attachment; filename=image.jpg
#
# /9j/4AAQSkZJRg==
#
# --mimepart_443d73ecc651_3ae1..fdbeb1ba4328
# Content-Type: text/plain; name=text.txt
# Content-Transfer-Encoding: Quoted-printable
# Content-Disposition: attachment; filename=text.txt
#
# Here's some text.=
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
#
# --mimepart_443d73ecc651_3ae1..fdbeb1ba4328--
```

If you're a minimalist, you can use the `net/smtp`library to send email without installing any gems. There's nothing in the Ruby standard library to help you with creating the email string, though; you'll have to build it manually. Once you've got the string, you can send it as an email message with code like this:

```
require 'net/smtp'
Net::SMTP.start('smtp.example.org', 25, 'example.org',
                'leonardr@example.org', 'my_password', :login) do |smtp|
  smtp.send_message(message_string, from_address, to_address)
end
```

Whether you use `Net::SMTP` or ActionMailer to deliver your mail, the possible SMTP authentication schemes are represented with symbols (`:login, :plain`, and `:cram_md5`). Any given SMTP server may support any or all of these schemes. Try them one at a time, or ask your system administrator or ISP which one to use.

See Also

- Recipe 15.19, "Sending Mail with Rails," if you're using Rails
- The ActionMailer documentation (http://www.lickey.com/rubymail/rubymail/doc/)
- The standard for email messages (RFC 2822)
- More ActionMailer examples (http://am.rubyonrails.com/classes/ActionMailer/Base.html)

Recipe 14.6. Reading Mail with IMAP

Credit: John Wells

Problem

You want to connect to an IMAP server in order to read and manipulate the messages stored there.

Solution

The `net/imap.rb` package, written by Shugo Maeda, is part of Ruby's standard library, and provides a very capable base on which to build an IMAP-oriented email application. In the following sections, I'll walk you through various ways of using this API to interact with an IMAP server.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

For this recipe, let's assume you have access to an IMAP server running at *mail.myhost.com* on the standard IMAP port 143. Your username is, conveniently, "username", and your password is "password".

To make the initial connection to the server, it's as simple as:

```
require 'net/imap'

conn = Net::IMAP.new('mail.myhost.com', 143)
conn.login('username', 'password')
```

Assuming no error messages were received, you now have a connection to the IMAP server. The `Net::IMAP` object puts all the capabilities of IMAP at your fingertips.

Before doing anything, though, you must tell the server which mailbox you're interested in working with. On most IMAP servers, your default mailbox is called "INBOX". You can change mailboxes with `Net::IMAP#examine`:

```
conn.examine('INBOX')
# Use Net::IMAP#select instead for read-only access
```

A search provides a good example of how a `Net::IMAP` object lets you interact with the server. To search for all messages in the selected mailbox from a particular address, you can use this code:

```
conn.search(['FROM', 'jabba@huttfoundation.org']).each do |sequence|
    fetch_result = conn.fetch(sequence, 'ENVELOPE')
    envelope = fetch_result[0].attr['ENVELOPE']
    printf("%s - From: %s - To: %s - Subject: %s\n", envelope.date,
         envelope.from[0].name, envelope.to[0].name, envelope.subject)
end
# Wed Feb 08 14:07:21 EST 2006 - From: The Hutt Foundation - To: You - Subject: Bwah!
# Wed Feb 08 11:21:19 EST 2006 - From: The Hutt Foundation - To: You - Subject: Go to
# do wa IMAP
```

Discussion

The details of the IMAP protocol are a bit esoteric, and to really understand it you'll need to read the RFC. That said, the code in the solution shouldn't be too hard to understand: it uses the IMAP `SEARCH` command to find all messages with the `FROM` field set to "jabba@huttfoundation.org".

The call to `Net::IMAP#search` returns an array of message sequence IDs: a key to a message within the IMAP server. We iterate over these keys and send each one back to the server, using IMAP's `FETCH` command to ask for the envelope (the headers) of each message. Note that the Ruby method for an IMAP instruction often shares the instruction's name, only in lowercase to keep with the Ruby way.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

The `ENVELOPE` parameter we pass to `Net::IMAP#fetch` tells the server to give us summary information about the message by parsing the RFC2822 message headers. This way we don't have to download the entire body of the message just to look at the headers.

You'll also notice that `Net::IMAP#fetch` returns an array, and that we access its first element to get the information we're after. This is because `Net::IMAP#fetch` lets you to pass an array of sequence numbers instead of just one. It returns an array of `Net::IMAP::FetchData` objects with an element corresponding to each number passed in. You get an array even if you only pass in one sequence number.

There are also other cool things you can do.

Check for new mail

You can see how many new messages have arrived by examining the responses sent by the server when you select a mailbox. These are stored in a hash: the `responses` member of your connection object. Per the IMAP spec, the value of RECENT is the number of new messages unseen by any client. EXISTS tells how many total messages are in the box. Once a client connects and opens the mailbox, the RECENT response will be unset, so you'll only see a new message count the first time you run the command:

```
puts "#{conn.responses["RECENT"]} new messages, #{conn.responses["EXISTS"]} total"
# 10 new messages, 1022 total
```

Retrieve a UID for a particular message

The sequence number is part of a relative sequential numbering of all the messages in the current mailbox. Sequence numbers get reassigned upon message deletion and other operations, so they're not reliable over the long term. The UID is more like a primary key for the message: it is assigned when a message arrives and is guaranteed not to be reassigned or reused for the life of the mailbox. This makes it a more reliable way of making sure you've got the right message:

```
uids = conn.search(["FROM", "jabba@huttfoundation.org"]).collect do |sequence|
  fetch_result = conn.fetch(sequence, "UID")
  puts "UID: #{fetch_result[0].attr["UID"]}"
end
# UID: 203
# UID: 206
```

Why are message UIDs useful? Consider the following scenario. We've just retrieved message information for messages between January 2000 and January 2006. While viewing the output, we saw a message that looked interesting, and noted the UID was 203.

To view the message body, we use code like this:

```
puts conn.uid_fetch(203, 'BODY[TEXT]')[0].attr['BODY[TEXT]']
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Reading headers made easy

In our first example in this recipe, we accessed message headers through use of the IMAP `ENVELOPE` parameter. Because displaying envelope information is such a common task, I prefer to take advantage of Ruby's open classes and add this functionality directly to `Net::IMAP`:

```
class Net::IMAP
  def get_msg_info(msg_sequence_num)
    # code we used above
    fetch_result = fetch(msg_sequence_num, '(UID ENVELOPE)')
    envelope = fetch_result[0].attr['ENVELOPE']
    uid = fetch_result[0].attr['UID']
    info = {'UID'     => uid,
            'Date'    => envelope.date,
            'From'    => envelope.from[0].name,
            'To'      => envelope.to[0].name,
            'Subject' => envelope.subject}
  end
end
```

Now, we can make use of this code wherever it's convenient. For example, in this search for all messages received in a certain date range:

```
conn.search(['BEFORE', '01-Jan-2006',
             'SINCE', '01-Jan-2000']).each do |sequence|
    conn.get_msg_info(sequence).each {|key, val| puts "#{key}: #{val}" }
end
```

Forwarding mail to a cell phone

As a final, somewhat practical example, let's say you're waiting for a very important email from someone at huttfoundation.org. Let's also assume you have an SMTP server at the same host as your IMAP server, running on port 25.

You'd like to have a program that could check your email every five minutes. If a new message from anyone at *huttfoundation.org* is found, you'd like to forward that message to your cell phone via SMS. The email address of your cell phone is 5555555555@mycellphoneprovider.com.

```
#!/usr/bin/ruby -w
# forward_important_messages.rb

require 'net/imap'
require 'net/smtp'

address = 'huttfoundation.org'
from = 'myhomeemail@my.mailhost.com'
to = '5555555555@mycellphoneprovider.com'
smtp_server = 'my.mailhost.com'
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
imap_server = 'my.mailhost.com'
username = 'username'
password = 'password'

while true do
  conn = imap = Net::IMAP.new(imap_server, 143)
  conn.login(username, password)
  conn.select('INBOX')
  uids = conn.search(['FROM', address, 'UNSEEN']).each do |sequence|
      fetch_result = conn.fetch(sequence, 'BODY[TEXT]')
    text = fetch_result[0].attr['BODY[TEXT]']
        count = 1
    while(text.size > 0) do
        # SMS messages limited to 160 characters
      msg = text.slice!(0, 159)
      full_msg = "From: #{from}\n"
      full_msg += "To: #{to}\n"
      full_msg += "Subject: Found message from #{address} (#{count})!\n"
      full_msg += "Date: #{Time.now}\n"
      full_msg += msg + "\n"
      Net::SMTP.start(smtp_server, 25) do |smtp|
          smtp.send_message full_msg, from, to
      end
      count += 1
    end
    # set Seen flag, so our search won't find the message again
    conn.store(sequence, '+FLAGS', [:Seen])
  end
  conn.disconnect
  # Sleep for 5 minutes.
  sleep (60*60*5)
end
```

This recipe should give you a hint of the power you have when you access IMAP mailboxes. Please note that to really understand IMAP, you need to read the IMAP RFC, as well as RFC2822, which describes the Internet Message Format. Multipart messages and MIME types are beyond of the scope of this recipe, but are both something you'll deal with regularly when accessing mailboxes.

See Also

- `ri Net::IMAP`
- The IMAP RFC (RFC3501) (http://www.faqs.org/rfcs/rfc3501.html)
- The Internet Message Format RFC (RFC2822) (http://www.faqs.org/rfcs/rfc2822.html)
- Recipe 3.12, "Running a Code Block Periodically"
- Recipe 14.5, "Sending Mail"

Recipe 14.7. Reading Mail with POP3

Credit: John Wells

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Problem

You want to connect to an POP server in order to read and download the messages stored there.

Solution

The `net/pop.rb` package, written by Minero Aoki, is part of Ruby's standard library, and provides a foundation on which to build a POP (Post Office Protocol)-oriented email application. As with the previous recipe on IMAP, we'll walk through some common ways of accessing a mail server with the POP API.

