CHAPTER 14

Ruby and the Internet

In this chapter we're going to look at how to use Ruby with the Internet and with the various services available on the Internet, from the Web to e-mail and file transfers.

The Internet has recently become an inescapable part of software development, and Ruby has a significant number of libraries available to deal with the plethora of Internet services available. In this chapter we'll focus on a few of the more popular services: the Web, e-mail (POP3 and SMTP), and FTP, along with how to process the data we retrieve.

In Chapter 15, we'll look at how to develop actual server or daemon code using Ruby along with lower-level networking features, such as *pinging*, *TCP/IP*, and *sockets*. However, this chapter focuses on accessing and using data from the Internet, rather than on the details of Ruby's networking features.

HTTP and the Web

HyperText Transfer Protocol (HTTP) is an Internet protocol that defines how Web servers and Web clients (such as Web browsers) communicate with each other. The basic principle of HTTP, and the Web in general, is that every resource (such as a Web page) available on the Web has a distinct Uniform Resource Locator (URL), and that Web clients can use HTTP "verbs" such as `GET`, `POST`, `PUT`, and `DELETE` to retrieve or otherwise manipulate those resources. For example, when a Web browser retrieves a Web page, a `GET` request is made to the correct Web server for that page, which then returns the contents of the Web page.

In Chapter 10 we looked briefly at HTTP and developed some simple Web server applications to demonstrate how Ruby applications could make their features available on the Internet. In this section we're going to look at how to retrieve data from the Web, parse it, and generate Web-compatible content.

Downloading Web Pages

One of the most basic actions you can perform on the Web is downloading a single Web page or document. First we'll look at how to use the most commonly used Ruby HTTP library, *net/http*, before moving on to a few notable alternatives.

The net/http Library

The net/http library comes standard with Ruby and is the most commonly used library to access Web sites. Here's a basic example:

```
require 'net/http'

Net::HTTP.start('www.rubyinside.com') do |http|
  req = Net::HTTP::Get.new('/test.txt')
  puts http.request(req).body
end
```

```
Hello Beginning Ruby reader!
```

This example loads the net/http library, connects to the Web server `www.rubyinside.com` (the semi-official blog associated with this book; take a look!), and performs an HTTP `GET` request for `/test.txt`. This file's contents are then returned and displayed. The equivalent URL for this request is `http://www.rubyinside.com/test.txt`, and if you load that URL in your Web browser, you'll get the same response as Ruby.

■ **Note** `http://www.rubyinside.com/test.txt` is a live document that you can use in all the HTTP request tests safely, and has been created specifically for readers of this book.

As the example demonstrates, the net/http library is a little raw in its operation. Rather than simply passing it a URL, you have to pass it the Web server to connect to and then the local filename upon that Web server. You also have to specify the `GET` HTTP request type and trigger the request using the `request` method. You can make your work easier by using the URI library that comes with Ruby, which provides a number of methods to turn a URL into the various pieces needed by net/http. Here's an example:

```
require 'net/http'

url = URI.parse('http://www.rubyinside.com/test.txt')

Net::HTTP.start(url.host, url.port) do |http|
  req = Net::HTTP::Get.new(url.path)
  puts http.request(req).body
end
```

In this example, you use the `URI` class (automatically loaded by net/http) to parse the supplied URL. An object is returned whose methods `host`, `port`, and `path` supply different parts of the URL for Net::HTTP to use. Note that in this example you provide two parameters to the main `Net::HTTP.start` method: the URL's hostname and the URL's port number. The port number is optional, but URI is clever enough to return the default HTTP port number of 80.

It's possible to produce an even simpler example:

```
require 'net/http'

url = URI.parse('http://www.rubyinside.com/test.txt')
response = Net::HTTP.get_response(url)
puts response.body
```

Instead of creating the HTTP connection and issuing the `GET` explicitly, `Net::HTTP.get_response` allows you to perform the request in one stroke. There are situations where this can prove less flexible, but if you simply want to retrieve documents from the Web, it's an ideal method to use.

Checking for Errors and Redirects

Our examples so far have assumed that you're using valid URLs and are accessing documents that actually exist. However, Net::HTTP will return different responses based on whether the request is a success or not or if the client is being redirected to a different URL, and you can check for these. In the following example, a method called `get_web_document` is created that accepts a single URL as a parameter. It parses the URL, attempts to get the required document, and then subjects the response to a `case`/`when` block:

```
require 'net/http'

def get_web_document(url)
  uri = URI.parse(url)
  response = Net::HTTP.get_response(uri)

  case response
    when Net::HTTPSuccess:
      return response.body
    when Net::HTTPRedirection:
      return get_web_document(response['Location'])
    else
      return nil
  end
end
```

```
puts get_web_document('http://www.rubyinside.com/test.txt')
puts get_web_document('http://www.rubyinside.com/non-existent')
```

```
Hello Beginning Ruby reader!
nil
```

If the response is of the `Net::HTTPSuccess` class, the content of the response will be returned; if the response is a redirection (represented by a `Net::HTTPRedirection` object being returned) then `get_web_document` will be called again with the URL specified as the target of the redirection by the remote server. If the response is neither a success nor a redirection request, an error of some sort has occurred and `nil` will be returned.

If you wish, you can check for errors in a more granular way. For example, the error 404 means "File Not Found" and is specifically used when trying to request a file that does not exist on the remote Web server. When this error occurs, Net::HTTP returns a response of class `Net::HTTPNotFound`. However, when dealing with error 403, "Forbidden," Net::HTTP returns a response of class `Net::HTTPForbidden`.

■ **Note** A list of HTTP errors and their associated Net::HTTP response classes is available at `http://www.ruby-doc.org/stdlib/libdoc/net/http/rdoc/classes/Net/HTTP.html`.

