

5 *Web services*

This chapter covers

- Using HTTP from Ruby
- Building REST services with Rails
- Using REST services with ActiveResource
- Using SOAP with SOAP4R

Throughout this book, we show you ways of integrating Ruby with different applications and services. Some scenarios depend on a particular protocol or architecture, such as using Lightweight Directory Access Protocol (LDAP), sharing data through a relational database, or moving messages around with WebSphere MQ (WMQ). In this chapter, we'll explore the web architecture and look at how we can use web services across language, platform, and application boundaries.

To this end, we'll focus on Service Oriented Architecture (SOA) with Ruby. SOA is not a particular product, technology, or protocol—it's an architecture for building services with the intent to reuse them in different contexts and combine them into larger applications. In this chapter, we're going to choose common protocols and message formats and discuss three common styles for building web services: plain HTTP, REST, and SOAP.

First, we'll cover the foundation and show you how to use HTTP and URLs. Then we'll venture into the world of RESTful web services and show you how to handle resources, representations, and the uniform interface. Finally, we'll talk about

SOAP and using SOAP4R as a means for integrating with services developed around the SOAP stack, and in particular J2EE and .Net applications.

5.1 Using HTTP

The basic building blocks of the web are the HTTP protocol and the use of URLs for addressing, and the content is mostly HTML and various media types like images, music, and video. In this chapter, we're going to focus on the programmatic web, which is all about machines talking to machines and application-to-application integration. As a result, we'll pay much more attention to structured data formats like XML, JSON, and even CSV.

5.1.1 HTTP GET

We're going to start with the simplest scenario and show you how to retrieve data from a remote web service and then how to parse the resulting document into structured data that you can process with Ruby. For this example we picked CSV. Although most people equate web services with XML, there's a surprising abundance of structured data out there that is not XML. In this section, we'll take the opportunity to show you how easy it is to use non-XML data, delivered over the web.

Problem

You're building a market intelligence application that needs to retrieve historical stock prices about various public companies. That information is publicly available on the web, but you need to retrieve it and parse it into data you can use.

Solution

For this example, we're going to use Google Finance to get historical stock prices for Google itself. Google Finance has a URL you can use without registering or authenticating. It provides historical data in the form of a CSV document.

Ruby provides two libraries for working with HTTP. For full HTTP support, we'll turn to `Net::HTTP`, which we cover in the next section, but for the common use case of reading data from a web service, we'll use the more convenient `open-uri`.

So, to start, we're going to construct a URL and use the `open` method to create a connection and read the data:

```
url = "http://finance.google.com/finance/historical?q=NASDAQ:#{symbol}&
  ↳ output=csv"
data = open(url).read
```

Not all data on the web is HTML or XML, and in this example we retrieve a CSV document. We're going to use `FasterCSV` to parse the document (you will learn more about `FasterCSV` in chapter 13). Let's parse the document into a set of rows, and convert each row into a hash:

```
csv = FasterCSV.parse data, :headers=>true, :converters=>:numeric
csv.map { |r| r.to_hash }
```

Now we're going to roll all of this into a single method called `historical_stock_prices`, so we can use it to read historical stock prices in our applications. We're also going to follow another Ruby idiom that allows us to either require the

file as a library or run it from the command line as a script. Listing 5.1 shows the entire program.

Listing 5.1 Get historical stock prices from Google Finance

```
require 'rubygems'
require 'open-uri'
require 'faster-csv'

def historical_stock_prices(ticker)
  url = "http://finance.google.com/finance/historical
  ➔ ?q=NASDAQ:#{ticker}&output=csv"
  data = open(url).read
  csv = FasterCSV.parse data, :headers=>true,
    :converters=>:numeric
  csv.map { |r| r.to_hash }
end

if __FILE__ == $0
  pp historical_stock_prices(ARGV.first)
end
```

Here's what we get when we run the program from the command line, showing one line from the actual result:

```
$ ruby historical.rb GOOG
=> {"High"=>522.07, "Open"=>521.28, "Close"=>514.48, "Date"=>"10-Sep-07",
"Volume"=>3225800, "Low"=>510.88}
```

Discussion

Using the `open-uri` library is the easiest way to GET content from an HTTP server. It's designed to be simple to use and to handle the most common cases.

Our example was simple enough that we could treat the URL as a string. In more complex use cases, you will want to use URI objects instead. URI is a library that can parse a URI into its components, like host, path, and query string. You can also use it to construct URIs from their component parts and manipulate them. It includes classes for supporting HTTP and FTP URLs, email (`mailto:`), and LDAP URIs, and is easily extended.

The `open-uri` library adds a `read` method to HTTP and FTP URIs, so let's see how we would use it:

```
uri = URI("http://finance.google.com/finance/historical?output=csv")
uri.query << "&q=NASDAQ:#{CGI.escape(ticker)}"
data = uri.read
```

Note that we're using `CGI.escape` to properly escape a value before including it in the query string. It's not necessary for our example, since all the stock-ticker symbols we want to deal with are ASCII characters, but it's generally a good idea to deal with special characters like `=` and `&` appearing in query string parameters.

The `read` method returns the content of the HTTP response as a string. Most times that's all we care for, and we like the convenience of it. Occasionally, however,

we'll want more information from HTTP headers; for example, to determine the content type of the response, or to handle redirects.

In Ruby, each object implements a single class, but you can also add methods to an object instance, in addition to those provided by its class. There are several ways of doing this.

First, `open-uri` uses the `extend` method to add metadata methods to the `String` object it returns. We can use that to access various HTTP headers, like this:

```
puts "The actual URL after redirection: #{data.base_uri}"
puts "Content type: #{data.content_type}"
puts "Last modified: #{data.last_modified}"
puts "The document: #{data}"
```

If you access the web from behind a proxy server, you can tell `open-uri` to use that proxy server using the `:proxy` option. You can set the `HTTP_PROXY` environment variable and `open-uri` will pick it up by default. So let's use that to run our example from behind a proxy server:

```
$ export HTTP_PROXY="http://myproxy:8080"
$ ruby historical.rb GOOG
```

In the next example, we'll use `open-uri` to access a local server, bypassing the proxy, and using HTTP Basic Authentication:

```
open(url, :proxy=>nil, :http_basic_authentication=>["john", "secret"])
```

For downloading larger documents, `open-uri` will also allow you to use a progress bar. Check the `open-uri` documentation for more details.

Now that we have covered retrieving data from a web server, let's see how we can send data to a web server by making an HTTP POST request.

5.1.2 HTTP POST

The previous section was a simple scenario using `GET` to access publicly available information. In this section, we'll turn it up a notch and use `HTTP POST` to submit data, add authentication for access control, and handle status codes and errors.

Problem

In your network, you have an existing service that can receive and process orders. You need a way to turn orders into XML documents and send them over to the order-processing service.

Solution

Let's start with the data. To make this solution easy to use, we're going to support two types of arguments. The XML document can be passed directly, in the form of a string, or the data can be passed as a `Hash`, with the method converting it into an XML document.

For this simple example, we're going to use the `XmlSimple` library, so let's install it first:

```
gem install xml-simple
```

We will use `XmlSimple` to convert a hash into an XML document:

```
if Hash === data
  data = XmlSimple.xml_out(data, 'noattr'=>true, 'contentkey'=>'sku',
    'xmldeclaration'=>true, 'rootname'=>'order')
end
```

The XML document we're going to create will look like this:

```
<?xml version='1.0' standalone='yes'?>
<order>
  <item>
    <quantity>1</quantity>
    <sku>123</sku>
  </item>
  <item>
    <quantity>2</quantity>
    <sku>456</sku>
  </item>
</order>
```

Now that we have the data, it's time to create an HTTP connection. We'll start by parsing the URL string into a URI object, and set up `Net::HTTP` to use either the HTTP or HTTPS protocol:

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

Next, we're going to set the HTTP headers. We don't want the server to accept partial documents, which could happen if the connection drops, so we're going to tell it exactly how long the document is. And for extra measure, we're going to use an MD5 hash to make sure the document is not corrupted:

```
headers = { 'Content-Type'=>'application/xml',
  'Content-Length'=>data.size.to_s,
  'Content-MD5'=>Digest::MD5.hexdigest(data) }
```

In this example, we make a single request, so we'll let `Net::HTTP` deal with opening and closing the connection:

```
post = Net::HTTP::Post.new(uri.path, headers)
post.basic_auth uri.user, uri.password if uri.user
response = http.request post, data
```

We send the request, and we don't expect any data in the result, but we do want to know if our request was successful, so the last thing we'll do is look at the status code returned by the server. A successful response is anything with a 2xx status code. Some services return 200 (OK), but others may return 201 (Created), 202 (Accepted), or 204 (No Content). In this case, we expect 201 (Created) with the location of the new resource, but we'll also respond favorably to any other 2xx status code. All other responses are treated as error conditions:

```
case response
  when Net::HTTPCreated; response['Location']
```

```

when Net::HTTPSuccess; nil
else response.error!
end

```

Listing 5.2 shows all these pieces merged into a single file.