For this recipe, we assume you have access to a POP3 server running at mail.myhost. com on the standard POP3 port 110. Just as in the previous IMAP example, your username is "username", and password is (yep) "password".

To make the initial connection to the server, it's as simple as:

```
require 'net/pop'

conn = Net::POP3.new('mail.myhost.com')
conn.start('username', 'password')
```

If you receive no errors, you've got an open session to your POP3 server, and can use the `conn` object to communicate with the server.

The following code acts like a typical POP3 client: having connected to the server, it downloads all the new messages, and then deletes them from the server. The deletion is commented out so you don't lose mail accidentally while testing this code:

```
require 'net/pop'

conn = Net::POP3.new('mail.myhost.com')
conn.start('username', 'password')

conn.mails.each do |msg|
   File.open(msg.uidl, 'w') { |f| f.write msg.pop }
   # msg.delete
end

conn.finish
```

Discussion

POP3 is a much simpler protocol than IMAP, and arguably a less powerful one. It doesn't support the concept of folders, so there's no need to start off by selecting a particular folder (like we did in the IMAP recipe). Once you start a session, you have immediate access to all messages currently retained on the server.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

IMAP stores your folders and your messages on the server itself. This way you can access the same messages and the same folders from different clients on different machines. For example, you might go to work and access an IMAP folder with Mozilla Thunderbird, then go home and access the same folder with a web-based mail client.

With POP3, there are no server-side folders. You're supposed to archive your messages on the client side. If you use a POP3 client to download messages at work, when you get home you won't be able to access those messages. They're on your work computer, not on the POP3 server.

IMAP assigns a unique, unchanging ID to each message in the mailbox. By contrast, when you start a POP3 session, POP3 gives each message a "sequence number" reflecting its position in the mailbox at that time. The next time you connect to the POP3 server, the same message may have a different sequence number, as new, incoming messages can affect the sequencing. This is why POP3 clients typically download messages immediately and delete them from the server.

If we want to go outside this basic pattern, and leave the messages on the server, how can we keep track of messages from one connection to another? POP3 does provide a unique string ID for each message: a *Unique Identification Listing*, or UIDL. You can use a UIDL (which persists across POP3 sessions) to get a sequence number (which doesn't) and retrieve a message across separate connections.

This code finds the IDs of email messages from a particular source:

```
conn = Net::POP3.new('mail.myhost.com')
conn.start('username', 'password')
ids = conn.mails.collect {|msg| msg.uidl if msg.pop.match('jabba')}
conn.finish
# => ["UID2-1141260595", "UID3-1141260595"]
```

Now we have unique identifiers for each of our matching messages. Given these, we can start a new POP3 session and use these UIDLs to retrieve each message individually:

```
conn2 = Net::POP3.new('mail.myhost.com')
conn.start('username', 'password')

conn.each_mail {|msg| puts msg.pop if msg.uidl=='UID3-1141260595'}

conn.finish
# Return-Path: <jabba@huttfoundation.org>
# X-Original-To: username@my.mailhost.com
# Delivered-To: username@localhost
# …
```

Here we call the method `Net::POP3#each_mail` to iterate over all the messages in the mailbox. Each message is passed into the code block as a `Net::POPMail` message. We

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

look at each message's UIDL and, when we find the message we want, we call `Net::POPMail#pop` to print it out.

Forwarding mail to a cell phone

Let's revisit our example from the IMAP recipe. You're waiting for a very important email, and you want to have it forwarded to your cell phone as soon as it comes in. You're able to send mail through a SMTP server hosted on port 25 of the same machine as your POP3 server. The email address of your cell phone is 5555555555@mycellphoneprovider.com.

This program checks your POP3 server for new email every five minutes. If a new message from anyone at huttfoundation.org is found, it forwards the message to your cell phone via SMS.

```
#!/usr/bin/env ruby
# forward_important_messages.rb

require 'net/pop'
require 'net/smtp'

$address = 'huttfoundation.org'
$from = 'myhomeemail@my.mailhost.com'
$to = '5555555555@mycellphoneprovider.com'
smtp_server = 'my.mailhost.com'
pop_server = 'my.mailhost.com'
username = 'username'
password = 'password'

$found = Hash.new

def send_msg (text)
  count = 1
  while(text.size > 0) do
    # SMS messages limited to 160 characters
    msg = text.slice!(0, 159)
    full_msg = "From: #{$from}\n"
    full_msg += "To: #{$to}\n"
    full_msg += "Subject: Found message from #{$address} (#{count})!\n"
    full_msg += "Date: #{Time.now}\n"
    full_msg += msg + "\n"
    Net::SMTP.start(smtp_server, 25) do |smtp|
      smtp.send_message full_msg, $from, $to
    end
    count += 1
  end
end

loop do
  conn = Net::POP3.new(pop_server)
  conn.start('username', 'password')

  uidls = conn.mails.collect do |msg|
    msg.uidl if msg.pop.match(/#{$address}/)
  end

  uidls.each do |one_id|
    if ! $found.has_key? one_id
      $found[one_id] = true
      conn.each_mail do |msg|
        send_msg(msg.uidl) if msg.uidl==one_id
      end
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
    end
  end
  conn.finish
  # Sleep for 5 minutes.
  sleep (60*60*5)
end
```

See Also

- Recipe 14.6, "Reading Mail with IMAP"
- RFC1939 describes the POP3 protocol

Recipe 14.8. Being an FTP Client

Problem

You want to automatically connect to an FTP server, and upload or download files.

Solution

Use the `Net::FTP` class. It provides a filesystem-like interface to an FTP server. In this example, I log anonymously into a popular FTP site, browse one of its directories, and download two of its files:

```
require 'net/ftp'
ftp = Net::FTP.open('ftp.ibiblio.org') do |ftp|
  ftp.login
  ftp.chdir('pub/linux/')
  ftp.list('*Linux*') { |file| puts file }
  puts

  puts 'Saving a text file to disk while processing it.'
  ftp.gettextfile('How-do-I-get-Linux') { |line| puts "! #{line}" }
  puts "Saved #{File.size 'How-do-I-get-Linux'} bytes."
  puts

  puts 'Saving a binary file to disk.'
  ftp.getbinaryfile('INDEX.whole.gz')
 puts "Saved #{File.size 'INDEX.whole.gz'} bytes."
end
# -rw-r--r--   1 (?)      users    16979001 Jan 1 11:31 00-find.Linux.gz
# -rw-rw-r--   1 (?)      admin          73 Mar 9  2001 How-do-I-get-Linux

# Saving a text file to disk while processing it.
# !
# !   Browse to http://metalab.unc.edu/linux/HOWTO/Installation-HOWTO.html
# !
# Saved 73 bytes.

# Saving a binary file to disk.
# Saved 213507 bytes.
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Discussion

Once the preferred way of storing and serving files through the Internet, FTP is being largely superceded by SCP for copying files, the web for distributing files, and Bit-Torrent for distributing very large files. There are still many anonymous FTP servers, though, and many web hosting companies still expect you to upload your web pages through FTP.

The `login` method logs in to the server. Calling it without arguments logs you in anonymously, which traditionally limits you to download privileges. Calling it with a username and password logs you in to the server:

```
ftp.login('leonardr', 'mypass')
```

The methods `chdir` and `list` let you navigate the FTP server's directory structure. They work more or less like the Unix `cd` and `ls` commands (in fact, `list` is aliased to `ls` and `dir`).

There are also two "get" methods and two "put" methods. The "get" methods are `getbinaryfile` and `gettextfile`. They retrieve the named file from the FTP server and write it to disk. The `gettextfile` method converts between platform-specific newline formats as it downloads. This way you can download a text file from a Unix server to your Windows machine, and have the Unix newlines automatically converted into Windows newlines. On the other hand, if you use `gettextfile` on a binary file, you'll probably corrupt the file as you download it.

You can specify a local name for the file and a block to process the data as it comes in. A block passed into `gettextfile` will be called for each line of a downloaded file; a block passed into `getbinaryfile` will be passed for each downloaded chunk.

A file you download with one of the "get" methods will be written to disk even if you pass in a block to process it. If you want to process a file *without* writing it to disk, just define some methods like these:

```
class Net::FTP
  def processtextfile(remotefile)
    retrlines('RETR ' + remotefile) { |line| yield line }
  end

  def processbinaryfile(remotefile, blocksize=DEFAULT_BLOCKSIZE)
    retrbinary('RETR ' + remotefile, blocksize) { |data| yield data }
  end
end
```

The two "put" methods are (you guessed it) `puttextfile` and `putbinaryfile`. They are the exact opposites of their `get` counterparts: they take the path to a local file, and write it to a file on the FTP server. They, too, can take a code block that processes each line

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

or chunk of the file as it's read. This example automatically uploads the *index.html* file to my ISP's hosted web space.

```
require 'net/ftp'
Net::FTP.open('myisp.example.com') do |ftp|
  ftp.login('leonardr', 'mypass')
  ftp.chdir('public_html')
  ftp.puttextfile('index.html')
end
```

In general, you can't use the "put" methods if you're logged in as an anonymous user. Some FTP servers do have special *incoming/* directories to which anonymous users can upload their submissions.

See Also

- `ri Net::FTP`

Recipe 14.9. Being a Telnet Client

Problem

You want to connect to a telnet service or use telnet to get low-level access to some other kind of server.

Solution

Use the `Net::Telnet` module in the Ruby standard library.

The following code uses a `Telnet` object to simulate an HTTP client. It sends a raw HTTP request to the web server at http://www.oreilly.com. Every chunk of data received from the web server is passed into a code block, and its size is added to a tally. Eventually the web server stops sending data, and the telnet session times out.