Basic Authentication

As well as basic document retrieval, net/http supports the *Basic Authentication* scheme used by many Web servers to protect their documents behind a password-protected area. This demonstration shows how the flexibility of performing the entire request with `Net::HTTP.start` can come in useful:

```
require 'net/http'

url = URI.parse('http://www.rubyinside.com/test.txt')

Net::HTTP.start(url.host, url.port) do |http|
  req = Net::HTTP::Get.new(url.path)
  req.basic_auth('username', 'password')
  puts http.request(req).body
end
```

This demonstration still works with the Ruby Inside URL, because authentication is ignored on requests for unprotected URLs, but if you were trying to access a URL protected by Basic Authentication, `basic_auth` allowsyou to specify your credentials.

Posting Form Data

In our examples so far, we have only been retrieving data from the Web. Another form of interaction is to send data *to* a Web server. The most common example of this is when you fill out a *form* on a Web page. You can perform the same action from Ruby. For example:

```
require 'net/http'

url = URI.parse('http://www.rubyinside.com/test.cgi')

response = Net::HTTP.post_form(url,{'name' => 'David', 'age' => '24'})
puts response.body
```

```
You say David is 24 years old.
```

In this example, you use `Net::HTTP.post_form` to perform a `POST` HTTP request to the specified URL with the data in the hash parameter to be used as the form data.

■ **Note** `test.cgi` is a special program that returns a string containing the values provided by the `name` and `age` form fields, resulting in the preceding output. We looked at how to create CGI scripts in Chapter 10.

As with the basic document retrieval examples, there's a more complex, low-level way to achieve the same thing by taking control of each step of the form submission process:

```
require 'net/http'

url = URI.parse('http://www.rubyinside.com/test.cgi')

Net::HTTP.start(url.host, url.port) do |http|
  req = Net::HTTP::Post.new(url.path)
  req.set_form_data({ 'name' => 'David', 'age' => '24' })
  puts http.request(req).body
end
```

This technique allows you to use the `basic_auth` method if needed, too.

Using HTTP Proxies

Proxying is when HTTP requests do not go directly between the client and the HTTP server, but through a third party en route. In some situations it might be necessary to use an HTTP proxy for your HTTP requests. This is a common scenario in schools and offices where Web access is regulated or filtered.

net/http supports proxying by creating an HTTP proxy class upon which you can then use and perform the regular HTTP methods. To create the proxy class, use `Net::HTTP::Proxy`. For example:

```
web_proxy = Net::HTTP::Proxy('your.proxy.hostname.or.ip', 8080)
```

This call to `Net::HTTP::Proxy` generates an HTTP proxy class that uses a proxy with a particular hostname on port 8080. You would use such a proxy in this fashion:

```
require 'net/http'

web_proxy = Net::HTTP::Proxy('your.proxy.hostname.or.ip', 8080)

url = URI.parse('http://www.rubyinside.com/test.txt')

web_proxy.start(url.host, url.port) do |http|
  req = Net::HTTP::Get.new(url.path)
  puts http.request(req).body
end
```

In this example, `web_proxy` replaces the reference to Net::HTTP when using the `start` method. You can use it with the simple `get_response` technique you used earlier, too:

```
require 'net/http'

web_proxy = Net::HTTP::Proxy('your.proxy.hostname.or.ip', 8080)
url = URI.parse('http://www.rubyinside.com/test.txt')

response = web_proxy.get_response(url)
puts response.body
```

These examples demonstrate that if your programs are likely to need proxy support for HTTP requests, it might be worth generating a proxy-like system even if a proxy isn't required in every case. For example:

```
require 'net/http'

http_class = ARGV.first ? Net::HTTP::Proxy(ARGV[0], ARGV[1]) : Net::HTTP
url = URI.parse('http://www.rubyinside.com/test.txt')

response = http_class.get_response(url)
puts response.body
```

If this program is run and an HTTP proxy hostname and port are supplied on the command line as arguments for the program, an HTTP proxy class will be assigned to `http_class`. If no proxy is specified, `http_class` will simply reference Net::HTTP. This allows `http_class` to be used in place of Net::HTTP when requests are made, so that both proxy and nonproxy situations work and are coded in exactly the same way.

Secure HTTP with HTTPS

HTTP is a plain text, unencrypted protocol, and this makes it unsuitable for transferring sensitive data such as credit card information. HTTPS is the solution, as it's the same as HTTP but routed over Secure Socket Layer (SSL), which makes it unreadable to any third parties.

Ruby's net/https library makes it possible to access HTTPS URLs, and you can make net/http use it semi-transparently by setting the `use_ssl` attribute on a Net::HTTP instance to `true`, like so:

```
require 'net/http'
require 'net/https'

url = URI.parse('https://example.com/')

http = Net::HTTP.new(url.host, url.port)
http.use_ssl = true if url.scheme == 'https'

request = Net::HTTP::Get.new(url.path)
puts http.request(request).body
```

Note that you use the `scheme` method of `url` to detect if the remote URL is in fact one that requires SSL to be activated.

It's trivial to mix in the form-posting code to get a secure way of sending sensitive information to the remote server:

```
require 'net/http'
require 'net/https'

url = URI.parse('https://example.com/')

http = Net::HTTP.new(url.host, url.port)
http.use_ssl = true if url.scheme == 'https'

request = Net::HTTP::Post.new(url.path)
request.set_form_data({ 'credit_card_number' => '1234123412341234' })
puts http.request(request).body
```

net/https also supports associating your own client certificate and certification directory with your requests, as well as retrieving the server's peer certificate. However, these are advanced features only required in a small number of cases, and are beyond the scope of this section. Refer to Appendix C for links to further information.

The open-uri Library

open-uri is a library that wraps up the functionality of net/http, net/https, and net/ftp into a single package. Although it lacks some of the raw power of using the constituent libraries directly, open-uri makes it a lot easier to perform all the main functions.