Listing 5.2 Using HTTP POST and XmlSimple to send a document to the web server

```

require 'rubygems'
require 'uri'
require 'net/https'
require 'xmlsimple'
require 'md5'

def send_order(url, data)
  if Hash === data
    data = XmlSimple.xml_out(data, 'noattr'=>true,
                             'contentkey'=>'sku',
                             'xmldeclaration'=>true, 'rootname'=>'order')
  end

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

  headers = { 'Content-Type'=>'application/xml',
             'Content-Length'=>data.size.to_s,
             'Content-MD5'=>Digest::MD5.hexdigest(data) }

  post = Net::HTTP::Post.new(uri.path, headers)
  post.basic_auth uri.user, uri.password if uri.user
  response = http.request post, data

  case response
  when Net::HTTPCreated; response['Location']
  when Net::HTTPSuccess; nil
  else response.error!
  end
end

```

Converts Hash to XML document

Creates new Net::HTTP object

Sets headers for content type and size

Creates new POST request

Uses HTTP Basic Authentication

Checks response

Now let's see how we can send a new order with three lines of code (four if you count the print statement):

```

order = { 'item'=>[ { 'sku'=>'123', 'quantity'=>1 },
                  { 'sku'=>'456', 'quantity'=>2 } ] }
url = send_order('https://order.server/create', order)
puts "Our new order at: #{url}" if url

```

Discussion

Ruby has several libraries for dealing with XML, and which one you choose depends on the use case. When you need maximum flexibility and the ability to handle elements, attributes, and text nodes, you can use REXML to work with the XML document tree. For performance-intensive applications, you can use libxml, a native XML library. If you have data that you need to convert into an XML document, you can use Builder instead (we'll look at using it later in this chapter). Some XML documents map nicely into a hash, in which case XmlSimple is true to its name. For this particular example,

we chose XmlSimple because it fits nicely with what we wanted to do, but throughout this book we will use different XML libraries, always choosing the one that's best for the task at hand.

Some developers prefer to write or use APIs where authentication information is passed as a separate argument on each method call, or stored once in an object. We prefer to pass authentication information in the URL itself. You'll notice that the previous example takes the username and password from the URL and applies it to the POST request, using HTTP Basic Authentication. Since HTTP Basic Authentication passes the username and password as clear text, we'll use that in combination with HTTPS to encrypt the request from snooping eyes. A URL with authentication information in it will look like `https://john:secret@example.com/create`.

HTTP is a connectionless protocol; it handles each request individually. The server doesn't care whether we send each request in a separate connection or send multiple requests in a single connection. HTTP 1.1 provides keep-alive connections that we can use to open a connection once and send multiple requests—the benefit is that we don't have to create a TCP connection for each request, and it gives us better latency. We can do this:

```
http.start do |conn|
  response = conn.request post, data
end
```

Our example makes a single request. We could have opened the connection, made that one request, and closed it. In fact, we did just that when we called the `request` method on the `Net::HTTP` object; it was just masked behind a single method call. If your code is making several requests to the same server, consider explicitly opening a connection and using the connection object to make those requests.

As you've seen, the `Net::HTTP` library covers all the features of the HTTP protocol, from one-line requests all the way to persistent connections, from GET requests to documents and forms, with support for HTTPS and HTTP Basic Authentication. It supports all the common HTTP verbs like GET and POST, the less common ones like HEAD and OPTIONS, and even the WebDAV headers. It supports all the HTTP headers, and response codes help you distinguish between successful response codes, redirects, and errors.

Keep in mind, though, that `Net::HTTP` is a low-level library. Unlike `open-uri`, it will not set the proxy server for you; you'll need to do that yourself when creating connections. Most often, we use `Net::HTTP` for one-off tasks, or to write more convenient libraries. In section 5.2.3, we'll talk about one such library called `ActiveResource` that provides RESTful access to remote resources, built from `Net::HTTP`.

Before we do that, let's look at the other side of HTTP—the server side. We showed you how a client can retrieve data from and send data to a web server. Next we'll show you how to implement a simple web server that client applications can retrieve data from.

5.1.3 Serving HTTP requests

Now that we know how to access a web service, let's build a simple service. In the previous chapter, we talked about Rails, and as you'll see later in this chapter, Rails is a great framework for building web services. Yet, not all web services need a full-fledged framework, and sometimes working close to the protocol (HTTP) is better, so in this section we'll show you a simple service built straight into the HTTP server.

Problem

You have an analytics console that you could use to monitor traffic across all servers in the network. It works by pulling log files from each server and batch processing them. To make it work with your server, you need to set up a simple service that provides all the log files for a given day.

Solution

Ruby comes with a built-in web server called WEBrick, a simple and lightweight server that is an adequate choice for development and prototyping. For production, though, we recommend using Mongrel or Thin. Both are faster and more stable than WEBrick, and you can use them as standalone servers, or to deploy a cluster of load-balanced application servers behind Apache or Nginx.

In this example, we'll use Mongrel as a standalone server, and we'll start by installing it:

```
gem install mongrel
```

Next, we'll define the `LogService`, which needs to implement a single method called `process`, which handles the HTTP request and sets the response:

```
class LogService < Mongrel::HttpHandler
  def process(request, response)
    ...
  end
end
```

A request that ends with `/YYYY-MM-DD` retrieves all the log files for that particular day. A request that ends with `/last` retrieves the last set of log files, which happens to be yesterday's date:

```
case request.params['PATH_INFO']
  when /^\/(\d{4}-\d{2}-\d{2})$/
    package $1, response
  when '/last'
    package (Date.today - 1).to_s, response
  else
    response.start 404 do |head, out|
      head['Content-Type'] = 'text/html'
      script = request.params['SCRIPT_NAME']
      out.write "<h1>Request URL should be #{script}/last \"
        \" or#{script}/[yyyy] - [mm] - [dd]</h1>"
    end
  end
end
```


If we get a request we cannot process, we return the 404 (Not Found) status code, but we also include a short HTML snippet explaining the correct URL format. We do that to help our users, since getting the request URL wrong is a common mistake.

The package method will handle all valid requests, so let's look at it next. We're going to use the RubyZip library:

```
gem install rubyzip
```

Strictly speaking, we want to create an empty zip file, add all the log files into it, use default compression, and return that file to the client. We're going to decide on the zip filename first, and we'll make sure to use a distinct filename for each day:

```
zip_filename = "logs-#{date}.zip"
```

We're not going to create a file with that name. Imagine two requests coming at the same time, attempting to write into the same zip file at once. Not good. So instead, we'll create a temporary file:

```
tmp_file = Tempfile.open(zip_filename)
```

Next, we'll use RubyZip to add the log files:

```
Zip::ZipOutputStream.open(tmp_file.path) do |zip|
  Dir.glob("#{@path}/*-#{date}.log").each do |filename|
    zip.put_next_entry File.basename(filename)
    zip << File.read(filename)
  end
end
```

The `glob` method is named after the glob pattern, which we can use to match any number of files. An asterisk (*) matches any filename or partial filename, a question mark (?) matches any single character, and a double asterisk (**) matches directories recursively. You can find a few more options in the `Dir.glob` documentation. Here we'll find all filenames that contain the date and end with the extension *log*, such as `errors-2007-10-05.log`.

Once we have created the zip file, we'll return it to the client:

```
response.start 200 do |head, out|
  head['Content-Type'] = 'application/zip'
  head['Content-Length'] = File.size(tmp_file.path)
  head['Content-Disposition'] = %{attachment; filename="#{zip_filename}"}
  while buffer = tmp_file.read(4096)
    out.write buffer
  end
end
```

It's a simple matter of returning the status code 200 (OK) and copying the file over to the Mongrel response, which we do one block at a time to keep the service from eating up all available memory.

We're just about done. We have a Mongrel `HttpHandler` that responds to GET requests by returning a zip file with all the log files for a given date, and we can use that as a building block for a larger application that includes several services by simply

registering the handler on a given URL. We're not going to show you a larger application here. Instead we'll make it possible to run this service from the command line:

```
service = LogService.new(path)
puts "Starting Mongrel on port #{port}, serving log files from '#{path}'"
mongrel = Mongrel::HttpServer.new('0.0.0.0', port)
mongrel.register '/logs', service
mongrel.run.join
```

We set up the server to listen on IP 0.0.0.0, which means any network card including localhost. You can also specify a specific IP address or host name, or only allow requests from the same machine by listening to localhost (127.0.0.1).

Let's run the server:

```
ruby log_service.rb ~/logs
Starting Mongrel on port 3000, serving log files from '/home/assaf/logs'
```

To retrieve all the latest log files, simply open your browser and head over to `http://localhost:3000/logs/last`.

Now let's merge all that code into a single file, shown in Listing 5.3.