```
require 'net/telnet'

webserver = Net::Telnet::new('Host' => 'www.oreilly.com',
                             'Port' => 80,
                             'Telnetmode' => false)

size = 0
webserver.cmd("GET / HTTP/1.1\nHost: www.oreilly.com\n") do |c|
  size += c.size
  puts "Read #{c.size} bytes; total #{size}"
end
# Read 1431 bytes; total 1431
# Read 1434 bytes; total 2865
# Read 1441 bytes; total 4306
# Read 1436 bytes; total 5742
# …
# Read 1430 bytes; total 39901
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
# Read 2856 bytes; total 42757
# /usr/lib/ruby/1.8/net/telnet.rb:551:in `waitfor':
#   timed out while waiting for more data (Timeout::Error)
```

Discussion

Telnet is a lightweight protocol devised for connecting to a generic service running on another computer. For a long time, the most commonly exposed service was a Unix shell: you would "telnet in" to a machine on the network, log in, and run shell commands on the other machine as though it were local.

Because telnet is an insecure protocol, it's very rare now to use it for remote login. Everyone uses SSH for that instead (see the next recipe). Telnet is still useful for two things:

1. As a diagnostic tool (as seen in the Solution). Telnet is very close to being a generic TCP protocol. If you know, say, HTTP, you can connect to an HTTP server with telnet, send it a raw HTTP request, and view the raw HTTP response.
2. As a client to text-based services other than remote shells: mainly old-school entertainments like BBSes and MUDs.

`Telnet` objects implement a simple loop between you and some TCP server:

1. You send a string to the server.
2. You read data from the server a chunk at a time and process each chunk with a code block. The continues until a chunk of data contains text that matches a regular expression known as a *prompt*.
3. In response to the prompt, you send another string to the server. The loop restarts.

In this example, I script a `Telnet` object to log me in to a telnet-accessible BBS. I wait for the BBS to send me strings that match certain prompts ("What is your name?" and "password:"), and I send back strings of my own in response to the prompts.

```
require 'net/telnet'

bbs = Net::Telnet::new('Host' => 'bbs.example.com')

puts bbs.waitfor(/What is your name\?/)
# The Retro Telnet BBS
# Where it's been 1986 since 1993.
# Dr. Phineas Goodbody, proprietor
#
# What is your name? (NEW for new user)

bbs.cmd('String'=>'leonardr', 'Match'=>/password:/) { |c| puts c }
# Hello, leonardr. Please enter your password:

bbs.cmd('my_password') { |c| puts c }
# Welcome to the Retro Telnet BBS, leonardr.
# Choose from the menu below:
# …
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

The problem with this code is the "prompt" concept was designed for use with remote shells. A Unix shell shows you a prompt after every command you run. The prompt always ends in a dollar sign or some other character: it's easy for telnet to pick out a shell prompt in the data stream. But no one uses telnet for remote shells anymore, so this is not very useful. The BBS software defines a different prompt for every interaction: one prompt for the name and a different one for the password. The web page grabber in the Solution doesn't define a prompt at all, because there's no such thing in HTTP. For the type of problem we still solve with telnet, prompts are a pain.

What's the alternative? Instead of having `cmd` wait for a prompt, you can just have it wait for the server to go silent. Here's an implementation of the web page grabber from the Solution, which stops reading from the server if it ever goes more than a tenth of a second without receiving any data:

```
require 'net/telnet'

webserver = Net::Telnet::new('Host' => 'www.oreilly.com',
                             'Port' => 80,
                             'Waittime' => 0.1,
                             'Prompt' => /.*/,
                             'Telnetmode' => false)
size = 0
webserver.cmd("GET / HTTP/1.1\nHost: www.oreilly.com\n") do |c|
  size += c.size
  puts "Read #{c.size} bytes; total #{size}"
end
```

Here, the prompt matches any string at all. The end of every data chunk is potentially the "prompt" for the next command! But `Telnet` only acts on this if the server sends no more data in the next tenth of a second.

When you have `Telnet` communicate with a server this way, you never know for sure if you really got all the data. It's possible that the server just got really slow all of a sudden. If that happens, you may lose data or it may end up read by your next call to `cmd`. The best you can do is try to make your `Waittime` large enough so that this doesn't happen.

In this example, I use `Telnet` to script a bit of a text adventure game that's been made available over the net. This example uses the same trick (a `Prompt` that matches anything) as the previous one, but I've bumped up the `Waittime` because this server is slower than the oreilly.com web server:

```
require 'net/telnet'
adventure = Net::Telnet::new('Host' => 'games.example.com',
                             'Port' => 23266,
                             'Waittime' => 2.0,
                             'Prompt' => /.*/)

commands = ['no', 'enter building', 'get lamp'] # And so on…
commands.each do |command|
  adventure.cmd(command) { |c| print c }
end
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
# Welcome to Adventure!! Would you like instructions?
# no
#
# You are standing at the end of a road before a small brick building.
# Around you is a forest. A small stream flows out of the building and
# down a gully.
# enter building
#
# You are inside a building, a well house for a large spring.
# There are some keys on the ground here.
# There is a shiny brass lamp nearby.
# There is food here.
# There is a bottle of water here.
#
# get lamp
# OK
```

See Also

- The Ruby documentation for the `net/telnet` standard library
- Recipe 14.10, "Being an SSH Client"
- The telnet text adventure is based on the version of Colossal Cave hosted at forkexec.com; the site has lots of other games you can play via telnet (http://games.forkexec.com/)

Recipe 14.10. Being an SSH Client

Problem

You want to securely send data or commands back and forth between your computer, and another computer on which you have a shell account.

Solution

Use the `Net::SSH` module, which implements the SSH2 protocol. It's found in the `net-ssh` gem, although some operating systems package it themselves.[2] It lets you implement Ruby applications that work like the familiar `ssh` and `scp`.

[2] For instance, it's available on Debian GNU/Linux as the package `libnet-ssh-ruby1.8`.

You can start an SSH session by passing a hostname to `Net::SSH::start`, along with your shell username and password on that host. If you have an SSH public/private key pair set up between your computer and the remote host, you can omit the username and password:

```
require 'rubygems'
require 'net/ssh'
Net::SSH.start('example.com', :username=>'leonardr',
               :password=>'mypass') do |session|
  # Manipulate your Net::SSH::Session object here…
end
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

`Net::SSH::start` takes a code block, to which it passes a `Net::SSH::Session` object. You use the session object to send encrypted data between the machines, or to spawn processes on the remote machine. When the code block ends, the SSH session is automatically terminated.

Discussion

It seems strange now, but until the late 1990s, people routinely used unsecured protocols like telnet to get shell access to remote machines. Remote access was so useful that we were willing to jeopardize our electronic safety by sending our shell passwords (not to mention all the data we looked at) unencrypted across the network. Fortunately, we don't have to make that trade-off anymore. The SSH protocol makes it easy to send encrypted traffic between machines, and the client tools `ssh` and `scp` have almost completely replaced tools like RSH and nonanonymous FTP.

The `Net::SSH` library provides a low-level interface to the SSH2 protocol, but most of the time you won't need it. Instead, you'll use one of the abstractions that make it easy to spawn and control processes on a remote machine. The simplest abstraction is the `popen3` method, which works like the local `popen3` method in Ruby's `open3` library. It's covered in more detail in Recipe 20.10, but here's a simple example:

```
Net::SSH.start('example.com', :username=>'leonardr',
               :password=>'mypass') do |session|
  cmd = 'ls -l /home/leonardr/test_dir'
  session.process.popen3(cmd) do |stdin, stdout, stderr|
    puts stdout.read
  end
end
# -rw-rw-r--    1 leonardr leonardr       33 Dec 29 20:40 file1
# -rw-rw-r--    1 leonardr leonardr      102 Dec 29 20:40 file2
```

You can run a sequence of commands in a single user shell by calling `session.shell.sync`:

```
Net::SSH.start('example.com', :username=>'leonardr',
               :password=>'mypass') do |session|
  shell = session.shell.sync
  puts "Original working directory: #{shell.pwd.stdout}"
  shell.cd 'test_dir'
  puts "Working directory now: #{shell.pwd.stdout}"
  puts 'Directory contents:'
  puts shell.ls("-l").stdout
  shell.exit
end
# Original working directory: /home/leonardr
# Working directory now: /home/leonardr/test_dir
# Directory contents:
# -rw-rw-r--1   leonardr leonardr         33 Dec 29 20:40 file1
# -rw-rw-r--1   leonardr leonardr        102 Dec 29 20:40 file2
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

The main downside of a synchronized shell is that you usually can't pass standard input data into the commands you run. There's no way to close the standard input stream, so the process will hang forever waiting for more standard input.[3] To pass standard input into a remote process, you should use `popen3`. With a little trickery, you can control multiple processes simultaneously through your SSH connection; see Recipe 14.11 for details.

[3] The exception is a command like `bc`, which terminates itself if it sees the line "quit\n" in its standard input. Commands like `cat` always look for more standard input.

If your public/private key pair for a host is protected by a passphrase, you will be prompted for the passphrase `Net::SSH` tries to make a connection to that host. This makes your key more secure, but it will foil your plans to use `Net::SSH` in an automated script.

You can also use `Net::SSH` to do TCP/IP port forwarding. As of this writing, you can't use it to do X11 forwarding.

See Also

- Recipe 20.10, "Controlling a Process on Another Machine," covers `Net:SSH`'s implementation of `popen3` in more detail. Recipe 14.11 shows how to implement an `scp`-like service on top of the `Net:SSH` API, but these three recipes together only scratch the surface of what's possible with `Net:SSH`. The library manual (http://net-ssh.rubyforge.org/) is comprehensive and easy to read; it covers many topics not touched upon here, like low-level SSH2 operations, callback methods other than `on_success`, port forwarding, and nonsynchonized user shells
- Recipe 14.2, "Making an HTTPS Web Request," has information on installing the OpenSSL extension
- Learn more about public/private keys in the article "OpenSSH key management, Part 1" (http://www-128.ibm.com/developerworks/library/l-keyc.html)

Recipe 14.11. Copying a File to Another Machine

Problem

You want to programatically send files to another computer, the way the Unix `scp` command does.