A key part of open-uri is the way it abstracts common Internet actions and allows file I/O techniques to be used upon them. Retrieving a document from the Web becomes much like opening a text file on the local machine:

```
require 'open-uri'

f = open('http://www.rubyinside.com/test.txt')
puts f.readlines.join
```

```
Hello Beginning Ruby reader!
```

As with `File::open`, `open` returns an I/O object (technically, a `StringIO` object), and you can use methods such as `each_line`, `readlines`, and `read`, as you did in Chapter 9.

```
require 'open-uri'

f = open('http://www.rubyinside.com/test.txt')

puts "The document is #{f.size} bytes in length"

f.each_line do |line|
  puts line
end
```

```
The document is 29 bytes in length
Hello Beginning Ruby reader!
```

Also, in a similar fashion to the `File` class, you can use `open` in a block style:

```
require 'open-uri'

open('http://www.rubyinside.com/test.txt') do |f|
  puts f.readlines.join
end
```

■ **Note** HTTPS and FTP URLs are treated transparently. You can use any HTTP, HTTPS, or FTP URL with `open`.

As well as providing the `open` method as a base method that can be used anywhere, you can also use it directly upon `URI` objects:

```
require 'open-uri'

url = URI.parse('http://www.rubyinside.com/test.txt')
url.open { |f| puts f.read }
```

Or you could use this code if you were striving for the shortest open-uri code possible:

```
require 'open-uri'
puts URI.parse('http://www.rubyinside.com/test.txt').open.read
```

Note Ruby developers commonly use quick hacks, such as in the prior example, but to catch errors successfully, it's recommended to surround such one-liners with the `begin`/`ensure`/`end` structure to catch any exceptions.

In addition to acting like an I/O object, open-uri enables you to use methods associated with the object it returns to find out particulars about the HTTP (or FTP) response itself. For example:

```
require 'open-uri'

f = open('http://www.rubyinside.com/test.txt')

puts f.content_type
puts f.charset
puts f.last_modified
```

```
text/plain
iso-8859-1
Sun Oct 15 02:24:13 +0100 2006
```

Last, it's possible to send extra header fields with an HTTP request by supplying an optional hash parameter to `open`:

```
require 'open-uri'

f = open('http://www.rubyinside.com/test.txt',
         {'User-Agent' => 'Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0)'})

puts f.read
```

In this example, a "user agent" header is sent with the HTTP request that makes it appear as if you're using Internet Explorer to request the remote file. Sending a user agent header can be a useful technique if you're dealing with a Web site that returns different information to different types of browsers. Ideally, however, you should use a `User-Agent` header that reflects the name of your program.

Generating Web Pages and HTML

Web pages are created using a variety of technologies; the most popular is HyperText Markup Language (HTML). HTML is a language that can be represented in plain text and is composed of numerous *tags* that indicate the meaning of each part of the document. For example:

```
<html>
  <head>
    <title>This is the title</title>
  </head>
  <body>
    <p>This is a paragraph</p>
  </body>
</html>
```

A tag begins like so:

```
<tag>
```

And it ends like so:

```
</tag>
```

Anything between a start tag and an end tag belongs, semantically, to that tag. So, the text in between the `<p>` and `</p>` tags is part of a single paragraph (where `<p>` is the HTML tag for a paragraph). This explains why the entire document is surrounded by the `<html>` and `</html>` tags, as the entire document is HTML.

Web applications, and other applications that need to output data to be shown on the Web, usually need to produce HTML to render their output. Ruby provides a number of libraries to make this easier than simply producing HTML in a raw fashion using strings. In this section we'll look at two such libraries, *Markaby* and *RedCloth*.

Markaby—Markup As Ruby

Markaby is a library developed by Tim Fletcher (`http://tfletcher.com/`) and "why the lucky stiff" (`http://whytheluckystiff.net/`) that allows you to create HTML by using Ruby methods and structures. Markaby is distributed as a gem, so it's trivial to install using the gem client, like so:

```
gem install markaby
```

Once Markaby is installed, the following example should demonstrate the basic principles of generating HTML with it:

```
require 'rubygems'
require 'markaby'

m = Markaby::Builder.new

m.html do
  head { title 'This is the title' }

  body do
    h1 'Hello world'
    h2 'Sub-heading'
    p %q{This is a pile of stuff showing off Markaby's features}
    h2 'Another sub-heading'
    p 'Markaby is good at:'
    ul do
      li 'Generating HTML from Ruby'
      li 'Keeping HTML structured'
      li 'Lots more..'
    end
  end
end

puts m
```

```
<html><head><meta content="text/html; charset=utf-8" http-equiv="Content-
Type"/><title>This is the title</title></head><body><h1>Hello world</h1><h2>Sub-
heading</h2><p>This is a pile of stuff showing off Markaby's features</p><h2>
Another sub-heading</h2><p>Markaby is good at:</p><ul><li>Generating HTML from
Ruby</li><li>Keeping HTML structured</li><li>Lots more..</li></ul></body></html>
```

The output is basic HTML that could be viewed in any Web browser. Markaby works by interpreting method calls as HTML tags, so that you can create HTML tags merely by using method calls. If a method call is passed a code block (as with `m.html` and `body` in the previous example), then the code within the code block will form the HTML that goes within the start and stop tags of the parent.

Because Markaby offers a pure Ruby way to generate HTML, it's possible to integrate Ruby logic and flow control into the HTML generation process:

```
require 'rubygems'
require 'markaby'

m = Markaby::Builder.new

items = ['Bread', 'Butter', 'Tea', 'Coffee']

m.html do
  body do
    h1 'My Shopping List'
    ol do
      items.each do |item|
        li item
      end
    end
  end
end

puts m
```

```
<html><body><h1>My Shopping
List</h1><ol><li>Bread</li><li>Butter</li><li>Tea</li>
<li>Coffee</li></ol></body></html>
```

If you viewed this output in a typical Web browser, it'd look somewhat like this:

```
My Shopping List

1. Bread
2. Butter
3. Tea
4. Coffee
```

A common feature of HTML that you might want to replicate is to give elements class or ID names. To give an element a class name, you can use a method call attached to the

element method call. To give an element an ID, you can use a method call attached to the element method call that ends with an exclamation mark (`!`). For example:

```
div.posts! do
  div.entry do
    p.date Time.now.to_s
    p.content "Test entry 1"
  end

  div.entry do
    p.date Time.now.to_s
    p.content "Test entry 2"
  end
end
```

```
<div id="posts"><div class="entry"><p class="date">Mon Oct 16 02:48:06 +0100
2006</p><p class="content">Test entry 1</p></div><div class="entry"><p
class="date">Mon Oct 16 02:48:06 +0100 2006</p><p class="content">Test entry
2</p></div></div>
```

In this case, a parent `div` element is created with the id of `"posts"`. Child `div`s with the class name of `"entry"` are created, containing paragraphs with the class names of `"date"` and `"content"`.