Listing 5.3 A service for packaging log files and serving them as a zip file

```
require 'rubygems'
require 'mongrel'
require 'zip/zip'

class LogService < Mongrel::HttpHandler

  def initialize(path)
    @path = path
  end

  def process(request, response)
    unless request.params['REQUEST_METHOD'] == 'GET'
      return response.status = 405
    end

    case request.params['PATH_INFO']
    when /^\/(\d{4}-\d{2}-\d{2})$/
      package $1, response
    when '/last'
      package (Date.today - 1).to_s, response
    else
      response.start 404 do |head, out|
        head['Content-Type'] = 'text/html'
        script = request.params['SCRIPT_NAME']
        out.write "<h1>Request URL should be #{script}/last\
          " or #{script}/[yyyy]-[mm]-[dd]</h1>"
      end
    end
  end

  private

  def package(date, response)
    zip_filename = "logs-#{date}.zip"
```

Accepts only
GET requests

Determines
resource from
request path

Returns 404 if
resource not found

```

tmp_file = Tempfile.open(zip_filename)  ← Creates a temporary file
begin
  Zip::ZipOutputStream.open(tmp_file.path) do |zip|
    Dir.glob("#{@path}/*-#{date}.log").each do |file|
      zip.put_next_entry File.basename(file)
      zip << File.read(file)
    end
  end
  response.start 200 do |head, out|
    head['Content-Type'] = 'application/zip'
    head['Content-Length'] = File.size(tmp_file.path)
    head['Content-Disposition'] = %{attachment;
  → filename="#{zip_filename}"}
    while buffer = tmp_file.read(4096)
      out.write buffer
    end
  end
  ensure
    tmp_file.close!  ← Discards temporary file
  end
end

end

if __FILE__ == $0
  unless path = ARGV[0]
    puts "Usage:"
    puts " ruby log_service.rb <log_dir> [<port>]"
    exit
  end
  port = ARGV[1] || 3000
  service = LogService.new(path)
  puts "Starting Mongrel on port #{port}, serving log files from '#{path}'"
  mongrel = Mongrel::HttpServer.new('0.0.0.0', port)
  mongrel.register '/logs', service
  mongrel.run.join
end

```

Uses RubyZip to compress files

Returns successful (200) response

Streams file to the client

Handles command-line arguments

Starts a new Mongrel server

Discussion

This example shows you how to set up a simple web service without going the route of a web framework. We do advocate using web frameworks when they help you get better results with less work, and in the next section we'll delve into RESTful services using Rails. Sometimes, though, a web framework just gets in the way, and we wanted to make you feel comfortable using the simplest solution for each situation.

Another thing web frameworks do is hide, or abstract away, the HTTP protocol. In our experience, you'll do better if you learn how to use HTTP with all its richness, whether you're writing the code yourself, learning how to use a web framework, or evaluating a library for use in your application. Our service only supports GET requests, so we used the status code 405 (Method Not Allowed) to deny all other HTTP methods. That status code tells the client exactly why his request was rejected.

Along with the response, we sent three headers. The first, `Content-Type`, tells the client the file type, and a web browser can use this information to open the file with the

right application. The second, `Content-Length`, tells the client how long the response is, which is particularly useful for large responses, and for showing a progress bar of the download. The HTTP protocol allows the server to close the connection once it's done sending the request, but if the connection drops (and sometimes it does), the client doesn't know whether or not it received the full response. The `Content-Length` header gets around that problem. We also used the `Content-Disposition` header to suggest a filename. Without this header, a request to `/logs/last` would attempt to download and save a file called "last." With this header, the browser will offer to save the file under a name like `logs-2007-10-05.zip`.

The last thing we did was send the request progressively, in blocks of 4,096 bytes. That allows the client to start reading in, and if necessary, to start saving the response, without waiting for the server to be done reading the file. It also saves the server from loading the entire file, which could well be gigabytes of data, into memory. Paying attention to these details will improve the performance and scalability of your applications, and the responsiveness of your web servers.

Now that we have covered the basics of HTTP, we're going to go one step further and explore the REST style. We'll show you how to create resources, handle multiple representations, and use the uniform interface to build RESTful web services.

5.2 REST with Rails

So far, we've shown you how to build services and clients that use the HTTP protocol. We'll take this a step further now and show you how to build RESTful services using Rails.

The Representational State Transfer (REST) architectural style is modeled after the web. Basically, it codifies the principles and methods behind web servers that lead to the creation of the largest distributed system ever built. For some people, "distributed" is about the plumbing—sending messages to remote servers. However, we're also thinking of the way large-scale systems emerge from smaller services built independently by different groups of people—systems that are distributed in design and in implementation.

When we follow the REST style, we follow those same web principles: modeling our services in terms of resources, making sure they are addressable as URLs, connecting them by linking from one resource to another, handling representations based on content type, performing stateless operations, and so forth. In the following sections, we'll show an example using Rails. You'll also quickly realize why we picked Rails for this task.

5.2.1 RESTful resources

Besides being the largest collection of useless information, personal opinions, and short video clips, the web is also a large-scale system built from resources. Each page is a resource identified by its URL. We use links to navigate from one resource to another, and forms to operate on those resources. Applying these same principles, we can build web services that are simple for both people and applications to use, and we can wire them together to create larger applications.

Problem

You're designing a task manager that your employees will use to manage their day-to-day assignments. You're also planning several applications and workflows that will create and act upon these tasks. How can you design your task manager as a web service that both people and applications can use?

Solution

Obviously one part of the solution is supporting programmable web formats like XML and JSON, which we'll handle in the next section. Before we get to deal with that, we need to understand how to structure our resources so we can consume them from web browsers and client applications.

When we develop a web service, our aim is to build the service once and support any number of clients that want to connect to it. The more client applications that can reuse our service, the more we get out of the initial effort that goes into building that service. We're always on the lookout for those principles and practices that would make our service loosely coupled and ripe for reuse. In this section, we're going to do just that by applying REST principles to a task manager.

We'll start by identifying the most important resources we need to provide. We have one resource representing the collection of tasks, which we'll make apparent by using the URL path `/tasks`. And since we also plan to operate on individual tasks, we'll give each task its individual resources, and we'll do so hierarchically by placing each task in a resource of the form `/tasks/{id}`.

We'll handle all of these through the `TasksController`, so the first thing we'll do is define the resource so Rails can map incoming requests to the right controller. We do that in the `config/routes.rb` file:

```
ActionController::Routing::Routes.draw do |map|
  # Tasks resources handled by TasksController
  map.resources :tasks
end
```

Retrieving the list of all tasks in the collection is done by the `index` action:

```
class TasksController < ApplicationController
  # GET on /tasks
  # View: tasks/index.html.erb
  def index
    @tasks = Task.for_user(@user_id)
  end
  ...
end
```

For individual tasks, we're going to use the `show` action when the client asks to retrieve that one task:

```
# GET on /tasks/{id}
# View: tasks/show.html.erb
def show
  @task = Task.find(params[:id])
end
```

What else would we want to do with a task? We'll want to change (update) it, and we'll need to offer a way to delete it. We can do all three on the same resource. We can use HTTP GET to retrieve the task, PUT to update the task, and DELETE to discard it. So let's add two more actions that operate on a member task:

```
# PUT on /tasks/{id}
def update
  @task = Task.find(params[:id])
  @task.update_attributes! params[:task]
  respond_to do |format|
    format.html { redirect_to:action=>'edit', :id=>task.id }
    format.xml { render :xml=>task }
  end
end

# DELETE on /tasks/{id}
def destroy
  Task.find(params[:id]).destroy

  head :no_content
end
```

We got a bit ahead of ourselves. Before we can do all these things with a task, we need some way to create it. Since we have a resource representing the collection of tasks, and each task is represented by its own resource, we're going to use HTTP POST to create a new task in the collection:

```
# POST on /tasks
def create
  task = Task.create!(params[:task])
  respond_to do |format|
    format.html { redirect_to:action=>'show', :id=>task.id }
    format.xml { render :xml=>@task, :status=>:created,
                      :location=>url_for(:action=>'show', :id=>task.id) }
  end
end
```

We can now start to write applications that create, read, update, and delete tasks. The beauty is that we've done it entirely using one resource to represent the collection and one resource to represent each member, and we've used the HTTP methods POST (create), GET (read), PUT (update), and DELETE (delete). When it comes time to develop another service, say for managing users or orders, we can follow the same conventions, and we can take what we learned from one service and apply it to all other services.

We're not done, though. We want to expose this service to both people and applications. Our employees are going to use a web browser; they're not going to send a POST or PUT request, but do that using forms. So we need two forms: one for creating a task, and one for updating an existing task. We can place those inside the task list and individual task view respectively. For larger forms—and our tasks will require several fields, taking up most of the page—we want to offer separate pages linked from existing view pages, so we're going to offer two additional resources.