Solution

Use the `Net:SSH` library to get a secure shell connection to the other machine. Start a `cat` process on the other machine, and write the file you want to copy to its standard input.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
require 'rubygems'
require 'net/ssh'

def copy_file(session, source_path, destination_path=nil)
  destination_path ||= source_path
  cmd = %{cat > "#{destination_path.gsub('"', '\"')}"}
  session.process.popen3(cmd) do |i, o, e|
    puts "Copying #{source_path} to #{destination_path}… "
    open(source_path) { |f| i.write(f.read) }
    puts 'Done.'
  end
end

Net::SSH.start('example.com', :username=>'leonardr',
               :password=>'mypass') do |session|
  copy_file(session, '/home/leonardr/scripts/test.rb')
  copy_file(session, '/home/leonardr/scripts/"test".rb')
end
# Copying /home/leonardr/scripts/test.rb to /home/leonardr/scripts/test.rb…
# Done.
# Copying /home/leonardr/scripts/"test".rb to /home/leonardr/scripts/"test".rb…
# Done.
```

Discussion

The `scp` command basically implements the old `rcp` protocol over a secured connection. This code uses a shortcut to achieve the same result: it uses the high-level SSH interface to spawn a process on the remote host which writes data to a file.

Since you can run multiple processes at once over your SSH session, you can copy multiple files simultaneously. For every file you want to copy, you need to spawn a `cat` process:

```
def do_copy(session, source_path, destination_path=nil)
  destination_path ||= source_path
  cmd = %{cat > "#{destination_path.gsub('"', '\"')}"}
  cat_process = session.process.open(cmd)

  cat_process.on_success do |p|
    p.write(open(source_path) { |f| f.read })
    p.close
    puts "Copied #{source_path} to #{destination_path}."
  end
end
```

The call to `session.process.open` creates a process-like object that runs a `cat` command on the remote system. The call to `on_success` registers a callback code block with the process. That code block will run once the `cat` command has been set up and is accepting standard input. Once that happens, it's safe to start writing data to the file on the remote system.

Once you've set up all your copy operations, you should call `session.loop` to perform all the copy operations simultaneously. The processes won't actually be initialized until you call `session.loop`.

```
Net::SSH.start('example.com', :username=>'leonardr',
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
                :password=>'mypass') do |session|
  do_copy(session, '/home/leonardr/scripts/test.rb')
  do_copy(session, '/home/leonardr/new_index.html',
                   '/home/leonardr/public_html/index.html')
  session.loop
end
# Copied /home/leonardr/scripts/test.rb to /home/leonardr/scripts/test.rb
# Copied /home/leonardr/new_index.html to /home/leonardr/public_html/index.html
```

Recipe 14.12. Being a BitTorrent Client

Problem

You want to write a Ruby script that downloads or shares large files with BitTorrent.

Solution

The third-party RubyTorrent library implements the BitTorrent protocol; you can use it to write BitTorrent clients. The RubyTorrent package has no `setup.rb` file, so you'll need to manually copy the files into your Ruby classpath or package them with your application.

The `BitTorrent` class acts as a BitTorrent client, so to download a torrent, all you have to do is give it the path or URL to a *.torrent* file. This code will download the classic B-movie *Night of the Living Dead* to the current working directory:

```
require 'rubytorrent'
file = 'http://publicdomaintorrents.com/bt/btdownload.php?type=torrent' +
       '&file=Night_of_the_Living_Dead.avi.torrent'
client = RubyTorrent::BitTorrent.new(file)
```

Run this in `irb`, keep your session open, and in a few hours (or days), you'll have your movie![4]

[4] That is, assuming the torrent is still active when you read this. Incidentally, *Night of the Living Dead* is in the public domain because of a mishap regarding the copyright notice.

Discussion

BitTorrent is the most efficient way yet devised for sharing large files between lots of people. As you download the file you're also sharing what you've downloaded with others: the more people are trying to download the file, the faster it is for everyone.

RubyTorrent is a simple client library to the BitTorrent protocol. In its simplest form, you simply construct a `BitTorrent` object with the URL or path to a torrent information file, and wait for the download to complete. However, there's a lot more you can do to provide a better user interface.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

The `BitTorrent` object has several methods that let you keep track of the progress of the download:

```
client.num_active_peers                     # => 9
# That is, 9 other people are downloading this file along with me.

client.ulrate                               # => 517.638825414351
client.dlrate                               # => 17532.608916979
# That is, about 3 kb/sec uploading and 17 kb/sec downloading.

client.percent_completed                    # => 0.25
```

You can also register code blocks to be run at certain points in the client's lifecycle. Here's a more advanced BitTorrent client that registers code blocks to let the user know about new and dropped peer connections. It also uses a thread to occasionally report on the progress of the download. The user can specify which port to use when uploading data to peers, and a maximum upload rate in kilobytes.

```
#!/usr/bin/ruby
# btclient.rb
require 'rubytorrent'

def download(torrent, destination=nil, local_port=6881, max_ul=40)
  client = RubyTorrent::BitTorrent.new(torrent, destination,
                                       :port => local_port,
                                       :ulratelim => max_ul * 1024)

  thread = Thread.new do
    until client.complete?
      if client.tracker
        puts '%s: %dk of %dk (%.2f%% complete)' % [Time.now,
          client.bytes_completed / 1024, client.total_bytes / 1024,
          client.percent_completed]
        sleep(60)
      else
        sleep(5)
      end
    end
  end

client.on_event(self, :tracker_connected) do |src, url|
  puts "[Connected to tracker at #{url}]"
  end
  client.on_event(self, :added_peer) do |src, peer|
    puts "[Connected to #{peer}.]"
  end
  client.on_event(self, :removed_peer) do |src, peer|
    puts "[Lost connection to #{peer.name}.]"
  end
  client.on_event(self, :complete) do
    puts 'Download complete.'
    thread.kill
    client.shutdown
  end

  thread.join
end

download(*ARGV)
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

See Also

- Get RubyTorrent at http://rubytorrent.rubyforge.org/; see especially the API reference at http://rubytorrent.rubyforge.org/api.txt
- The `btpeer.rb` and `rtpeer-ncurses.rb` files in the RubyTorrent package provide more in-depth client examples
- A few sources for interesting BitTorrent files:
  - http://www.publicdomaintorrents.com/
  - http://torrent.ibiblio.org/

Recipe 14.13. Pinging a Machine

Problem

You want to check whether a particular machine or domain name can be reached from your computer.

Solution

Use Ruby's standard `ping` library. Its single method, `Ping.pingecho`, tries to get some machine on the network to respond to its entreaties. It takes either a domain name or an IP address, and returns true if it gets a response.

```
require 'ping'

ping.pingecho('oreilly.com')                              # => true

# timeout of 10 seconds instead of the default 5 seconds
Ping.pingecho('127.0.0.1', 10)                          # => true
# ping port 80 instead of the default echo port
Ping.pingecho('slashdot.org', 5, 80)                    # => true

Ping.pingecho('no.such.domain')                         # => false
Ping.pingecho('222.222.222.222')                        # => false
```

Discussion

`Ping.pingecho` performs a TCP echo: it tries to make a TCP connection to the given machine, and if the machine responds (even if to refuse the connection) it means the machine was reachable.

This is not the ICMP echo of the Unix `ping` command, but the difference almost never matters. If you absolutely need an ICMP echo, you can invoke `ping` with a system call and check the return value:

```
system('ping -c1 www.oreilly.com')
# 64 bytes from 208.201.239.36: icmp_seq=0 ttl=42 time=27.2 ms
```

```
#
# --- www.oreilly.com ping statistics --
# 1 packets transmitted, 1 packets received, 0% packet loss
# round-trip min/avg/max = 27.2/27.2/27.2 ms
# => true
```

If the domain has a DNS entry but can't be reached, `Ping::pingecho` may raise a `Timeout::Error` instead of returning false.

Some very popular or very paranoid domains, such as microsoft.com, don't respond to incoming ping requests. However, you can usually access the web server or some other service on the domain. You can see whether such a domain is reachable by using one of Ruby's other libraries:

```
ping.pingecho('microsoft.com')                          # => false

require 'net/http'
Net::HTTP.start('microsoft.com') { 'success!' }         # => "success!"
Net::HTTP.start('no.such.domain') { "success!" }
# SocketError: getaddrinfo: Name or service not known
```

Recipe 14.14. Writing an Internet Server

Problem

You want to run a server for a TCP/IP application-level protocol, but no one has written a Ruby server for the protocol yet. This may be because it's a protocol you've made up.

Solution

Use the `gserver` library in Ruby's standard library. It implements a generic TCP/IP server suitable for small to medium-sized tasks.

Here's a very simple chat server written with gserver. It has no end-user features to speak of. People connect to the server with a telnet client, and are identified to each other only by hostname. But it's a fully functional, multithreaded, logging server written in about 30 lines of Ruby.

```
#!/usr/bin/ruby -w
# chat.rb
require 'gserver'

class ChatServer < GServer

  def initialize(port=20606, host=GServer::DEFAULT_HOST)
    @clients = []
    super(port, host, Float::MAX, $stderr, true)
  end

  def serve(sock)
    begin
      @clients << sock
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
      hostname = sock.peeraddr[2] || sock.peeraddr[3]
      @clients.each do |c|
        c.puts "#{hostname} has joined the chat." unless c == sock
      end
      until sock.eof? do
        message = sock.gets.chomp
        break if message == "/quit"
        @clients.each { |c| c.puts "#{hostname}: #{message}" unless c == sock }
      end
    ensure
      @clients.delete(sock)
      @clients.each { |c| c.puts "#{hostname} has left the chat." }
    end
  end
end

server = ChatServer.new(*ARGV[0..2] || 20606)
server.start(-1)
server.join
```

Start the server in a Ruby session, and then use several instances of the telnet program to connect to port 20606 (from several different hosts, if you can). Your telnet sessions will be able to communicate with each other through the server. Your Ruby session will see a log of the connections and disconnections.

Discussion

The `GServer` class wraps Ruby's underlying `TCPServer` class in a loop that continually receives TCP connections and spawns new threads to process them. Each new thread passes its TCP connection (a `TCPSocket` object) into the `GServer#serve` method, which your subclass is responsible for providing.

The `TCPSocket` works like a bidirectional file. Writing to it pushes data to the client, and reading from it reads data from the client. A server like the sample chat server reads one line at a time from the client; a web server would read the entire request before sending back any data.