It's important to note that the HTML generated by Markaby is not necessarily strictly valid HTML. Using tags and structure correctly is your responsibility as a developer.

■ **Note** To learn more about Markaby, refer to the official Markaby homepage and documentation at `http://markaby.rubyforge.org/`.

RedCloth

RedCloth is a library that provides a Ruby implementation of the *Textile* markup language. The Textile markup language is a special way of formatting plain text to be converted into HTML.

Here's a demonstration of Textile, followed by the HTML that a Textile interpreter would generate from it:

```
h1. This is a heading.

This is the first paragraph.

This is the second paragraph.

h1. Another heading

h2. A second level heading

Another paragraph
```

```
<h1>This is a heading.</h1>
<p>This is the first paragraph.</p>
<p>This is the second paragraph.</p>
<h1>Another heading</h1>
<h2>A second level heading</h2>
<p>Another paragraph</p>
```

Textile provides a more human-friendly language that can be converted easily to HTML. RedCloth makes this functionality available in Ruby.

RedCloth is available as a RubyGem and can be installed in the usual way (such as with `gem install redcloth`), or see Chapter 7 for more information. To use RedCloth, create an instance of the `RedCloth` class and pass in the Textile code you want to use:

```
require 'rubygems'
require 'redcloth'

text = %q{h1. This is a heading.

This is the first paragraph.

This is the second paragraph.

h1. Another heading

h2. A second level heading
```

```
Another paragraph}

document = RedCloth.new(text)
puts document.to_html
```

The `RedCloth` class is a basic extension of the `String` class, so you can use regular string methods with `RedCloth` objects, or you can use the `to_html` method to convert the RedCloth/Textile document to HTML.

The Textile language is a powerful markup language, but its syntax is beyond the scope of this chapter. It supports easy ways to convert plain text to complex HTML containing tables, HTML entities, images, and structural elements. To learn more about RedCloth and Textile, refer to the official RedCloth Web site at `http://redcloth.rubyforge.org/`.

■ **Note** BlueCloth is another markup library for Ruby that also exists as a gem. You can learn more about its operation in Chapter 16 or at the official BlueCloth Web site at `http://www.deveiate.org/projects/BlueCloth`.

Processing Web Content

As you saw earlier, retrieving data from the Web is a snap with Ruby. Once you've retrieved the data, it's likely you'll want to do something with it. Parsing data from the Web using regular expressions and the usual Ruby string methods is an option, but several libraries exist that make it easier to deal with different forms of Web content specifically. In this section we'll look at some of the best ways to process HTML and XML (including feed formats such as RSS and Atom).

Parsing HTML with Hpricot

In previous sections we used Markaby and RedCloth to generate HTML from Ruby code and data. In this section, we'll look at doing the reverse by taking HTML code and extracting data from it in a structured fashion.

Hpricot is a Ruby library by "why the lucky stiff" designed to make HTML parsing fast, easy, and fun. It's available as a RubyGem via `gem install hpricot`. Though it relies on a compiled extension written in C for its speed, a special Windows build is available via RubyGems with a precompiled extension.

Once installed, Hpricot is easy to use. The following example loads the Hpricot library, places some basic HTML in a string, creates a `Hpricot` object, and then searches

for H1 tags (using `search`). It then retrieves the first (using `first`, as `search` returns an array), and looks at the HTML within it (using `inner_html`):

```
require 'rubygems'
require 'hpricot'

html = <<END_OF_HTML
<html>
<head>
  <title>This is the page title</title>
</head>

<body>
  <h1>Big heading!</h1>
  <p>A paragraph of text.</p>
  <ul><li>Item 1 in a list</li><li>Item 2</li><li class="highlighted">Item
3</li></ul>
</body>
</html>
END_OF_HTML

doc = Hpricot(html)
puts doc.search("h1").first.inner_html
```

```
Big heading!
```

Hpricot can work directly with open-uri to load HTML from remote files, as in the following example:

```
require 'rubygems'
require 'hpricot'
require 'open-uri'

doc = Hpricot(open('http://www.rubyinside.com/test.html'))
puts doc.search("h1").first.inner_html
```

■ **Note** `http://www.rubyinside.com/test.html` contains the same HTML code as used in the prior example.

Using a combination of search methods, you can search for the list within the HTML (defined by the <ul> tags, where the <li> tags denote each item in the list) and then extract each item from the list:

```
list = doc.search("ul").first
list.search("li").each do |item|
  puts item.inner_html
end
```

```
Item 1 in a list
Item 2
Item 3
```

As well as searching for elements and returning an array, Hpricot can also search for the first instance of an element only, using `at`:

```
list = doc.at("ul")
```

However, Hpricot can search for more than element or tag names. It also supports XPath and CSS expressions. These querying styles are beyond the scope of this chapter, but here's a demonstration of using CSS classes to find certain elements:

```
list = doc.at("ul")
highlighted_item = list.at("/.highlighted")
puts highlighted_item.inner_html
```

```
Item 3
```

This example finds the first list in the HTML file, then looks for a child element that has a class name of `highlighted`. The rule `.highlighted` looks for a class name of `highlighted`, whereas a rule of `#highlighted` would search for an element with the ID of `highlighted`.

■ **Note** You should prefix CSS expressions with a forward slash (`/`).