From the tasks list, we're going to link to a separate resource representing a form for creating new tasks, and following our hierarchical design, we'll assign it the URL path `/tasks/new`. Likewise, we'll associate each individual task with a URL for viewing and editing it:

```
# GET on /tasks/new
# View: tasks/new.html.erb
def new
  @task = Task.new
end

# GET on /tasks/{id}/edit
# View: tasks/edit.html.erb
def edit
  @task = Task.find(params[:id])
end
```

Now it's becoming clearer why we choose to lay out the resources hierarchically. If you like tinkering with the browser's address bar, try this: open the edit form for a given task, say `/tasks/123/edit`, and change the URL to go up one level to the task view at `/tasks/123`, and up another level to the tasks list at `/tasks`. Besides being a nice browser trick, this setup helps developers understand how all the resources relate to each other. This is one case where picking intuitive URLs is worth a thousand words of documentation.

So let's pause and review what we have so far:

- GET request to `/tasks` returns the list of all tasks.
- POST request to `/tasks` creates a new task and redirects back to the tasks list.
- GET request to `/tasks/new` returns a form that we can use to create a new task; it will POST to `/tasks`.
- GET request on `/tasks/{id}` returns a single task.
- PUT request on `/tasks/{id}` updates that task.
- DELETE request on `/tasks/{id}` deletes that task.
- GET request on `/tasks/{id}/edit` returns a form that we can use to update an existing task; it will PUT these changes to `/tasks/{id}`.

We didn't get here by accident. We intentionally chose these resources so that we need to keep track of only one reference (URL) to the tasks list and one reference to each individual task. Helping us was the fact that we can use all four HTTP methods, which already define the semantics of operations we can do against these resources. Notice that while adding more actions to our controllers, we made no change to our routing configuration. These conventions are a matter of practical sense, and Rails follows them as well, so our one-line definition of the resource captures all that logic, and all we had to do was fill in the actions.

Next, we're going to add a couple of actions that are specific to our task manager and extend our resource definition to cover those.

The first resource we're going to add is for viewing the collection of completed tasks. We can follow the same rules to add resources for viewing pending tasks, tasks

scheduled to complete today, high-priority tasks, and so forth. We're going to place it at the URL path `/tasks/completed`.

The second resource we're going to add will make it easier to change task priority. Right now, making a change to the task requires updating the task resource. We want to develop a simple AJAX control that shows five colored numbers and sets the task priority when the user clicks on one of those numbers. We'll make it easy by providing a resource to represent the task priority, so we can write an `onClick` event handler that updates the resource priority directly. We'll associate the priority resource with the URL path `/tasks/{id}/priority`.

Let's add these two resources together and create the routes shown in listing 5.4.

Listing 5.4 Defining our task manager resources in `config/routes.rb`

```
ActionController::Routing::Routes.draw do |map|
  # Tasks resources handled by TasksController
  map.resources :tasks,
    :collection => { :completed=>:get },
    :member => { :priority=>:put }
end
```

Next, let's add the controller actions to `TaskController`:

```
# GET on /tasks/completed
# View: tasks/completed.html.erb
def completed
  @tasks = Task.completed_for_user(@user_id)
end

# PUT on /tasks/{id}/priority
def priority
  @task = Task.find(params[:id])
  @task.update_attributes! :priority=>request.body.to_i
  head :ok
end
```

Will it work? We certainly hope so, but we won't know until we check. Rails resource definitions are easy to work with, but we still occasionally make mistakes and create something different from what we intended. So let's investigate our route definitions using the `routes` task:

```
$ rake routes
```

The output should look something like listing 5.5.

Listing 5.5 Routes for our RESTful tasks list

```
completed_tasks GET    /tasks/completed    {:action=>"completed"}
  tasks GET          /tasks              {:action=>"index"}
  POST             /tasks              {:action=>"create"}
  new_task GET       /tasks/new          {:action=>"new"}
completion_task PUT    /tasks/:id/completion {:action=>"completion"}
  edit_task GET      /tasks/:id/edit     {:action=>"edit"}
  task GET          /tasks/:id          {:action=>"show"}
```



```

PUT    /tasks/:id    {:action=>"update"}
DELETE /tasks/:id    {:action=>"destroy"}

```

The actual output is more verbose; we trimmed it to fit the page by removing the controller name (no surprise, it's always “tasks”) and the formatting routes, which we'll cover in the next section. You can see how each HTTP method (in the second column) and URL template (third column) map to the correct controller action (rightmost column). A quick peek tells us all we need to know.

The leftmost column deserves a bit more explanation. Rails creates several friendly-looking routing methods that we can use instead of the catch-all `url_for`. For example, since our tasks list needs a link to the URL for the task-creation form, we can write this:

```

<%= link_to "Create new task",
  url_for(:controller=>'tasks', :action=>'new') %>

```

Or, using the named-route method, we can shorten it to this:

```

<%= link_to "Create new task", new_task_url %>

```

Likewise, we could have the task list link to each task's individual page:

```

<%= link_to task.title, task_url(task) %>

```

Or we can include a link for the task-editing form:

```

<%= link_to "Edit this task", edit_task_url(task) %>

```

We're done, so let's have a look at what our controller looks like with all the actions brought together in one file. As we write it up, we're going to make a couple of minor tweaks. First, we'll use named routes instead of `url_for`. Second, we'll add a filter to load the task into the controller, for the benefit of actions operating on individual tasks. Listing 5.6 shows the resulting controller.

Listing 5.6 Routes for our RESTful tasks list

```

class TasksController < ApplicationController
  before_filter :set_task, :only=>[:show, :edit, :update,
                                   :destroy, :priority]
  def index
    @tasks = Task.for_user(@user_id)
  end
  def completed
    @tasks = Task.completed_for_user(@user_id)
  end
  def new
    @task = Task.new
  end
  def create
    task = Task.create!(params[:task])
    respond_to do |format|
      format.html { redirect_to task_url(task) }
    end
  end
end

```

← Filters for actions on individual tasks

← Redirects browser to task view

```

    format.xml { render :xml=>task, :status=>:created,
                     :location=>task_url(task) }
  end
end

def show
end

def edit
end

def update
  @task.update_attributes! params[:task]
  respond_to do |format|
    format.html { redirect_to edit_task_url(@task) }
    format.xml { render :xml=>@task }
  end
end

def priority
  @task.update_attributes! :priority=>request.body.to_i
  head :ok
end

def destroy
  @task.destroy
end
head :no_content
end

private

def set_task
  @task = Task.find(params[:id])
end
end

```

**Returns XML document
for new task**

**Associates
controller with
task from URL**



Discussion

We showed you how to build a simple RESTful web service using Rails. However, there are a few more things worth noting about this example and how we used Rails to apply the principles of REST.

One of the core principles of REST is the uniform interface. HTTP provides several methods you can use on each resource; the four we're showing here are POST (create), GET (read), PUT (update), and DELETE (delete). They have clear semantics, and everyone understands them the same way. Clients know what GET does and how it differs from DELETE, servers operate differently on POST and PUT, caches know they can cache the response to a GET but must invalidate it on DELETE, and so forth. You can also use that consistency to build more reliable applications; for example, PUT and DELETE are idempotent methods, so if you fail while making a request, you can simply repeat it. The uniform interface saves us from having to reinvent and document these semantics for each and every application, and it helps that we can always do the same thing the same way.

Unfortunately, while we get this variety for the programmable web, web browsers have not yet caught up, and some cannot properly handle PUT and DELETE. A common

workaround is to use `POST` to simulate `PUT` and `DELETE` by sending the real HTTP method in the `_method` parameter. Rails understands this convention, and so do many AJAX libraries like `Prototype.js` and `jQuery`, so you can safely use these with Rails to keep your resources RESTful.

You will notice, in our example, that when updating an existing resource (the task), we respond to the `PUT` request with the default status code 200 (OK) and an XML representation of the updated resource. On the other hand, when creating a resource, we respond to the `POST` request with the status code 201 (Created), an XML representation of the new resource, and the `Location` header. The latter tells the client application that we just created a resource and where to find that resource, to retrieve and update it later on. In both responses, we return a document that may be different from the one we received, perhaps with added fields like `id`, `version`, and `updated_at`. Either way, we're using the full semantics of the HTTP protocol to distinguish between creating a resource and updating an existing one.

People work differently from applications, however, and when responding to a web browser, we need to consider the user experience. The way browsers work, if we simply responded to a `POST` request with a render, and the user then refreshed the page, the browser would make another `POST` request—the double-submit problem. We don't want that to happen, so we redirect instead. We also don't need to send back a representation of the resource, or its location; instead, we take the user back to the tasks lists.

You may be wondering, what happens if someone makes a request to `/tasks/456`, but there is no such task? Clearly this should return a 404 (Not Found) response, yet we show no such thing in our example. One way in which Rails simplifies deployment is by taking care of all these details and applying default behavior, so you don't have to worry about it unless you want to change the way it behaves. So we let Rails figure it out.

When we call `Task.find` and it can't find a task with that identifier, it throws an `ActiveRecord::RecordNotFound` exception. Rails catches this exception and maps it to the 404 (Not Found) status code. The default behavior is to send back a static page that you can find (and customize to your application) in `public/404.html`.