In the chat server example, the server echoes one client's input to all the others. In most applications, the client sockets won't even know about each other (think a web or FTP server).

The `GServer` constructor deserves a closer look. Here's its signature, from `gserver.rb`:

```
def initialize(port, host = DEFAULT_HOST, maxConnections = 4,
    stdlog = $stderr, audit = false, debug = false)
```

The port and host should be familiar to you from other types of server. `maxConnections` controls the maximum number of clients that can connect to the server at once. Because a chat server is very high-latency, I set the number effectively to infinity in `ChatServer`.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

`stdlog` is an `IO` object to be used as a log. You can write a timestamped entry to the log by calling `GServer#log`. Setting `audit` to true turns on some default log messages: these are displayed, for instance, whenever a client connects to or disconnects from the server. Finally, setting `debug` to `true` means that, if your code throws an exception, the exception object will be passed into `GServer#error`. You can override this method to do your own error handling.

`Gserver` is easy to use, but not as efficient as a Ruby Internet server could be. For high-performance servers, you'll want to use `IO.select` and `TCPServer` objects, programming to the C sockets API.

See Also

- `ri GServer`

Recipe 14.15. Parsing URLs

Problem

You want to parse a string representation of a URL into a data structure that articulates the parts of the URL.

Solution

`URI.parse` transforms a string describing a URL into a `URI` object.[5] The parts of the URL can be determined by interrogating the `URI` object.

[5] The class name is `URI`, but I use both "URI" and "URL" because they are more or less interchangeable.

```
require 'uri'

URI.parse('https://www.example.com').scheme           # => "https"
URI.parse('http://www.example.com/').host             # => "www.example.com"
URI.parse('http://www.example.com:6060/').port        # => 6060
URI.parse('http://example.com/a/file.html').path      # => "/a/file.html"
```

`URI.split` transforms a string into an array of URL parts. This is more efficient than `URI.parse`, but you have to know which parts correspond to which slots in the array:

```
URI.split('http://example.com/a/file.html')
# => ["http", nil, "example.com", nil, nil, "/a/file.html", nil, nil, nil]
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Discussion

The `URI` module contains classes for five of the most popular URI schemas. Each one can store in a structured format the data that makes up a URI for that schema. `URI.parse` creates an instance of the appropriate class for a particular URL's scheme.

Every URI can be decomposed into a set of *components*, joined by constant strings. For example: the components for a HTTP URI are the scheme ("http"), the hostname ("www.example.com (http://www.example.com)"), and so on. Each URI schema has its own components, and each of Ruby's `URI` classes stores the names of its components in an ordered array of symbols, called `component`:

```
URI::HTTP.component
# => [:scheme, :userinfo, :host, :port, :path, :query, :fragment]

URI::MailTo.component
# => [:scheme, :to, :headers]
```

Each of the components of a `URI` class has a corresponding accessor method, which you can call to get one component of a URI. You can also instantiate a URI class directly (rather than going through `URI.parse`) by passing in the appropriate component symbols as a map of keyword arguments.

```
URI::HTTP.build(:host => 'example.com', :path => '/a/file.html',
                :fragment => 'section_3').to_s
# => "http://example.com/a/file.html#section_3"
```

The following debugging method iterates over the components handled by the scheme of a given URI object, and prints the corresponding values:

```
class URI::Generic
  def dump
    component.each do |m|
      puts "#{m}: #{send(m).inspect}"
    end
  end
end
```

`URI::HTTP` and `URI::HTTPS` are the most commonly encountered subclasses of `URI`, since most URIs are the URLs to web pages. Both classes provide the same interface.

```
url = 'http://leonardr:pw@www.subdomain.example.com:6060' +
    '/cgi-bin/mycgi.cgi?key1=val1#anchor'
URI.parse(url).dump
# scheme: "http"
# userinfo: "leonardr:pw"
# host: "www.subdomain.example.com"
# port: 6060
# path: "/cgi-bin/mycgi.cgi"
# query: "key1=val1"
# fragment: "anchor"
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

A URI::FTP object represents an FTP server, or a path to a file on an FTP server. The typecode component indicates whether the file in question is text, binary, or a directory; it typically won't be known unless you create a URI::FTP object and specify one.

```
URI::parse('ftp://leonardr:password@ftp.example.com/a/file.txt').dump
# scheme: "ftp"
# userinfo: "leonardr:password"
# host: "ftp.example.com"
# port: 21
# path: "/a/file.txt"
# typecode: nil
```

A URI::Mailto represents an email address, or even an entire message to be sent to that address. In addition to its component array, this class provides a method (to_mailtext) that formats the URI as an email message.

```
uri = URI::parse('mailto:leonardr@example.com?Subject=Hello&body=Hi!')
uri.dump
# scheme: "mailto"
# to: "leonardr@example.com"
# headers: [["Subject", "Hello"], ["body", "Hi!"]]

puts uri.to_mailtext
# To: leonardr@example.com
# Subject: Hello
#
# Hi!
```

A URI::LDAP object contains a path to an LDAP server or a query against one:

```
URI::parse("ldap://ldap.example.com").dump
# scheme: "ldap"
# host: "ldap.example.com"
# port: 389
# dn: nil
# attributes: nil
# scope: nil
# filter: nil
# extensions: nil

URI::parse('ldap://ldap.example.com/o=Alice%20Exeter,c=US?extension').dump
# scheme: "ldap"
# host: "ldap.example.com"
# port: 389
# dn: "o=Alice%20Exeter,c=US"
# attributes: "extension"
# scope: nil
# filter: nil
# extensions: nil
```

The URI::Generic class, superclass of all of the above, is a catch-all class that holds URIs with other schemes, or with no scheme at all. It holds much the same components as URI::HTTP, although there's no guarantee that any of them will be non-nil for a given URI::Generic object.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

`URI::Generic` also exposes two other `components` not used by any of its built-in subclasses. The first is `opaque`, which is the portion of a URL that couldn't be parsed (that is, everything after the scheme):

```
uri = URI.parse('tag:example.com,2006,my-tag')
uri.scheme                                    # => "tag"
uri.opaque                                    # => "example.com,2006,my-tag"
```

The second is `registry`, which is only used for URI schemes whose naming authority is registry-based instead of server-based. It's likely that you'll never need to use `registry`, since almost all URI schemes are server-based (for instance, HTTP, FTP, and LDAP all use the DNS system to designate a host).

To combine the `components` of a URI object into a string, simply call `to_s`:

```
uri = URI.parse('http://www.example.com/#anchor')
uri.port = 8080
uri.to_s                              # => "http://www.example.com:8080/#anchor"
```

See Also

- Recipe 11.13, "Extracting All the URLs from an HTML Document"
- `ri URI`

Recipe 14.16. Writing a CGI Script

Credit: Chetan Patil

Problem

You want to expose Ruby code through an existing web server, without having to do any special configuration.

Solution

Most web servers are set up to run CGI scripts, and it's easy to write CGI scripts in Ruby. Here's a simple CGI script that calls the Unix command `ps`, parses its results, and outputs the list of running processes as an HTML document.[6] Anyone with access to the web server can then look at the processes running on the system.

[6] On Windows, you could do this example by running some other command such as `dir`, listing the running Windows services as seen in Recipe 23.2, or just printing a static message.

```
#!/usr/bin/ruby
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
# ps.cgi

processes = %x{ps aux}.collect do |proc|
  '<tr><td>' + proc.split(/\s+/, 11).join('</td><td>') + '</td></tr>'
end

puts 'Content-Type: text/html'
# Output other HTTP headers here…
puts "\n"

title = %{Processes running on #{ENV['SERVER_NAME'] || `hostname`.strip}}
puts <<-end
  <HTML>
   <HEAD><TITLE>#{title}</TITLE></HEAD>
   <BODY>
   <H1>#{title}</H1>
   <TABLE>
    #{processes.join("\n")}
  </TABLE>
  </BODY>
  </HTML>
end

exit 0
```

Discussion

CGI was the first major technology to add dynamic elements to the previously static Web. A CGI resource is requested like any static HTML document, but behind the scenes the web server executes an external program (in this case, a Ruby script) instead of serving a file. The output of the program—text, HTML, or binary data—is sent as part of the HTTP response to the browser.

CGI has a very simple interface, based on environment variables and standard input and output; one that should be very familiar to writers of command-line programs. This simplicity is CGI's weakness: it leaves too many things undefined. But when a Rails application would be overkill, a CGI script might be the right size.

CGI programs typically reside in a special directory of the web server's web space (often the `/cgi-bin` directory). On Unix systems, CGI files must be made executable by the web server, and the first line of the script must point to the system's Ruby interpreter (usually `/usr/bin/ruby` or `/usr/local/bin/ruby`).

A CGI script gets most of its input from environment variables like `QUERY_STRING` and `PATH_INFO`, which are set by the web server. The web server also uses environment variables to tell the script where and how it's being run: note how the sample script uses `ENV['SERVER_NAME']` to find the machine's hostname for display.

There are only a few restrictions on the output of a CGI script. Before the "real" output, you need to send some HTTP headers. The only required header is `Content-Type`, which tells the browser what MIME type to expect from the document the CGI is going to output.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

This is also your chance to set other HTTP headers, such as Contentlength, Expires, Location, Pragma, and Status.

The headers are separated from the content by a blank line. If the blank line is missing, the server may incorrectly interpret the entire data stream as a HTTP header—a leading cause of errors. Other possible problems include:

- The first line of the file contains the wrong path to the Ruby executable.
- The permissions on the CGI script don't allow the web server to access or execute it.
- You used binary mode FTP to upload the script to your server from another platform, and the server doesn't understand that platform's line endings: use text mode FTP instead.
- The web server is not configured to run Ruby scripts as CGI, or to run CGI scripts at all.
- The script contains a compile error. Try running it manually from the command line.

If you get the dreaded error "premature end of script headers" from your web server, these issues are the first things to check.

Newer versions of Ruby include the CGI support library `cgi`. Except for extremely simple CGIs, it's better to use this library than to simply write HTML to standard output. The `CGI` class makes it easy to retrieve HTTP request parameters and to manage cookies. It also provides custom methods for generating HTML, using Ruby code that has the same structure as the eventual output.