You can learn more about Hpricot and the syntaxes and styles it supports at the official site at `http://code.whytheluckystiff.net/hpricot/`. Hpricot is a work in progress, and its feature set is likely to have grown since the publishing of this book.

Parsing XML with REXML

Extensible Markup Language (XML) is a simple, flexible, plain-text data format that can represent many different structures of data. An XML document, at its simplest, looks a little like HTML:

```
<people>
  <person>
    <name>Peter Cooper</name>
    <gender>Male</gender>
  </person>
  <person>
    <name>Fred Bloggs</name>
    <gender>Male</gender>
  </person>
</people>
```

This extremely simplistic XML document defines a set of people containing two individual persons, each of whom has a name and gender. In previous chapters we've used YAML in a similar way to how XML is used here, but although YAML is simpler and easier to use with Ruby, XML is more popular outside the Ruby world.

XML is prevalent when it comes to sharing data on the Internet in a form that's easy for machines to parse, and is especially popular when using APIs and machine-accessible services provided online, such as Yahoo!'s search APIs and other programming interfaces to online services. Due to XML's popularity, it's worthwhile to see how to parse it with Ruby.

Ruby's primary XML library is called *REXML* and comes with Ruby by default as part of the standard library.

REXML supports two different ways of processing XML files: tree parsing and stream parsing. Tree parsing is where a file is turned into a single data structure that can then be searched, traversed, and otherwise manipulated. Stream parsing is when a file is processed and parsed on the fly by calling special *callback* functions whenever something in the file is found. Stream parsing is less powerful in most cases than tree parsing, although it's slightly faster. In this section we'll focus on tree parsing, as it makes more sense for most situations.

Here's a basic demonstration of parsing an XML file looking for certain elements:

```
require 'rexml/document'

xml = <<END_XML
<people>
  <person>
    <name>Peter Cooper</name>
```

```
    <gender>Male</gender>
  </person>
  <person>
    <name>Fred Bloggs</name>
    <gender>Male</gender>
  </person>
</people>
END_XML

tree = REXML::Document.new(xml)

tree.elements.each("people/person") do |person|
  puts person.get_elements("name").first
end
```

```
<name>Peter Cooper</name>
<name>Fred Bloggs</name>
```

You built the tree of XML elements by creating a new `REXML::Document` object. Using the `elements` method of `tree` returns an array of every element in the XML file. `each` accepts an XPath query (a form of XML search query), and passes matching elements into the associated code block. In this example you look for each `<person>` element within the `<people>` element.

Once you have each `<person>` element in `person`, you use `get_elements` to retrieve any `<name>` elements into an array, and then pull out the first one. Because there's only one name per person in your XML data, the correct name is extracted for each person in the data.

REXML has support for most of the basic XPath specification, so if you become familiar with XPath, you can search for anything within any XML file that you need.

■ **Note** You can learn more about XPath and its syntax at `http://en.wikipedia.org/wiki/XPath`. REXML also has support for XQuery, which you can learn more about at `http://en.wikipedia.org/wiki/XQuery`.

Further resources relating to processing XML and using REXML and XPath within Ruby are provided in Appendix C.

Parsing Web Feeds with FeedTools

Web feeds (sometimes known as news feeds, and more commonly as just "feeds") are special XML files designed to contain multiple items of content (such as news). They're commonly used by blogs and news sites as a way for users to subscribe to them. A feed reader reads RSS and Atom feeds (the two most popular feed formats) from the sites the user is subscribed to, and whenever a new item appears within a feed, the user is notified by his or her feed client, which monitors the feed regularly. Most feeds allow users to read a synopsis of the item and to click a link to visit the site that has updated.

■ **Note** Another common use for feeds has been in delivering *podcasts*, a popular method of distributing audio content online in a radio subscription–type format.

Processing RSS and Atom feeds has become a popular task in languages such as Ruby. As feeds are formatted in a machine-friendly format, they're easier for programs to process and use than scanning through inconsistent HTML.

FeedTools (`http://sporkmonger.com/projects/feedtools/`) is a Ruby library for handling RSS and Atom feeds. It's available as a RubyGem with `gem install feedtools`. It's a liberal feed parser, which means it tends to excuse as many faults and formatting problems in the feeds it reads as possible. This makes it an ideal choice for processing feeds, rather than creating your own parser manually with REXML or another XML library.

For the examples in this section you'll use the RSS feed provided by RubyInside.com, a popular Ruby weblog. Let's look at how to process a feed rapidly by retrieving it from the Web and printing out various details about it and its constituent items:

```
require 'rubygems'
require 'feed_tools'

feed = FeedTools::Feed.open('http://www.rubyinside.com/feed/')

puts "This feed's title is #{feed.title}"
puts "This feed's Web site is at #{feed.link}"

feed.items.each do |item|
  puts item.title + "\n---\n" + item.description + "\n\n"
end
```

Parsing feeds is even easier than downloading Web pages, because FeedTools handles all the technical aspects for you. It handles the download of the feed and even caches the contents of the feed for a short time so you're not making a large number of requests to the feed provider.

In the preceding example you opened a feed, printed out the title of the feed, printed out the URL of the site associated with the feed, and then used the array of items in `feed.items` to print the title and description of each item in the feed.

As well as `description` and `title`, feed items (objects of class `FeedTools::FeedItem`) also offer methods such as `author`, `categories`, `comments`, `copyright`, `enclosures`, `id`, `images`, `itunes_author`, `itunes_duration`, `itunes_image_link`, `itunes_summary`, `link`, `published`, `rights`, `source`, `summary`, `tags`, `time`, and `updated`.

A full rundown of feed terminology is beyond the scope of this book, but if you want to learn more, refer to the Web feed section on Wikipedia at `http://en.wikipedia.org/wiki/Web_feed`. You can find feeds for most major Web sites nowadays, so processing news with your Ruby scripts can be an easy reality.

E-Mail

E-mail predates the invention of the Internet, and is still one of the most important and popular technologies used online. In this section you'll look at how to retrieve and manage e-mail located on POP3 servers, as well as how to send e-mail using an SMTP server.