Likewise, if we tried to create or update a task by sending a field it doesn't understand, such as an XML document with the element `<address>` (our tasks don't have an address field), Rails will throw an `ActiveRecord::RecordInvalid` or `ActiveRecord::RecordNotSaved` exception. It will then catch this exception and map it to a 422 (Unprocessable Entity) status code.

Rails similarly deals with unsupported content types by returning 406 (Not Acceptable), which we'll put into action in the next section. You can add your own logic for catching and dealing with these exceptions, and you can introduce your own exception and handling logic. Have a look at `ActionController::Rescue`, particularly the `rescue_from` method.

One common mistake web developers make is storing a copy of an object in the session, like this:

```
Task.find_by_user(session[:user])
```

What's wrong with this code? Updating the user's record in the database, or even deleting it, will not update the session, and the session will keep using stale data. It is much better to store the record identifier, which doesn't change, and access the record as necessary. The common alternative looks like this:

```
Task.find_by_user_id(session[:user_id])
```

This code works better, as long as you're using sessions. When developing applications that use a web service, it's much easier to work with HTTP Basic Authentication, as we've shown in the previous sections. It's easier to use than going through a custom login form and then carrying the session cookie around.

Fortunately, it's a trivial matter to write controllers that support both means of authentication. Simply add a filter that can use HTTP Basic Authentication or the session to identify the user, and store their identifier in the `@user_id` instance variable. We recommend doing that in `ApplicationController`, which is why we're not showing this filter in our example.

We talked about the ease of mapping resources for CRUD (create, read, update, delete) operations. Resource mapping is another area where we encourage you to explore more. You can take hierarchical resources one step further and create nested resources, such as `/books/598/chapters/5`. You can use the `to_param` method to create more friendly URLs, such as `/books/598-ruby-in-practice/chapters/5-web-services`. Also, have a look at some of the form helper methods that will generate the right form from an `ActiveRecord` object, using the most suitable resource URL. This combination will not only make it easier to develop web applications, but also help you do the right thing from the start.

When building RESTful web services, another thing we have to deal with are multiple content types. We briefly touched upon this, using HTML for end users and XML for applications, and in the next section we'll explore it further, adding support for JSON and Atom.

5.2.2 **Serving XML, JSON, and Atom**

Every resource has a representation. In fact, a given resource can have more than one representation. Users accessing our task manager will want to see an HTML page listing all their tasks, or they may choose to use a feed reader to subscribe to their task list, and feed readers expect an Atom or RSS document. If we're writing an application, we would want to see the tasks list as an XML document or JSON object, or perhaps to pull it into a calendar application in the form of an iCal list of to-dos and events.

In this section, we're going to explore resources by looking at multiple representations, starting with HTML and adding XML, JSON, and Atom representations for our tasks list.

Problem

As you're building your task manager, you realize you need to support a number of clients, specifically feed readers and programmable clients, by adding XML, JSON, and Atom representations to the tasks list.

Solution

One reason we recommend Rails for building web services is the ease of adding different representations for the same underlying resource. So let's start with a simple action that displays the current task list in one of several formats:

```
def index
  @tasks = Task.for_user(@user_id)
end
```

Since most Rails examples look like this and only support HTML, we won't fault you for thinking this example shows just an HTML output, but in fact it supports as many formats as we have views. When you leave it up to Rails to render the response, it tries to find a suitable view based on the action name and expected format. If we wrote a view called `index.html.erb`, Rails would use it to render HTML responses. If we added a view called `index.xml.builder`, Rails would use this one to render XML responses. For Atom, we would use `index.atom.builder`, and for iCal, `index.ics.erb`.

Notice the pattern here? The first part tells Rails which action this view represents, the second part tells it which format it applies to, and the last part tells it which templating engine to use. Rails comes with three templating engines: ERB (eRuby), Builder, and RJS. This is a new feature introduced in Rails 2.0. Earlier versions were less flexible, and always matched a combination of format and templating engine, so for HTML it would default to ERB by looking up the view `index.rhtml`, and for XML it would default to Builder by looking up the view `index.rxml`. Rails 2.0 gives you more flexibility in mixing and matching formats and templating engines, and also makes it easier to add new template handlers (for example, for using Liquid templates or HAML).

In a moment, we're going to show you Builder, when we use it to create an Atom feed for our tasks list. For XML and JSON, we're not going to go through the trouble of creating and maintaining a custom view. Instead we'll let ActiveRecord do a trivial transformation of our records into an XML document or a JSON object:

```
def index
  @tasks = Task.for_user(@user_id)
  case request.format
  when Mime::XML
    response.content_type = Mime::XML
    render :text=>@tasks.to_xml
  when Mime::JSON
    response.content_type = Mime::JSON
    render :text=>@tasks.to_json
  when Mime::HTML, Mime::ATOM
    # Let Rails find the view and render it.
  else
    # Unsupported content format: 406
    head :not_acceptable
  end
end
```

The preceding code shows the long way of doing things. You can see the short way to respond with different content types in listing 5.7.

Listing 5.7 Responding with different content types

```
def index
  @tasks = Task.for_user(@user_id)
  respond_to do |format|
    format.html
    format.atom
    format.xml { render :xml=>@tasks }
    format.json { render :json=>@tasks }
  end
end
```

← Sets instance variable for use in templates

Renders HTML/Atom using appropriate template

Renders XML/JSON document from object

We're using the `respond_to` method to match each format we support with the logic to render it. It's similar to the case statement in the previous code example, but simpler to specify and more declarative. We're also letting the `render` method do all the hard work by asking it to convert the array into an XML document or JSON object and to set the `Content-Type` header appropriately. It's shorter to write and easier to maintain.

Now it's time to handle the Atom view, for which we'll create a view file called `index.atom.builder`, as shown in listing 5.8.

Listing 5.8 An Atom feed for our tasks list

```
atom_feed do |feed|
  feed.title "My tasks list"
  feed.updated @tasks.first.created_at

  @tasks.each do |task|
    feed.entry task do |entry|
      entry.title task.title
      entry.content task.description, :type => 'html'
    end
  end
end
```

Sets feed title and updates timestamp

Produces one feed entry for each task

The call to `atom_feed` creates an XML document with the right wrapping for a feed, including the XML document type declaration, `feed` element with ID, and alternate link back to our site. It also creates an `AtomFeedBuilder` object and yields to the block. From the block, we're going to create the feed title, specify the last update, and add all the feed entries.

We now have a `tasks` resource that responds to `GET` and returns the task list in different content types: HTML for web browsers, Atom for feed readers, and either XML or JSON for client applications.

Discussion

The HTTP protocol allows clients to request data in a particular format using content negotiation. When the client sends a request to the server, it uses the `Accept` header to indicate all the content types it supports in order of preference. The server can pick the most suitable content type and use it when responding to the client. If the server doesn't support any of the listed content types, it simply responds with 406 (Not Acceptable). Another status code, 415 (Unsupported Media Type), tells the client that the server does not support the content type of a POST or PUT request.

That's the basic idea behind content negotiation. In some cases, it's clearly the right thing to do. We can use one resource URL and send it to all our clients, and each client can see a different representation of the same resource. A web browser will see an HTML page, a feed reader will see an Atom feed, and other applications may see XML or CSV.

Another approach uses different resource URLs for each representation. Some people prefer this approach, since it allows you to manage different representations. For example, if you want to download a CSV document using a web browser, you need a URL that will always send back a CSV document.

There is no one best way to construct these URLs, but there are two common conventions. One adds a query parameter that indicates the expected content type. For example, you can decide to use the `format` query parameter, and use a URL like `/tasks?format=xml`. Another convention is to use an extension suffix on the URL path, such as `/tasks.xml`. We recommend using the extension suffix for the simple reason that saving the document with a web browser will retain the suffix, and a file called `tasks.xml` will always open in the right application.

How does Rails handle this? When we use the built-in mechanism to decide on the content type, as we did in listing 5.7, Rails picks up the expected format from the `format` query parameter, or from the URL path suffix, or from the `Accept` header, in order of preference. Which way you request different content types is up to you—a Rails application can support all three.

You'll notice that in listing 5.6, when we wrote an action to create a new task, we did this:

```
Task.create!(params[:task])
```

Multiple representations work both ways. If we can create a response and send back an XML document, we had better be able to process a request by accepting the same XML document. When Rails processes an XML request, it converts the XML document into a `Hash`, using the document element's name for the parameter name. The preceding example expects the document to contain the element `<task>` and passes the `Hash` to `ActiveRecord`.

It works the same way for HTML forms, if you follow the simple naming guidelines set by Rails. In our forms, we will have fields like `task[title]` and `task[priority]`. Rails uses this naming convention to figure out how the fields relate to each other, and turns them into a `Hash` parameter, so we can use the same line of code to process an XML document or the submission of an HTML form.