Here's the code from `ps.cgi`, rewritten to use the `CGI` class. Instead of writing HTML, we make the `CGI` class do it. `CGI` also takes care of the content type, since we're using the default (`text/html`).

```
#!/usr/bin/ruby
# ps2.cgi

require 'cgi'

# New CGI object
cgi = CGI.new('html3')
processes = `ps aux`.collect { |proc| proc.split(/\s+/, 11) }

title = %{Processes running on #{ENV['SERVER_NAME'] || %x{hostname}.strip}}

cgi.out do
  cgi.html do
    cgi.head { cgi.title { title } } + cgi.body do
      cgi.table do
        (processes.collect do |fields|
          cgi.tr { fields.collect { |field| cgi.td { field } }.join " " }
        end).join "\n"
      end
    end
  end
end

exit 0
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Since CGI allows any user to execute an external CGI program on your web server, security is of paramount importance. Popular CGI hacks include corrupting the program's input by inserting special characters in the `QUERY_STRING`, stealing confidential user data by modifying the parameters posted to the CGI program, and launching denial-of-service attacks to render the web server inoperable. CGI programs need to be carefully inspected for possible bugs and exploits. A few simple techniques will improve your security: call `taint` on external data, set your `$SAFE` variable to 1 or higher, and don't use methods like `eval, system`, or `popen` unless you have to.

See Also

- The CGI documentation (http://hoohoo.ncsa.uiuc.edu/cgi/), especially the list of environment variables (http://hoohoo.ncsa.uiuc.edu/cgi/env.html)
- Recipe 14.17, "Setting Cookies and Other HTTP Response Headers"
- Recipe 14.18, "Handling File Uploads via CGI"
- Chapter 15

Recipe 14.17. Setting Cookies and Other HTTP Response Headers

Credit: Mauro Cicio

Problem

You're writing a CGI program and you want to customize the HTTP headers you send in response to a request. For instance, you may want to set a client-side cookie so that you can track state between HTTP requests.

Solution

Pass a hash of headers into the `CGI#out` method that creates the HTTP response. Each key of the hash is the name of a header to set, or a special value (like `cookie`), which the `CGI` class knows how to interpret.

Here's a CGI script that demonstrates how to set some response headers, including a cookie and a custom HTTP header called "Recipe Name".

First we process any incoming cookie. Every time you hit this CGI, the value stored in your cookie will be incremented, and the date of your last visit will be reset.

```
#!/usr/bin/ruby
# headers.cgi
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
require "cgi"
cgi = CGI.new("html3")

# Retrieve or create the "rubycookbook" cookie
cookie = cgi.cookies['rubycookbook']
cookie = CGI::Cookie.new('rubycookbook', 'hits=0',
                         "last=#{Time.now}") if cookie.empty?

# Read the values in the cookie for future use
hits = cookie.value[0].split('=')[1]
last = cookie.value[1].split('=')[1]

# Set new values in the cookie
cookie.value[0] = "hits=#{hits.succ}"
cookie.value[1] = "last=#{Time.now}"
```

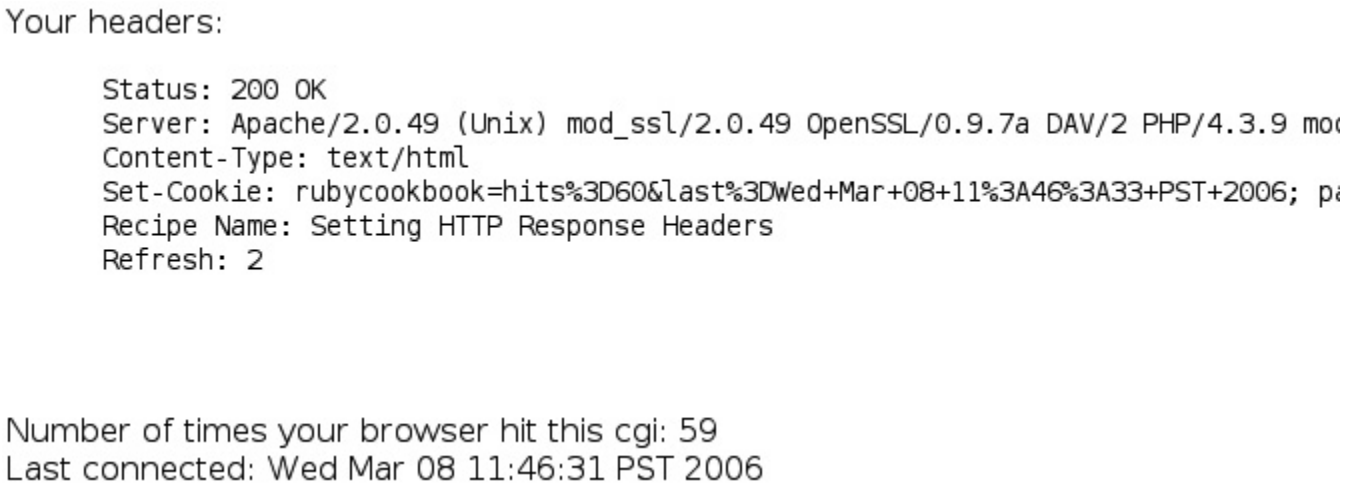
Next, we build a hash of HTTP headers, and send the headers by passing the hash into `CGI#out`. We then generate the output document. Since the end user doesn't usually see the HTTP headers they're served, we'll make them visible by repeating them in the output document (Figure 14-1):

```
# Create a hash of HTTP response headers.
header = { 'status'      => 'OK',
           'cookie'      => [cookie],
           'Refresh'     => 2,
           'Recipe Name' => 'Setting HTTP Response Headers',
           'server'      => ENV['SERVER_SOFTWARE'] }

cgi.out(header) do
  cgi.html('PRETTY' => ' ') do
    cgi.head { cgi.title { 'Setting HTTP Response Headers' } } +
    cgi.body do
      cgi.p('Your headers:') +
      cgi.pre{ cgi.header(header) } +
      cgi.pre do
        "Number of times your browser hit this cgi: #{hits}\n"+
        "Last connected: #{last}"
      end
    end
 end
end
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Figure 14-1. This CGI lets you see the response headers, including the cookie



The `Refresh` header makes your web browser refresh the page every two seconds. You can visit this CGI once and watch the number of hits (stored in the client-side cookie) start to mount up.

Discussion

An HTTP Response consists of two sections (a header section and a body section) separated by a blank line. The body contains the document to be rendered by the browser (usually an HTML page) and the header carries metadata: information about the connection, the response, and the document itself. The `CGI#out` method takes a hash representing the HTTP headers, and a code block that generates the body.

`CGI#out` recognizes a few special values that make it easier to set custom headers. For instance, the header hash in the example above maps the key "cookie" to a `CGI::Cookie` object. `CGI#out` knows enough to turn `cookie` into the standard HTTP header `Set-Cookie`, and to transform the `CGI::Cookie` object into a string rendition.

If `CGI#out` doesn't know about a certain key, it simply sends it as an HTTP header, as-is. `CGI#out` has no special knowledge of our "Refresh" and "Recipe Name" headers, so it writes them verbatim to the HTTP response. "Refresh" is a standard HTTP response header recognized by most web browsers; "Recipe Name" is a header I made up for this recipe, and web browsers should ignore it.

See Also

- The CGI documentation (http://www.ruby-doc.org/core/classes/CGI.html), especially the list of recognized header keys and status codes

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Recipe 14.18. Handling File Uploads via CGI

Credit: Mauro Cicio

Problem

You want to let a visitor to your web site upload a file to the web server, either for storage or processing.

Solution

The `CGI` class provides a simple interface for accessing data sent through HTTP file upload. You can access an uploaded file through `CGI#params` as though it were any other CGI form variable.

If the uploaded file size is smaller than 10 kilobytes, its contents are made available as a `StringIO` object. Otherwise, the file is put into a `Tempfile` on disk: you can read the file from disk and process it, or move it to a permanent location.

Here's a CGI that accepts file uploads and saves the files to a special directory on disk:

```
#!/usr/bin/ruby
# upload.rb

# Save uploaded files to this directory
UPLOAD_DIR = "/usr/local/www/uploads"

require 'cgi'
require 'stringio'
```

The CGI has two main parts: a method that prints a file upload form and a method that processes the results of the form. The method that prints the form is very simple:

```
def display_form(cgi)
 action = env['script_name']
 return <<EOF
<form action="#{action}" method="post" enctype="multipart/form-data">
 File to Upload: <input type="file" name="file_name"><br>
 Your email address: <input type="text" name="email_address"
                      value="guest@example.com"><br>
 <input type="submit" name="Submit" value="Submit Form">
 </form>
EOF
end
```

The method that processes the form is a little more complex:

```
def process_form(cgi)
  email = cgi.params['email_address'][0]
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
  fileObj = cgi.params['file_name'][0]

  str = '<h1>Upload report</h1>' +
    "<p>Thanks for your upload, #{email.read}</p>"
  if fileObj
    path = fileObj.original_filename
    str += "Original Filename : #{path}" + cgi.br
    dest = File.join(UPLOAD_DIR, sanitize_filename(path))

    str += "Destination : #{dest} <br>"
    File.open(dest.untaint, 'wb') { |f| f << fileObj.read }

    # Delete the temporary file if one was created
    local_temp_file = fileObj.local_path()
    File.unlink(local_temp_file) if local_temp_file
  end
  return str
end
```

The `process_form` method calls a method `sanitize_filename` to pick a new filename based on the original. The new filename is stripped of characters in the upload file's name that aren't valid on the server's filesystem. This is important for security reasons. It's also important to pick a new name because Internet Explorer on Windows submits filenames like "c:\hot\fondue.txt" where other browsers would submit "fondue.txt". We'll define that method now:

```
def sanitize_filename(path)
  if RUBY_PLATFORM =~ %r{unix|linux|solaris|freebsd}
    # Not required for unix platforms since all characters
    # are allowed (except for /, which is stripped out below).
  elsif RUBY_PLATFORM =~ %r{win32}
    # Replace illegal characters for NTFS with _
    path.gsub!(/[\x00-\x1f\/|?*]/,'_')
  else
    # Assume a very restrictive OS such as MSDOS
    path.gsub!(/[\/|\?*+\]\[ \x00-\x1fa-z]/,'_')
  end

  # For files uploaded by Windows users, strip off the beginning path.
  return path.gsub(/^.*[\\\/]/, '')
end
```