Receiving Mail with POP3

Post Office Protocol 3 (POP3) is the most popular protocol used to retrieve e-mail from a mail server. If you're using an e-mail program that's installed on your computer (as opposed to webmail, such as HotMail or Yahoo! Mail) it probably uses the POP3 protocol to communicate with the mail server that receives your mail from the outside world.

With Ruby it's possible to use the *net/pop* library to do the same things that your e-mail client can, such as preview, retrieve, or delete mail. If you were feeling creative, you could even use net/pop to develop your own anti-spam tools.

■ **Note** In this section our examples won't run without adjustments, as they need to operate on a real mail account. If you wish to run them, you would need to replace the server name, username, and passwords with those of a POP3/mail account that you have access to. Ideally, you'll be able to create a test e-mail account if you want to play with the examples here, or have a backup of your mail first, in case of unforeseen errors. That's because although you cannot delete mail directly from your local e-mail program, you might delete any new mail waiting on your mail server. Once you're confident of your code and what you want to achieve, then you change your settings to work upon a live account.

The basic operations you can perform with a POP3 server are to connect to it, receive information about the mail an account contains, view that mail, delete the mail, and

disconnect. First, you'll connect to a POP3 server to see if there are any messages available for download, and if so, how many:

```
require 'net/pop'

mail_server = Net::POP3.new('mail.mailservernamehere.com')

begin
  mail_server.start('username','password')
  if mail_server.mails.empty?
    puts "No mails"
  else
    puts "#{mail_server.mails.length} mails waiting"
  end
rescue
  puts "Mail error"
end
```

This code first creates an object referring to the server and then uses the `start` method to connect. The entire section of the program that connects to and works with the mail server is wrapped within a `begin/ensure/end` block so that connection errors are picked up without the program crashing out with an obscure error.

Once `start` has connected to the POP3 server, `mail_server.mails` contains an array of `Net::POPMail` objects that refer to each message waiting on the server. You use `Array`'s `empty?` method to see if any mail is available, and if so, the size of the array is used to tell how many mails are waiting.

You can use the `Net::POPMail` objects' methods to manipulate and collect the server-based mails. Downloading all the mails is as simple as using the `pop` method for each `Net::POPMail` object:

```
mail_server.mails.each do |m|
  mail = m.pop
  puts mail
end
```

As each mail is retrieved (or popped, if you will) from the server, the entire contents of the mail, with headers and body text, are placed into the `mail` variable, before being displayed onscreen.

To delete a mail, you can use the `delete` method, although mails are only *marked* for deletion later, once the session has ended:

```
mail_server.mails.each do |m|
  m.delete if m.pop =~ /\bthis is a spam e-mail\b/i
end
```

This code goes through every message in the account and marks it for deletion if it contains the string `this is a spam e-mail`.

You can also retrieve *just* the headers. This is useful if you're looking for a mail with a particular subject or a mail from a particular e-mail address. Whereas `pop` returns the entire mail (which could be up to many megabytes in size), `header` only returns the mail's header from the server. The following example deletes messages if their subject contains the word "medicines":

```
mail_server.mails.each do |m|
  m.delete if m.header =~ /Subject:.+?medicines\b/i
end
```

To build a rudimentary anti-spam filter, you could use a combination of the mail retrieval and deletion techniques to connect to your mail account and delete unwanted mails before your usual mail client ever sees them. Consider what you could achieve by downloading mail, passing it through several regular expressions, and then choosing to delete depending on what you match.

Sending Mail with SMTP

Where POP3 handles the client-side operations of retrieving, deleting, and previewing e-mail, Simple Mail Transfer Protocol (SMTP) handles sending e-mail and routing e-mail between mail servers. In this section you won't be looking at this latter use, but will use SMTP simply to send mails to an e-mail address.

The net/smtp library allows you to communicate with SMTP servers directly. On many Unix machines, especially servers on the Internet, you can send mail to the SMTP server running on the local machine and it will be delivered across the Internet. In these situations, sending e-mail is as easy as this:

```
require 'net/smtp'

message = <<MESSAGE_END
From: Private Person <me@privacy.net>
To: Author of Beginning Ruby <test@rubyinside.com>
Subject: SMTP e-mail test

This is a test e-mail message.
MESSAGE_END

Net::SMTP.start('localhost') do |smtp|
  smtp.send_message message, 'me@privacy.net', 'test@rubyinside.com'
end
```

You place a basic e-mail in `message`, using a *here document*, taking care to format the headers correctly (e-mails require a From, To, and Subject header, separated from the body of the e-mail with a blank line, as in the preceding code). To send the mail you use Net::SMTP to connect to the SMTP server on the local machine and then use the `send_message` method along with the message, the from address, and the destination address as parameters (even though the from and to addresses are within the e-mail itself, these aren't always used to route mail).

If you're not running an SMTP server on your machine, you can use Net::SMTP to communicate with a remote SMTP server. Unless you're using a webmail service (such as Hotmail or Yahoo! Mail), your e-mail provider will have provided you with outgoing mail server details that you can supply to Net::SMTP, as follows:

```
Net::SMTP.start('mail.your-domain.com')
```

This line of code connects to the SMTP server on port 25 of `mail.your-domain.com` without using any username or password. If you need to, though, you can specify port number and other details. For example:

```
Net::SMTP.start('mail.your-domain.com', 25, 'localhost', 'username', 'password',➥
:plain)
```

This example connects to the SMTP server at `mail.your-domain.com` using a username and password in plain text format. It identifies the client's hostname as `localhost`.

■ **Note** Net::SMTP also supports LOGIN and CRAM-MD5 authentication schemes. To use these, use `:login` or `:cram_md5` as the sixth parameter passed into `start`.

Sending Mail with ActionMailer

ActionMailer (`http://wiki.rubyonrails.org/rails/pages/ActionMailer`) makes sending e-mail more high-level than using the SMTP protocol (or net/smtp) *directly*. Instead of talking directly with an SMTP server, you create a descendent of `ActionMailer::Base`, implement a method that sets your mail's subject, recipients, and other details, and then you call that method to send e-mail.