It helps that we're using Rails' form helper methods:

```
<% form_for @task do |f| %>
  <%= f.text_field :title %>
  <%= f.text_field :priority %>
<% end %>
```

The `form_for` creates the `<form>` element, figures out the action URL, and takes care to map the field names from `title` to `task[title]`. Give it a new record and it will

point the form to the URL for creating a new resource (`tasks_url`, POST method); give it an existing record and it will point the form to the URL for updating an existing resource (`task_url(@task)`, PUT method). That is why we used `Task.new` to render the form in the `new` action: we can use a single template to both create and update a record. Rails comes with built-in support for HTML forms, XML, JSON, and YAML, and if that's not enough, you can always add custom parameter parsers. Have a look at `ActionController::Base.param_parsers` for more information.

In listing 5.8 we showed you how to use `AtomFeedBuilder`, a templating mechanism for generating Atom feeds. `AtomFeedBuilder` itself extends the more generic XML templating mechanism provided by `Builder::XmlMarkup`. Let's take a moment to look at `Builder` and what you can do with it.

`Builder` is a simple templating mechanism for creating XML documents from Ruby code. Because it always produces well-formed documents, some developers even use it to generate XHTML pages. It's available as a gem you can use in any application that needs to generate XML, and it's also included as part of Rails. `Builder` is very simple to understand and intuitive to use, and it's a good example of what can be done with a little bit of metaprogramming.

When you call a method on a `Builder` object, it takes the method name and uses it to create an XML element with the same name. This is done through `method_missing`, and there is no need to specify any of these methods in advance. `AtomFeedBuilder` only specifies a few methods that do a lot more than just generate an XML element, so it defines `entry` but doesn't bother to define `title` or `content`.

As you can imagine from this example, passing a string argument to a `Builder` object will use that value for the element content, a hash argument specifies the element's attributes, and blocks are used to nest one element within another.

Besides these, there are some special methods you can call, such as `tag!` to create an element with a given name (for example, to handle special characters or namespaces), `text!`, `cdata!`, `comment!` (each of which do exactly what you think they would), and `instruct!` to create the XML declaration at the top of the document.

We mentioned before that you can use different URLs for the various response representations. When we defined the `tasks` resource, Rails created several named route methods like `tasks_url` and `task_url`. What we didn't show before is that, in addition, Rails created named route methods that accept a format and return a URL that specifies that output format in the form of a path suffix. These method names start with `formatted_` and accept an additional argument that specifies the output format, and they will show up when you run the `rake routes` task. Let's add a link that users can use to subscribe to the Atom feed, using a named route:

```
<%= link_to "Subscribe", formatted_tasks_url(:atom) %>
```

In this section, we showed you how to build a RESTful web service. But what if you want to access that service from another application? In the next section, we'll talk about `ActiveResource`, Rail's way of accessing remote resources using an `ActiveRecord`-like API.

Method_missing and BlankSlate

Builder uses `method_missing` in an interesting way. Ruby's objects use method passing—when you call a method on an object, Ruby first tries to match it against a known method definition, and if it doesn't find any method, passes it on to the object's `method_missing`. The default implementation throws `NoMethodError`. Builder, on the other hand, uses `method_missing` to catch method calls and convert them into XML elements, so we don't need to declare an XML Schema or build any skeleton objects to get this simple creation of XML documents from Ruby code.

Existing object methods may clash with XML element names; for example, names like *id* and *type* are commonly used as element names. To solve that, Builder uses `BlankSlate`, a class that has most of its standard methods removed. (In Ruby 1.9 you can achieve the same using `BasicObject`.)

5.2.3 Using ActiveSupport

We started this chapter by showing you how easy it is to use `open-uri` and `Net::HTTP`. Well, easy is a relative term. Building a client library to access our task manager service will still require a fair amount of boilerplate code—more than we care to write, test, and maintain. We also showed you some principles and conventions for designing RESTful web services. In this section, we'll take it a step further and show you how we can use them to develop a client library for the task manager using `ActiveResource`.

Problem

Now that the task manager service is up and running, you need to develop your workflow application. As part of that application, you'll need to create and manage tasks. You want to reuse our task manager service, and you want to get it done before the day is over.

Solution

We'll build a client application that uses `ActiveResource` to access the task manager service. We'll start by writing a class to represent the resources for handling a task list and individual tasks:

```
class Task < ActiveSupport::Base
  self.site = 'https://john:secret@taskmanager.example.com/'
end
```

Remember from section 5.1.2, we're using the URL to specify the username and password for accessing the service, and these map to HTTP Basic Authentication, using HTTPS when we need to access it over public networks.

We've not yet implemented a single method in our new `Task` class, but let's first see what we can do with it. Let's start by creating a new task:

```
task = Task.create(:title=>'Read about ActiveSupport', :priority=>1)
puts "Created task #{task.id}"
=> "Created task 1"
```

Doesn't this code look very familiar? We're using `ActiveResource` here to operate against remote resources, but the patterns are the same as in the previous section, where we used `ActiveRecord` to access the database.

Let's see what happens behind the scenes of the `create` method:

```
task = Task.new
task.title = 'Read about ActiveResource'
task.priority = 1
task.save
```

It starts by creating a new object in memory and setting its attributes, and it saves the object by making a `POST` request to the resource `/tasks`, with an XML document containing the task definition. Our simple implementation, you may recall from section 5.2.1, receives the XML document, parses the attributes, and uses them to create a record in the database. It then tells the client what the new task resource is, which is all our `ActiveResource` needs to know.

Let's follow up by updating the task:

```
task.title << ' and try this example'
task.save
```

This time, since the task already exists, we make a `PUT` request to the resource and updated it. So we can create and update resources. We can also read and delete them:

```
task = Task.find(1)
task.delete
tasks = Task.find(:all)
Task.delete(tasks.first.id)
```

All of this is just a matter of conventions. `ActiveResource` follows the same conventions we used when we built the task manager service, so we got all this functionality just by specifying a URL.

How do we know our `Task` class sends requests to the right URL? We assumed it uses XML by default, but is there a way to find out for sure? Let's try the equivalent of the rake `routes task`:

```
puts Task.collection_path
=> /tasks.xml
puts Task.element_path(1)
=> /tasks/1.xml
```

We built our task manager around all the common patterns, but we also added two resources specific to our task manager. We had one resource for listing all the completed tasks, and we'll want to use that from our client as well. Let's list those tasks:

```
Task.find(:all, :from=>:completed)
```

As you can guess, this is just a request against the `/tasks/completed.xml` path. We also had a resource for quickly updating the task priority, which we designed to support our AJAX controls. Let's try to use that as well:

```
task.put(:priority, nil, 5)
```

This time, the request goes to `/tasks/{id}/priority`, substituting the task identifier in the URL template. The `put` method takes two additional arguments, the first being a hash that is passed along as query string parameters, and the second being the body of the message. Remember from section 5.2.1, we're passing a priority number in the body of the message.

As you might expect, there are other custom methods you can use, like `get`, `post`, and `delete`. We're going to hide the details of `put` from the application by wrapping it in a method; in fact, we'll add a couple more to create an `ActiveResource` class that represents our task manager service. The result is shown in listing 5.9.

Listing 5.9 Using our task manager with `ActiveResource`

```
class Task < ActiveResource::Base
  self.site = 'https://taskmanager.example.com/'
  def self.completed
    find(:all, :from=>:completed)
  end
  def self.update_priority(id, value)
    Task.new(:id=>id).priority!(value)
  end
  def priority!(value)
    put(:priority, nil, value.to_i)
  end
end
```

← Sets the tasks service URL

← Returns completed tasks

← Updates task priority

← Updates task priority on current resource

Now let's try it out by running this snippet using `script/console`:

```
Task.site.user_info = 'john:secret'

puts 'Completed tasks'
Task.completed.map { |task| task.id }.to_sentence
=> "1, 2 and 3"

puts 'Changing priority for task 123'
Task.update_priority(123, 5)
Task.find(123).priority
=> 5
```

Discussion

As you've seen from our examples, Rails makes it extremely easy to build web services that follow the REST principles and work equally well with web browsers and the programmable web. In fact, a lot of that simplicity comes directly from following these principles. We didn't have to tell our client how to create, read, update, or delete the resource—those all followed from using the proper HTTP methods. All we had to do is point our client at the right place. Likewise, we didn't have to build two different task manager applications for people and for applications. We managed both at the same time by using different content types.

If you follow Rails conventions, you get the basic CRUD operations for free. In practice, that's often not enough, and you'll find that you need more specific resources and you'll need to layer additional actions into your controllers. We showed

you how easy it is to add these custom methods on both the server and the client sides. There are, of course, other things you'll need to do. A fully functional task manager would need to handle deadlines and exceptions, send notifications, and even spawn workflows that would involve even more tasks and interact with other services. Those are all possible to do within the constraints of REST; unfortunately, it's more than we can show in the limited span of one chapter.

In the last three solutions, we have talked extensively about Rails, but we want you to take their general principles home with you even if you use other web frameworks or programming languages. The first principle was the recommended practice of building RESTful web services and the benefits that come from following the REST architectural style. The other was the benefit of picking up on conventions, which can help you design better, develop faster, and end up with code that's easier to understand and maintain. If nothing else, there will be less to document. Conventions are not just for Rails; when you're building your own applications, think how conventions could help you work less and get more done.