Finally we have the CGI code itself, which calls the appropriate method and prints out the results in an HTML page:

```
cgi = CGI.new('html3')
if cgi.request_method !~ %r{POST}
  buf = display_form(cgi)
else
  buf = process_form(cgi)
end
cgi.out() do
  cgi.html() do
    cgi.head{ cgi.title{'Upload Form'} } + cgi.body() { buf }
  end
end

exit 0
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Discussion

This CGI script presents the user with a form that lets them choose a file from their local system to upload. When the form is POSTed, `CGI` accepts the uploaded file data and stores it as a CGI parameters. As with any other CGI parameter (like `email_address`), the uploaded file is keyed off of the name of the HTML form element: in this case, `file_name`.

If the file is larger than 10 kilobytes, it will be written to a temporary file and the contents of `CGI[:file_name]` will be a `Tempfile` object. If the file is small, it will be kept directly in memory as a `StringIO` object. Either way, the object will have a few methods not found in normal `Tempfile` or `StringIO` objects. The most useful of these are `original_filename`, `content_type`, and `read`.

The `original_filename` method returns the name of the file, as seen on the computer of the user who uploaded it. The `content_type` method returns the MIME type of the uploaded file, again as estimated by the computer that did the upload. You can use this to restrict the types of file you'll accept as uploads (note, however, that a custom client can lie about the content type):

```
# Limit uploads to BMP files.
raise 'Wrong type!' unless fileObj.content_type =~ %r{image/bmp}
```

Every `StringIO` object supports a `read` method that simply returns the contents of the underlying string. For the sake of a uniform interface, a `Tempfile` object created by file upload also has a `read` method that returns the contents of a file. For most applications, you don't need to check whether you've got a `StringIO` or a `Tempfile`: you can just call `read` and get the data. However, a `Tempfile` can be quite large—there's a reason it was written to disk in the first place—so don't do this unless you trust your users or have a lot of memory. Otherwise, check the size of a `Tempfile` with `File.size` and read it a block at a time.

To see where a `Tempfile` is located on disk, call its `local_path` method. If you plan to write the uploaded file to disk, it's more efficient to move a `Tempfile` with `FileUtils.mv` than to read it into memory and immediately write it back to another location.

Temporary files are deleted when the Ruby interpreter exits, but some web frameworks keep a single Ruby interpreter around indefinitely. If you're not careful, a long-running application can fill up your disk or partition with old temporary files. Within a CGI script, you should explicitly delete temporary files when you're done with them—except, of course, the ones you move to permanent positions elsewhere on the filesystem.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

See Also

- RFC1867 describes HTTP file upload
- For more on the `StringIO` and `Tempfile` classes used to store uploaded files, see Recipe 6.8, "Writing to a Temporary File," and Recipe 6.15, "Pretending a String Is a File"
- http://wiki.rubyonrails.com/rails/pages/HowtoUploadFiles

Recipe 14.19. Running Servlets with WEBrick

Credit: John-Mason Shackelford

Problem

You want to embed a server in your Ruby application. Your project is not a traditional web application, or else it's too small to justify the use of a framework like Rails or Nitro.

Solution

Write a custom servlet for WEBrick, a web server implemented in Ruby and included in the standard library.[7]

[7] Don't confuse WEBrick servlets with Java servlets. The concepts are similar, but they don't implement the same API.

Configure `WEBrick` by creating a new `HTTPServer` instance and mouting servlets. The default `FileHandler` acts like a "normal" web server: it serves a URL-space corresponding to a directory on disk. It delegates requests for `*.cgi` files to the `CGIHandler`, renders `*.rhtml` files with `ERb` using the `ERBHandler` servlet, and serves other files (such as static HTML files) as they are.

This server mounts three servlets on a server running on port 8000 on your local machine. Each servlet serves documents, CGI scripts, and `.rhtml` templates from a different directory on disk:

```
#!/usr/bin/ruby
# simple_servlet_server.rb
require 'webrick'
include WEBrick

s = HTTPServer.new(:Port => 8000)
# Add a mime type for *.rhtml files
HTTPUtils::DefaultMimeTypes.store('rhtml', 'text/html')

# Required for CGI on Windows; unnecessary on Unix/Linux
s.config.store( :CGIInterpreter, "#{HTTPServlet::CGIHandler::Ruby}")
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
# Mount servlets
s.mount('/',       HTTPServlet::FileHandler, '/var/www/html')
s.mount('/bruce',  HTTPServlet::FileHandler, '/home/dibbbr/htdoc')
s.mount('/marty',  HTTPServlet::FileHandler, '/home/wisema/htdoc')

# Trap signals so as to shutdown cleanly.
['TERM', 'INT'].each do |signal|
  trap(signal){ s.shutdown }
end

# Start the server and block on input.
s.start
```

Discussion

WEBrick is robust, mature, and easy to extend. Beyond serving static HTML pages, `WEBrick` supports traditional CGI scripts, `ERb`-based templating like PHP or JSP, and custom servlet classes. While most of WEBrick's API is oriented toward responding to HTTP requests, you can also use it to implement servers that speak another protocol. (For more on this capability, see the Daytime server example on the WEBrick home page.)

The first two arguments to `HTTPServer#mount` (the mount directory and servlet class) are used by the `mount` method itself; any additional arguments are simply passed along to the servlet. This way, you can configure a servlet while you mount it; the `FileHandler` servlet requires an argument telling it which directory on disk contains the web content.

When a client requests a URL, WEBrick tries to match it against the entries in its mounting table. The mounting order is irrelevant. Where multiple mount locations might apply to a single directory, WEBrick picks the longest match.

When the request is for a directory (like *http://localhost/bruce/*), the server looks for the files *index.html, index.htm, index.cgi*, or *index.rhtml*. This is configurable via the `:DirectoryIndex` configuration parameter. The snippet below adds another file to the list of directory index files:

```
s.config.store(:DirectoryIndex,
                  s.config[:DirectoryIndex] << "default.htm")
```

When the standard handlers provided by `WEBrick` won't work for you, write a custom servlet. Rubyists have written custom `WEBrick` servlets to handle SOAP and XML-RPC services, implement a WebDAV server, process `eruby` templates instead of `ERb` templates, and fork processes to distribute load on machines with multiple CPUs.

To write your own `WEBrick` servlet, simply subclass `HTTPServlet::AbstractServlet` and write `do_` methods corresponding to the HTTP methods you wish to handle. Then mount your servlet class as shown in the Solution. The

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

following example handles `HTTP GET` requests via the `do_GET` method, and `POSTs` via an alias. `HEAD` and `OPTIONS` requests are implemented in the `AbstractServlet` itself.

```
#!/usr/bin/ruby
# custom_servlet_server.rb
require 'webrick'
include WEBrick

class CustomServlet < HTTPServlet::AbstractServlet
  def do_GET(request, response)
    response.status = 200 # Success
    response.body = "Hello World"
    response['Content-Type'] = 'text/plain'
  end

  # Respond with an HTTP POST just as we do for the HTTP GET.
  alias :do_POST :do_GET
end

# Mount servlets.
s = HTTPServer.new(:Port => 8001 )
s.mount('/tricia', CustomServlet )

# Trap signals so as to shutdown cleanly.
['TERM', 'INT'].each do |signal|
  trap(signal){ s.shutdown }
end

# Start the server and block on input.
s.start
```

Start that server, visit *http://localhost:8001/tricia/*, and you'll see the string "Hello World".

Beyond defining handlers for arbitrary HTTP methods and configuring custom servlets with mount options, we can also control how often servlet instances are initialized. Ordinarily, a new servlet instance is instantiated for every request. Since each request has its own instance of the servlet class, you are free to write custom servlets without worrying about the servlet's state and thread safety (unless, of course, you share resources between servlet instances).

But you can get faster request handling—at the expense of a slower startup time—by moving some work out of the `do_` methods and into the sevlet's `initialize` method. Instead of creating a new servlet instance with every request, you can override the class method `HTTPServlet::AbstractServlet.get_instance` and manage a pool of servlet instances. This works especially well when your request handling methods are reentrant, so that you can avoid cost costly thread synchronization.

The following example uses code from Recipe 12.13 to serve up a certificate of completion to the individual named by the HTTP request. We use the templating approach discussed in the PDF recipe to prepare most of the certificate ahead of time. During request handling, we do nothing but fill in the recipient's name.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

The `PooledServlet` class below does the work of pooling the servlet handlers:

```
#!/usr/bin/ruby
# certificate_server.rb
require 'webrick'
require 'thread'
require 'cgi'

include WEBrick

class PooledServlet < HTTPServlet::AbstractServlet

  INIT_MUTEX = Mutex.new
  SERVLET_POOL = []

  @@pool_size = 2

  # Create a single instance of the servlet to avoid repeating the costly
  # initialization.
  def self.get_instance(config, *options)
    unless SERVLET_POOL.size == @@pool_size
      INIT_MUTEX.synchronize do
        SERVLET_POOL.clear
        @@pool_size.times{ SERVLET_POOL << new( config, *options ) }
      end
    end
    s = SERVLET_POOL.find{|s| ! s.busy?} while s.nil?
    return s
  end

  def self.pool_size( size )
    @@pool_size = size
  end

  def busy?
    @busy
  end

  def service(req, res)
    @busy = true
    super
    @busy = false
  end
end
```

Note that by placing the `synchronize` block within the `unless` block, we expose ourselves to the possibility that, when the server first starts up, the servlet pool may be initialized more than once. But it's not really a problem if that does happen, and if we put the `synchronize` block there we don't have to synchronize on every single request.

You've heard it before: "Avoid premature optimization." Assumptions about the impact of the servlet pool size on memory consumption and performance often prove to be wrong, given the complexities introduced by garbage collection and the variation in the efficiency of various operations on different platforms. Code first, tune later.

Here's the application-specific code. The file `certificate_pdf.rb` should contain the `Certificate` class defined in the Discussion of Recipe 12.13.

When the servlet is initialized, we generate the PDF certificate, leaving the name blank:

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
require 'certificate_pdf'

class PDFCertificateServlet < PooledServlet

  pool_size 10

  def initialize(server, *options)
    super
    @certificate = Certificate.new(options.first)
  end
```