ActionMailer is a part of the Ruby on Rails framework (as covered in Chapter 13), but can be used independently of it. If you don't have Ruby on Rails installed on your computer yet, you can install the ActionMailer gem with `gem install actionmailer`.

Here's a basic example of using ActionMailer:

```
require 'rubygems'
require 'action_mailer'

class Emailer < ActionMailer::Base
  def test_email(email_address, email_body)
    recipients(email_address)
    from "me@privacy.net"
    subject "This is a test e-mail"
    body email_body
  end
end

Emailer.deliver_test_email('me@privacy.net', 'This is a test e-mail!')
```

A class, `Emailer`, is defined and descends from `ActionMailer::Base`. The `test_email` method uses ActionMailer's helper methods to set the recipient, from address, subject, and body of the e-mail, but you never call this method directly. To send the mail, you call a dynamic class method on the `Emailer` class called `deliver_test_email` (or `deliver_` followed by whatever you called the method in the class).

In the preceding example, ActionMailer uses the default settings for mail output, and that is to try to connect to an SMTP server on the local machine. If you don't have one installed and running, you can instruct ActionMailer to look for an SMTP server elsewhere, like so:

```
ActionMailer::Base.server_settings = {
  :address => "mail.your-domain.com",
  :port => 25,
  :authentication => :login,
  :user_name => "username",
  :password => "password",
}
```

These settings are similar to those you used to set up Net::SMTP and can be changed to match your configuration.

File Transfers with FTP

File Transfer Protocol (FTP) is a basic networking protocol for transferring files on any TCP/IP network. Although files can be sent back and forth on the Web, FTP is still commonly used for large files, or for access to large file repositories that have no particular

relevance to the Web. One of the benefits of FTP is that authentication and access control is built in.

The core part of the FTP system is an FTP server, a program that runs on a file server that allows FTP clients to download and/or upload files to that machine.

In a previous section of this chapter, "The open-uri Library," we looked at using the open-uri library to retrieve files easily from the Internet.The open-uri supports HTTP, HTTPS, and FTP URLs, and is an ideal library to use if you want to download files from FTP servers with as little code as possible. Here's an example:

```
require 'open-uri'

output = File.new('1.8.2-patch1.gz', 'w')
open('ftp://ftp.ruby-lang.org/pub/ruby/1.8/1.8.2-patch1.gz') do |f|
  output.print f.read
end
output.close
```

This example downloads a file from an FTP server and saves its contents into a local file.

■ **Note** The example might fail for you, as your network connection might not support active FTP and might require a passive FTP connection. This is covered later in this section.

However, for more complex operations, the net/ftp library is ideal, as it gives you lower-level access to FTP connections, as net/http does to HTTP requests.

Connection and Basic FTP Actions

Connecting to an FTP server with net/ftp using an FTP URL is a simple operation:

```
require 'net/ftp'
require 'uri'

uri = URI.parse('ftp://ftp.ruby-lang.org/')

Net::FTP.open(uri.host) do |ftp|
  ftp.login 'anonymous', 'me@privacy.net'
  ftp.passive = true
  ftp.list(uri.path) { |path| puts path }
end
```

```
drwxrwxr-x    2 0        103             6 Sep 10  2005 basecamp
drwxrwxr-x    3 0        103            41 Oct 13 04:53 pub
```

You use `URI.parse` to parse a basic FTP URL, and connect to the FTP server with `Net::FTP.open`. Once the connection is open, you have to specify login credentials (much like the authentication credentials when using Net::HTTP) with the `ftp` object's `login` method. Then you set the connection type to be passive (this is an FTP option that makes an FTP connection more likely to succeed when made from behind a firewall—the technical details are beyond the scope of this book), and then ask the FTP server to return a list of the files in the directory referenced in your URL (the root directory of the FTP server in this case).

Net::FTP provides a `login` method that you can use against a Net::FTP object, like so:

```
require 'net/ftp'

ftp = Net::FTP.new('ftp.ruby-lang.org')
ftp.passive = true
ftp.login
ftp.list('*') { |file| puts file }
ftp.close
```

■ **Note** If you know you're going to be connecting to an anonymous FTP server (one that is public and requires only generic credentials to log in), you don't need to specify any credentials with the login method. This is what happens in the preceding example.

This example demonstrates a totally different way of using Net::FTP to connect to an FTP server. As with Net::HTTP and `File` classes, it's possible to use Net::FTP within a structural block or by manually opening and closing the connection by using the reference object (`ftp` in this case).

As no username and password are supplied, the `login` method performs an anonymous login to `ftp.ruby-lang.org`. Note that in this example you connect to an FTP server by its hostname rather than with a URL. However, if a username and password are required, use this code:

```
ftp.login(username, password)
```

Once connected, you use the `list` method on the `ftp` object to get a list of all files in the current directory. Because you haven't specified a directory to change to, the current

directory is the one that the FTP server puts you in by default. However, to change directories, you can use the `chdir` method:

```
ftp.chdir('pub')
```

It's also possible to change to any directory in the remote filesystem:

```
ftp.chdir('/pub/ruby')
```

If you have permission to do so (this depends on your account with the FTP server) you might also be able to create directories. This is done with `mkdir`:

```
ftp.mkdir('test')
```

Performing this operation on an FTP server where you don't have the correct permissions causes an exception, so it's worth wrapping such volatile actions within blocks to trap any exceptions that arise.

Likewise, you can delete and rename files:

```
ftp.rename(filename, new_name)
ftp.delete(filename)
```

These operations will only work if you have the correct permissions.