The SOAP messaging protocol is another way to harness the HTTP protocol and build services that cross languages, platforms, and applications. In the following sections, we'll turn our attention to SOAP using the built-in SOAP4R library.

5.3 SOAP services

When it comes to talking with J2EE, .Net, and legacy applications, the first option that comes to mind is SOAP. And yes, Ruby does come with a SOAP stack called, not surprisingly, SOAP4R.

SOAP4R supports SOAP 1.1 with attachments, and service definitions specified using WSDL 1.1. Security options include HTTP Basic Authentication, SSL/TLS, and a separate library that covers WS-Security (WSS4R). SOAP4R participates in interoperability testing, and if you're working at the level of WS-I Basic Profile compliance, you can expect it to work with the various Java SOAP stacks (Axis, Glue, CXF), .Net, and others.

In the next two sections, we'll cover the basics of using SOAP4R as we implement a simple task manager service and a client application to invoke it.

5.3.1 Implementing the service

The appeal of the SOAP protocol is in crossing language and platform boundaries, so there's no need to build the service in Ruby in order to use it from Ruby. We can easily imagine the task manager being a service implemented in Java and the client application in Ruby, or the other way around. Regardless, we want to make this chapter self-contained, so we're going to show you how to implement the service in Ruby, and also how to invoke it using Ruby.

We're picking up on the same task manager service we developed in the previous sections of this chapter, but this time using SOAP instead of REST.

Problem

You need to develop a task manager service that you can use from a variety of client applications using the SOAP messaging protocol.

Solution

We recommend contract-first service development. In our experience, it leads to more robust services that are easier to reuse and maintain. In contract-first, we start by specifying the functionality offered by the service, typically in the form of human-readable documentation and a WSDL service definition. Once that's done, we use the service definition to start building the service implementation and any applications that need to invoke the service. In fact, once we have a service definition, we can branch off to develop both pieces in parallel.

We also prefer document style with literal encoding, fondly known as *doc/lit*, which makes it easier to evolve the service definition over time, so we're going to use that for our service definition. Since WSDL is verbose and we only have so much space in this book, we'll keep our example to the bare minimum. We'll build our service to support a single operation, for creating a new task, and only care about two parameters, the task title and priority.

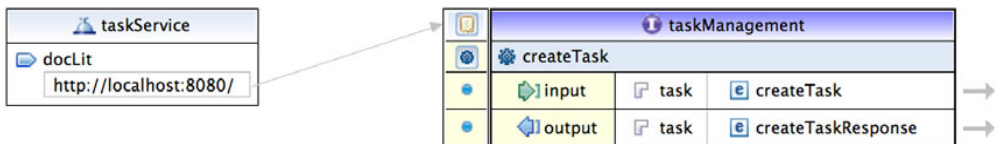


Figure 5.1 Simple task manager service

Figure 5.1 shows a simple outline of the service definition, visualized using Eclipse Web Service Toolkit. It was generated from the WSDL document given in listing 5.10.

Listing 5.10 WSDL describing our task manager service

```
<?xml version="1.0" encoding="utf-8"?>
<definitions name='taskService'
  targetNamespace='http://example.com/taskManager'
  xmlns='http://schemas.xmlsoap.org/wsdl/'
  xmlns:tns='http://example.com/taskManager'
  xmlns:xsd='http://www.w3.org/2001/XMLSchema'
  xmlns:soap='http://schemas.xmlsoap.org/wsdl/soap/'>
  <types>
    <xsd:schema elementFormDefault='unqualified'
      targetNamespace='http://example.com/taskManager'>
      <xsd:element name='createTask'>
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name='title' type='xsd:string'/>
            <xsd:element name='priority' type='xsd:int' minOccurs='0'/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name='createTaskResponse'>
        <xsd:complexType>
          <xsd:sequence>
```

```

        <xsd:element name='id' type='xsd:string' />
    </xsd:sequence>
</xsd:complexType>
</xsd:element>
</xsd:schema>
</types>

<message name='createTask'>
  <part name='task' element='tns:createTask' />
</message>
<message name='createTaskResponse'>
  <part name='task' element='tns:createTaskResponse' />
</message>

<portType name='taskManagement'>
  <operation name='createTask'>
    <input message='tns:createTask' />
    <output message='tns:createTaskResponse' />
  </operation>
</portType>

<binding name='taskManagementDocLit' type='tns:taskManagement'>
  <soap:binding transport='http://schemas.xmlsoap.org/soap/http'
    style='document' />
  <operation name='createTask'>
    <soap:operation style='document' />
    <input>
      <soap:body use='literal' />
    </input>
    <output>
      <soap:body use='literal' />
    </output>
  </operation>
</binding>

<service name='taskService'>
  <port name='docLit' binding='tns:taskManagementDocLit'>
    <soap:address location='http://localhost:8080/' />
  </port>
</service>
</definitions>

```

Like any full-featured stack, SOAP4R allows us to work directly with SOAP messages using the low-level SOAP object model, and to do our own routing between incoming messages and application components. Even for our simple example, doing that would be tedious, so instead we'll use the WSDL service definition to create a service skeleton and extend it with the application logic.

We'll start by creating a working directory for the server side, and run the `wSDL2ruby.rb` command-line tool to create the service skeleton files:

```

$ mkdir server
$ cd server
$ wSDL2ruby.rb --wSDL ../taskService.wsdl --type server --module TaskManager

```

Now let's have a look at the generated files.

The first file, `taskService.rb` defines a class called `TaskManagement`. The name comes from the `portType`, and you will notice that SOAP4R capitalizes the first letter, since Ruby class names are CamelCase. It defines a single operation for creating a new task. When you look at the source code, you will notice that it specifies the input and output messages with all their message parts (one each, in this case), the declared faults (of which we have none), and the document encoding in use. All that information comes from the WSDL document and is used to configure SOAP4R.

The second file, `taskServiceMappingRegistry.rb` defines mapping between the XML elements used in the messages and the Ruby classes holding that data. As SOAP messages come in, they get converted into Ruby objects that our application can easily handle. In this case, the `createTask` element is parsed to instantiate a `CreateTask` object. Likewise, the `CreateTaskResponse` object we return from the method will convert into a `createTaskResponse` element to be sent back in the response message.

Notice that this file contains two mapping registries, one called `EncodedRegistry` (SOAP encoding) and one called `LiteralRegistry` (literal encoding). Since we specified `doc/lit`, only the second mapping is defined and used.

The third file, `taskServiceServant.rb` contains the actual service skeleton. SOAP4R refers to it as “servant,” which simply means the logic behind the service interface. The file defines a single method in the `TaskManagement` class that returns a fault. It’s this skeleton file that we’re going to fill up with application logic to implement the task manager service.

We’re going to keep this example very simple. We’ll specify a couple of classes, `CreateTask` and `CreateTaskResponse`, to hold the request and response messages, and implement the `createTask` method to create a new task record in the database and return the task identifier. You can see the full service implementation in listing 5.11.

Listing 5.11 Our task manager servant

```
require 'taskService'

class Task < ActiveRecord::Base
end

module TaskManager
  # {http://example.com/taskManager}createTask
  class CreateTask
    attr_accessor :title, :priority
  end

  # {http://example.com/taskManager}createTaskResponse
  class CreateTaskResponse
    attr_accessor :id

    def initialize(id)
      @id = id
    end
  end

  class TaskManagement
    def createTask(request)
  end
end
```

```

    task = Task.create :title=>request.title, :priority=>request.priority
    return CreateTaskResponse.new(task.id)
  end
end
end
end

```

The common practice is building a number of services (or servants) and configuring a web server to host them all, exposing each one on a different endpoint URL. For this example, though, we only have one service, so we'll use the quick prototype stand-alone server provided by `taskService.rb`. In addition to the service definition, this file defines a standalone server application called `TaskManagementApp`. The only thing we need to do is start it (see listing 5.12).

Listing 5.12 A simple task manager SOAP service

```

$KCODE = 'UTF-8'      ← Uses UTF-8 all around
require 'rubygems'
require 'soap/driver' ← Requires SOAP4R
require 'taskService'

# Configure ActiveRecord database connection
config = YAML.load(File.read('database.yaml'))
ActiveRecord::Base.establish_connection(config)

# Configure the server
server = TaskManager::TaskManagementApp.new 'TaskManager',
  'http://example.com/taskManager', '0.0.0.0', 8080
# Shutdown when interrupted (Ctrl-C)
trap(:INT) { server.shutdown } ← Catches Ctrl-C, stops server
# Start the server
server.start ← Starts processing SOAP requests

```

Loads configuration for ActiveRecord connections

Creates new server, sets endpoint

We're almost ready to run. We're using ActiveRecord to access the database, so we also need to specify the database connection configuration. We'll do that in a separate file called `database.yaml`, which for our database setup looks like this:

```

adapter: mysql
host: localhost
username: dev
password: dev
database: task

```

Using ActiveRecord outside of Rails

ActiveRecord is a key piece of the Rails puzzle and provides the model part of the Model-View-Controller (MVC) design pattern, mapping Ruby objects to database tables. It's also an outstanding object-relational mapping framework on its own, and you can, as many developers do, use it outside of Rails. All it takes is installing the ActiveRecord gem, requiring it from your application, and using `establish_connection` to configure the database connection.