When the client makes a request, we load the certificate, fill in the name, and send it as the body of the HTTP response:

```
def do_GET(request, response)
  if name = request.query['name']
    filled_in = @certificate.award_to(CGI.unescape(name))

    response.body            = filled_in.render
    response.status          = 200                    # Success
    response['Content-Type'] = 'application/pdf'
    response['Size']         = response.body.size
  else
    raise HTTPStatus::Forbidden.new("missing attribute: 'name'")
 end
end
```

The rest of the code should look familiar by now:

```
  # Respond with an HTTP POST just as we do for the HTTP GET
  alias :do_POST :do_GET
end

# Mount servlets
s = HTTPServer.new(:Port => 8002)
s.mount('/', PDFCertificateServlet, 'Ruby Hacker')

# Trap signals so as to shutdown cleanly.
['TERM', 'INT'].each do |signal|
  trap(signal){ s.shutdown }
end
# Start the server and block on input.
s.start
```

Start this server, and you can visit *http://localhost:8002/?name=My+Name* to get a customized PDF certificate.

The code above illustrates many other basic features of WEBrick: access to request parameters, servlet configuration at mount time, use of a servlet pool to handle expensive operations up front, and error pages.

Besides `HTTPStatus::Forbidden`, demonstrated above, WEBrick provides exceptions for each of the HTTP 1.1 status codes. The classes are not listed in the RDoc, but you can infer them from `HTTPStatus::StatusMessage` table. The class names correspond to the names given in the WC3 reference listed below.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

See Also

- Recipe 12.13, "Generating PDF Files," for the `CertificatePDF` class used by the certificate server
- WEBrick's web site (http://webrick.org/) offers a number of examples as well as links to related libraries
- Mongrel is an up-and-coming Ruby web server that might be the next WEBrick (http://mongrel.rubyforge.org/)
- The RDoc is available online at http://www.ruby-doc.org/stdlib/libdoc/webrick/rdoc/index.html
- *Gnome's Guide to WEBrick* at http://microjet.ath.cx/webrickguide/html/html_webrick.html provides the most comprehensive coverage of WEBrick beyond the RDoc and the source itself; the *Guide* is available in both html and PDF formats
- Eric Hodel has written a couple of short articles on WEBrick servlets and working with HTTP cookies (http://segment7.net/projects/ruby/WEBrick/index.html)
- An article on the *Linux Journal* web site, "At the Forge—Getting Started with Ruby," provides a basic introduction to Ruby CGI and WEBrick servlets (http://www.linuxjournal.com/article/8356)
- For a complete list of HTTP 1.1 status codes and explanations as to what they mean, see http://www.w.org/Protocols/rfc2616/rfc2616-sec10.html

Recipe 14.20. A Real-World HTTP Client

The first three recipes in this chapter cover different ways of fetching web pages. The techniques they describe work well if you just need to fetch one specific web page, but in the interests of simplicity they omit some details you'll need to consider when writing a web spider, a web browser, or any other serious HTTP client. This recipe creates a library that deals with the details.

Mixed HTTP and HTTPS

Any general client will have to be able to make both HTTP and HTTPS requests. But the simple `Net::HTTP` methods that work in Recipe 14.1 can't be used to make HTTPS requests. Our library will use use `HTTPRequest` objects for everything. If the user requests a URL that uses the "https" scheme, we'll flip the request object's `use_ssl` switch, as seen in Recipe 14.2.

Redirects

Lots of things can go wrong with an HTTP request: the page might have moved, it might require authentication, or it might simply be gone. Most HTTP errors call for

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

higher-level handling or human intervention, but when a page has moved, a smart client can automatically follow it to its new location.

Our library will automatically follow redirects that provide "Location" fields in their responses. It'll prevent infinite redirect loops by refusing to visit a URL it's already visited. It'll prevent infinite redirect chains by limiting the number of redirects. After all the redirects are followed, it'll make the final URI available as a member of the response object.

Proxies

Users use HTTP proxies to make high-latency connections work faster, surf anonymously, and evade censorship. Each individual client program needs to be programmed to use a proxy, and it's an easy feature to overlook if you don't use a proxy yourself. Fortunately, it's easy to support proxies in Ruby: the `Proxy` class will create a custom `Net::HTTP` subclass that works through a certain proxy.

This library defines a single new method: `Net::HTTP.fetch`, an all-singing, all-dancing factory for `HTTPRequest` objects. It silently handles HTTPS URLs (assuming you have `net/https` installed) and HTTP redirects, and it transparently handles proxies. This might go into a file called `http_fetch.rb`:

```
require 'net/http'
require 'set'

class Net::HTTPResponse
  attr_accessor :final_uri
end

module Net
  begin
    require 'net/https'
    HTTPS_SUPPORTED = true
  rescue LoadError
    HTTPS_SUPPORTED = false
  end

  class HTTP
    # Makes an HTTP request and returns the HTTPResponse object.
    # Args: :proxy_host, :proxy_port, :action (:get, :post, etc.),
    #       :data (for :post action), :max_redirects.
  def HTTP.fetch(uri, args={}.freeze, &before_fetching)
    # Process the arguments with default values
    uri = URI.parse(uri) unless uri.is_a? URI
    proxy_host = args[:proxy_host]
    proxy_port = args[:proxy_port] || 80
    action = args[:action] || :get
    data = args[:data]
    max_redirects = args[:max_redirects] || 10
```

We will always work on a `Proxy` object, even if no proxy is specified. A `Proxy` with no `proxy_host` makes direct HTTP connections. This way, the code works the same way whether we're actually using an HTTP proxy or not:

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
# Use a proxy class to create the request object
proxy_class = Proxy(proxy_host, proxy_port)
request = proxy_class.new(uri.host, uri.port)
```

We will use SSL to handle URLs of the "https" scheme. Note that we do not set any certificate paths here, or do any other SSL configuration. If you want to do that, you'll need to pass an appropriate code block into `fetch` (see below for an example):

```
request.use_ssl = true if HTTPS_SUPPORTED and uri.scheme == 'https'
yield request if block_given?
```

Now we activate the request and get an `HTTPResponse` object back:

```
response = request.send(action, uri.path, data)
```

Our `HTTPResponse` object might be a document, it might be an error, or it might be a redirect. If it's a redirect, we can make things easier for the caller of this method by following the redirect. This piece of the method finds the redirected URL and sends it into a recursive `fetch` call, after making sure that we aren't stuck in an infinite loop or an endless chain of redirects:

```
      urls_seen = args[:_urls_seen] || Set.new
      if response.is_a?(Net::HTTPRedirection)         # Redirect
        if urls_seen.size < max_redirects && response['Location']
          urls_seen << uri
          new_uri = URI.parse(response['Location'])
          break if urls_seen.member? new_uri         # Infinite redirect loop

          # Request the new location just as we did the old one.
          new_args = args.dup
          puts "Redirecting to #{new_uri}" if $DEBUG
          new_args[:_urls_seen] = urls_seen
          response = HTTP.fetch(new_uri, new_args, &before_fetching)
        end
      else                          # No redirect
        response.final_uri = uri
      end
      return response
    end
  end
end
```

That's pretty dense code, but it ties a lot of functionality into a single method with a relatively simple API. Here's a simple example, in which `Net::HTTP.fetch` silently follows an HTTP redirect. Note the `final_uri` is different from the original URI.

```
response = Net::HTTP.fetch("http://google.com/")
puts "#{response.final_uri} body is #{response.body.size} bytes."
# http://www.google.com/ body is 2444 bytes.
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

With `fetch`, redirects work even through proxies. This example accesses the Google homepage through a public HTTP proxy in Singapore. When it requests "http://google.com/", it's redirected to "http://www.google.com/", as in the previous example. But when Google notices that the IP address is coming from Singapore, it sends *another* redirect:

```
response = Net::HTTP.fetch("http://google.com/",
                           :proxy_host => "164.78.252.199")
puts "#{response.final_uri} body is #{response.body.size} bytes."
# http://www.google.com.sg/ body is 2853 bytes.
```

There are HTTPS proxies as well. This code uses an HTTPS proxy in the U.S. to make a secure connection to "https://paypal.com/". It's redirected to "https://paypal.com/us/". The second request is secured in the same way as the one that caused the redirect. Note that this code will only work if you have the Ruby SSL library installed.

```
response = Net::HTTP.fetch("https://paypal.com/",
                           :proxy_host => "209.40.194.8") do |request|
  request.ca_path = "/etc/ssl/certs/"
  request.verify_mode = OpenSSL::SSL::VERIFY_PEER
end
puts "#{response.final_uri} body is #{response.body.size} bytes."
# https://paypal.com/us/ body is 16978 bytes.
```

How does this work? The code block is actually called twice: once before requesting "https://paypal.com/" and once before requesting "https://paypal.com/us/". This is what `fetch`'s code block is for: it's run on the `HTTPRequest` object before the request is actually made. If the code block were only called once, then the second request wouldn't have access to any certificates.

`Net::HTTP.fetch` will follow redirects served by the web server, but it won't follow redirects contained in the META tags of an HTML document. To follow those redirects, you'll have to parse the document as HTML.

See Also

- Recipe 14.1, "Grabbing the Contents of a Web Page"
- Recipe 14.2, "Making an HTTPS Web Request"
- Recipe 14.3, "Customizing HTTP Request Headers"
- Several web sites have lists of public HTTP and HTTPS proxies (for instance, http://www.samair.ru/proxy/ and http://tools.rosinstrument.com/proxy/); if you want to set up a proxy on your local network, Squid is a good choice (http://www.squid-cache.org/)

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01