Downloading Files

Downloading files from an FTP server is easy if you know the filename and what type of file you're trying to download. Net::FTP provides two useful methods to download files, `getbinaryfile` and `gettextfile`. Plain text files and binary files (such as images, sounds, or applications) are sent in a different way, so it's essential you use the correct method. In most situations you'll be aware ahead of time which technique is required. Here's an example showing how to download a binary file from the official Ruby FTP server:

```
require 'net/ftp'

ftp = Net::FTP.new('ftp.ruby-lang.org')
ftp.passive = true
ftp.login
ftp.chdir('/pub/ruby/1.8')
ftp.getbinaryfile('1.8.2-patch1.gz')
ftp.close
```

`getbinaryfile` accepts several parameters, only one of which is mandatory. The first parameter is the name of the remote file (`1.8.2-patch1.gz` in this case), an optional

second parameter is the name of the local file to write to, and the third optional parameter is a block size that specifies in what size chunks (in bytes) the file is downloaded. If you omit the second parameter, the downloaded file will be written to the same filename in the local directory, but if you want to write the remote file to a particular local location, you can specify this.

One problem with using `getbinaryfile` in this way is that it locks up your program until the download is complete. However, if you supply `getbinaryfile` with a code block, the downloaded data will be supplied into the code block as well as saved to the file:

```
ftp.getbinaryfile('stable-snapshot.tar.gz', 'local-filename', 102400) do |blk|
  puts "A 100KB block of the file has been downloaded"
end
```

This code prints a string to the screen whenever another 100 kilobytes of the file have been downloaded. You can use this technique to provide updates to the user, rather than make him or her wonder whether the file is being downloaded or not.

You could also download the file in blocks such as this and process them on the fly in the code block, like so:

```
ftp.getbinaryfile('stable-snapshot.tar.gz', 'local-filename', 102400) do |blk|
  .. do something with blk here ..
end
```

Each 100KB chunk of the file that's downloaded is passed into the code block. Unfortunately, the file is still saved to a local file, but if this isn't desired, you could use Tempfile (as covered in Chapter 9) to use a temporary file that's then immediately deleted.

Downloading text or ASCII-based files uses the same technique as in the preceding code, but demands using `gettextfile` instead. The only difference is that `gettextfile` doesn't accept the third block size parameter, and instead returns data to the code block line by line.

Uploading Files

Uploading files to an FTP server is only possible if you have write permissions on the server in the directory to which you want to upload. Therefore, none of the examples in this section will work unedited, as you can't provide an FTP server with write access (for obvious reasons!).

Uploading is the exact opposite of downloading, and net/ftp provides `putbinaryfile` and `puttextfile` methods that accept the same parameters as `getbinaryfile` and `gettextfile`. The first parameter is the name of the local file you want to upload, the optional second parameter is the name to give the file on the remote server (defaults to the

same as the uploaded file's name if omitted), and the optional third parameter for `putbinaryfile` is the block size to use for the upload. Here's an upload example:

```
require 'net/ftp'

ftp = Net::FTP.new('ftp.domain.com')
ftp.passive = true
ftp.login
ftp.chdir('/your/folder/name/here')
ftp.putbinaryfile('local_file')
ftp.close
```

As with `getbinaryfile` and `gettextfile`, if you supply a code block, the uploaded chunks of the file are passed into it, allowing you to keep the user informed of the progress of the upload.

```
require 'net/ftp'

ftp = Net::FTP.new('ftp.domain.com')
ftp.passive = true
ftp.login
ftp.chdir('/your/folder/name/here')

count = 0

ftp.putbinaryfile('local_file', 'local_file', 100000) do |block|
  count += 100000
  puts "#{count} bytes uploaded"
end

ftp.close
```

If you need to upload data that's just been generated by your Ruby script and isn't within a file, you need to create a temporary file with Tempfile and upload from that. For example:

```
require 'net/ftp'
require 'tempfile'

tempfile = Tempfile.new('test')

my_data = "This is some text data I want to upload via FTP."
tempfile.puts my_data
```

```
ftp = Net::FTP.new('ftp.domain.com')
ftp.passive = true
ftp.login
ftp.chdir('/your/folder/name/here')

ftp.puttextfile(tempfile.path, 'my_data')

ftp.close
tempfile.close
```

Summary

In this chapter we've looked at Ruby's support for using various Internet systems and protocols, how Ruby can work with the Web, and how to process and manipulate data retrieved from the Internet.

Let's reflect on the main concepts covered in this chapter:

- *HTTP*: HyperText Transfer Protocol. A protocol that defines the way Web browsers (clients) and Web servers talk to each other across a network such as the Internet.
- *HTTPS*: A secure version of HTTP that ensures data being transferred in either direction is only readable at each end. Anyone intercepting an HTTPS stream cannot decipher it. It's commonly used for e-commerce and for transmitting financial data on the Web.
- *HTML*: HyperText Markup Language. A text formatting and layout language used to represent Web pages.
- *WEBrick*: An HTTP server toolkit that comes as standard with Ruby. WEBrick makes it quick and easy to put together basic Web servers.
- *Mongrel*: Another HTTP server library, developed by Zed Shaw, that's available as a gem and is faster and more scalable in operation than WEBrick.
- *Markaby*: A Ruby library that makes it possible to produce HTML directly from Ruby methods and logic.
- *RedCloth*: A Ruby implementation of the Textile markup language that makes it easy to produce HTML documents from specially formatted plain text.
* *Hpricot*: A self-proclaimed "fast and delightful" HTML parser developed to make it easy to process and parse HTML with Ruby. It is noted for its speed, with intensive sections written in C.

* *POP3*: Post Office Protocol 3. A mail server protocol commonly used when retrieving e-mail.

* *SMTP*: Simple Mail Transfer Protocol. A mail server protocol commonly used to transfer mail to a mail server or between mail servers. SMTP is used for sending mail, rather than receiving it.

* *FTP*: File Transfer Protocol. An Internet protocol for providing access to files located on a server and allowing users to download and upload to it.

In this chapter we've covered a variety of Internet-related functions, but in Chapter 15 we're going to look more deeply at networking, servers, and network services. Most of what is covered in Chapter 15 is also applicable to the Internet, but is at a much lower level than FTP or using the Web.