Now let's start the server:

```
$ ruby server.rb
```

We'll write a client application that uses this service in the next section, so keep the service running and you can test the client application against it.

Discussion

We showed you how easy it is to get started developing SOAP services with Ruby. Before we move to the next section, there are a few more things you'll need to know when building real live services.

SOAP4R comes in two flavors. It's part of the Ruby standard library available in your Ruby installation, and it is also available as a packaged gem that you can install from the public gem repository at <http://RubyForge.org>.

Like any standard library, SOAP4R gets updated with major releases of the Ruby runtime, and as we're writing this book, Ruby 1.8.6 is the predominant runtime shipping with SOAP4R 1.5.5. The gem version is updated more frequently and is currently at version 1.5.8 and pushing toward 1.6. We recommend you stay up with the latest enhancements and bug fixes by installing and using the SOAP4R gem in your environment.

If your service is sending and receiving text in languages other than English, you should consider using UTF-8 encoding. In our experience, it is far easier to use UTF-8 encoding all around than to switch encoding for each document or message. The Ruby global variable `$KCODE` specifies the default encoding used by the runtime, and SOAP4R picks up on it as the default encoding, so make sure to set `$KCODE` to UTF8 before requiring SOAP4R.

Our example was simple enough that we wrote the XML Schema type definitions inline, but for larger services you'll want to create separate type libraries and reuse these definitions across multiple services. This is fairly easy to do by aggregating them into XML Schema documents and importing those documents into various WSDLs. You will also want to use another SOAP4R tool called `xsd2ruby.rb` to read these XML Schema documents and create XML/Ruby mapping files, which you can then reuse in your code.

We did rapid prototyping by letting SOAP4R create a simple standalone server that uses WEBrick, the default web server that ships with Ruby. That's good enough for development and testing, but for production you should consider using Mongrel instead. The easiest way is to install the `mongrel-soap4r` gem, which lets you configure Mongrel for hosting SOAP4R services.

We cannot complete this section without mentioning ActionWebService. It's a lightweight framework on top of SOAP4R designed specifically for use inside Rails applications. It's particularly useful if you want to expose SOAP services as part of a Rails application, and it lets you implement the service operations inside your controllers. You can also use it to invoke SOAP services from within a Rails application.

ActionWebService is also a good fit if you prefer code-first service development and want to define your services from working code. It has a simple, declarative API that feels very similar to the annotations used in J2EE and .Net. We like the

ActionWebService API, but we prefer contract-first design, especially when using SOAP across different languages and platforms, which is why we wrote this example using WSDL and SOAP4R.

Now let's turn our attention to service invocation and write a SOAP client to invoke the task manager service.

5.3.2 Invoking the service

The promise of services is reuse, which emphasizes the client side of the application. In this section, we'll show you how to write a client application to use the task manager service, and not surprisingly, it will be easier and quicker to write than the service itself.

Problem

You want to develop a client application that can use the task manager service described in the WSDL document.

Solution

In the previous section, we showed you how to use SOAP4R and a WSDL file to implement a simple task manager service. In this section, we'll use the same WSDL document to create two clients for that service.

SOAP4R refers to client stubs as “drivers,” and for legacy reasons the base class for all drivers is called `SOAP::RPC::Driver`. But don't get confused—that same driver will also support doc/lit services like the one we're using here.

Since we already have a WSDL service definition, we'll use `WSDLDriverFactory` to create a new driver for the task manager service:

```
wSDL = File.expand_path('taskService.wsdl')
SOAP::WSDLDriverFactory.new(wSDL).create_rpc_driver
```

The driver reads the WSDL service definition and adds all the operations available to the service, along with the endpoint URL and protocol bindings, so we can immediately begin using it. Messages are mapped from their internal SOAP representation to Ruby hashes:

```
response = driver.createTask(:title=>'Learn SOAP4R', :priority=>1)
puts "Created task #{response['id']}"
```

That's all there is to it. We can start adding more operations to our WSDL, create more complex message definitions, all the while using the same basic patterns. You can see the entire client application in listing 5.13.

Listing 5.13 Task manager client using WSDLDriver

```
$KCODE = 'UTF-8'    ← Uses UTF-8 all around
require 'rubygems'
require 'soap/driver'
require 'soap/wSDLDriver'  | Requires SOAP4R
                             | client library

wSDL = File.expand_path('../taskService.wsdl')
driver = SOAP::WSDLDriverFactory.new(wSDL).
  create_rpc_driver
```

← Creates SOAP client from WSDL file

```
response = driver.createTask(:title=>'Learn SOAP4R',
                             :priority=>1)
puts "Created task #{response['id']}"
```

← **Invokes task manager service**

SOAP::RPC::Driver can also map SOAP messages to and from Ruby objects, and when working with larger and more complex operations we prefer that, so the next step is to generate these mappings. Instead of doing it ourselves, we'll turn again to the `wsdl2ruby.rb` command-line tool:

```
$ mkdir client
$ cd client
$ wsdl2ruby.rb --wsdl ../taskService.wsdl --type client --module TaskManager
```

Let's have a look at the generated files.

The first file, `taskService.rb`, defines Ruby classes to represent each element we use in our messages, so in the source code are ready definitions for `CreateTask` and `CreateTaskResponse`.

The second file, `taskServiceMappingRegistry.rb`, specifies the mapping between XML elements and these Ruby classes. We covered that mapping in the previous section, when we showed how it is used by the service.

The third file, `taskServiceDriver.rb`, defines `TaskManagement`, again using the `portType` name for the class name, with the single operation we defined in the WSDL. It also includes a basic driver implementation that loads all the mappings and remembers the default endpoint URL. As before, all that information comes from the WSDL and is used here to configure SOAP4R, and once we generate this stub, we no longer need to reference the WSDL file.

Listing 5.14 shows how we would use the `TaskManagement` driver with the typed message parts in place of the WSDL driver and hashes.

Listing 5.14 Task manager client using generated stubs

```
$KCODE = 'UTF-8'
require 'rubygems'
require 'soap/driver'
require 'taskServiceDriver'

driver = TaskManager::TaskManagement.new
request = TaskManager::CreateTask.new('Learn SOAP4R', 1)
response = driver.createTask(request)
puts "Created task #{response.id}"
```

← **Uses UTF-8 all around**

← **Requires SOAP4R client library**

← **Creates SOAP client (driver) using stub**

← **Invokes task manager service**

Discussion

There are several strategies for working with SOAP services. If you're building an infrastructure piece, you may need to work with the bare metal, using the SOAP object model (classes like `SOAPBody` and `SOAPString`) to create and parse messages directly, making dynamic invocations using a generic `invoke` method. If you're building business applications, you'll want to work at a higher level of abstraction by using WSDL. WSDL documents help you define the service contract, from which you can quickly build client stubs and server skeletons that handle all the details of XML and messaging.

We advocate contract-first design and keeping your services compliant with WS-I Basic Profile as the way to build services that interoperate across J2EE, .Net, Ruby, and many other platforms and languages.

Deploying Ruby services on Java/C WS stacks

At the time of this writing, a few alternatives are emerging for building and deploying SOAP services using Ruby. Unfortunately, these are so new that we did not get a chance to cover them in this book.

One interesting possibility is to use JRuby and scripting support in Java 1.6 to deploy Ruby services on any number of Java-based WS stacks. One example we've seen is based on Axis2, another uses the Tuscany project (an SCA container that allows you to mix services written in different languages). If you prefer using Ruby MRI, have a look at C-based WS stacks that offer Ruby bindings, such as the Ruby bindings for Axis/C, provided by WS02, the lead developer of Axis.

5.4 Summary

In this chapter, we've shown you what you need to know to get started building web services with Ruby. We covered the basics of the web architecture using `open-uri` and `Net::HTTP`, how to build RESTful web services using Rails, and how to exchange messages using the SOAP protocol.

There are a few more libraries worth mentioning. This is Ruby, after all, and a lot of developers new to Ruby are surprised to find out how many libraries already exist for handling common tasks.

Want to talk to eBay's web services? `eBay4R` is the easiest way to get started. How about Amazon's on-demand services? Ruby has libraries for using Amazon S3, EC2, SQS, and SimpleDB. SAP NetWeaver? Have a look at `sapnwrvc`, optimized for NetWeaver web services. Salesforce? You can use the low-level `RForce`, or if you're much more comfortable with `ActiveRecord`, have a look at `ActiveSalesForce`.

We'll show you another example when we talk about asynchronous messaging in chapter 7 and integrate our internal business application with a web service, using `WMQ` and `ActiveSalesForce`. But first, we'll talk about automating communication, starting with e-mail and IM in the next chapter.