

Jeremy McAnally
Assaf Arkin

RUBY

IN PRACTICE



Unedited Draft

 MANNING

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>



**MEAP Edition
Manning Early Access Program**

Copyright 2008 Manning Publications

For more information on this and other Manning titles go to
www.manning.com

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

Licensed to Keith Steward <paypal1@keithsteward.com>

Table of Contents

1. Ruby under Microscope
2. Testing Ruby
3. Integration with Ruby
4. Rails Techniques
5. Web Services
6. Automating Communication
7. Asynchronous Messaging
8. Deployment and Management
9. Databases
10. Structured Documents
11. Authentication and Identity Brokering
12. Indexing and Searching
13. Document Processing and Reporting

Appendix A: Improving Ruby performance

Appendix B: Bridging Ruby

Appendix C: Your Development Environment

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

Licensed to Keith Steward <paypal1@keithsteward.com>

1 Ruby under the Microscope

Often people, especially computer engineers, focus on the machines. They think, “By doing this, the machine will run faster. By doing this, the machine will run more effectively. By doing this, the machine will something something something.” They are focusing on machines. But in fact we need to focus on humans, on how humans care about doing programming or operating the application of the machines. We are the masters. They are the slaves.

YUKIHIRO MATSUMOTO, CREATOR OF RUBY

You’ve heard it all before, right? A new language or framework becomes the flavor du jour and everyone starts talking about it. First there’s a low rumble on websites, then someone gets a hold of it, does something cool, and out comes the marketing speak. I’m sure you can imagine Dave from marketing barking at you about another amazing technology: “You will be more productive! Our synergistic approach to dynamic, domain-driven, development will allow you to get to market quicker and get a better return on investment! Get a lower TCO and higher ROI over J2EE with our XP driven Scrum model based on XML! Take apart your FOB and overhaul your BOB with our easy to use, turnkey solution!” To some in the world of software development, it sounds like Ruby is experiencing hype and buzz, but we’re here and this book is meant to show you that you really can develop “real” software with Ruby.

Maybe you heard the accolades and decided to read this book to find out if Ruby is right for you. Maybe you know Ruby already and you chose this book to pick up some practical techniques you can take back to the workplace. Whatever your reason for picking our book up, we’re glad you did and we hope that we can help you learn more about using Ruby in the real world. But before we get down to the nuts and bolts, let’s take a step back and look at a little history.

1.1 Why Ruby now?

Here’s a little known fact that surprises many people: Ruby came to the world in 1995, the same year as Java. Like many other open source technologies (like Linux and MySQL) it took its time to mature and get noticed. So what happened in those ten years that turned Ruby from a little known language into a hot item ticket without the help of a big vendor marketing machine? The adoption of Ruby on Rails, Ruby’s premier Web development framework, is the obvious answer, and it has without a doubt skyrocketed Ruby’s popularity. It brought on hordes of developers who use Ruby exclusively with Rails, and even more developers who came for Rails, but stayed for the Ruby. But while Rails played a major role in getting Ruby into the mainstream, it still doesn’t explain why it happened only recently, and not before. But one thing that can help explain Ruby’s meteoric rise is the recent rise in software complexity.

1.1.1 Dealing with complexity

If you work for a big company, chances are you have to deal with complex problems. Sales across different channels, multiple products and markets, suppliers and distributors, employees and contractors, accounting and SOX compliance, market dynamics and regulations, and the list goes on and on. It’s unavoidable: the problems of running any sizable business are complex. But what about the solutions? You’re probably thinking, there are

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

no simple solutions to complex problems, and complexity is the nature of any real business. But do solutions have to be unnecessarily complex?

That's true, but given the complexity that naturally arises from these business problems you are trying to solve, you don't want the technology you need to solve them to be unnecessarily complex. Even further, the more technology you throw at the problem – Web servers and databases, on-line and batch processing, messaging protocols and data formats – the more complexity you add. The only way to alleviate this complexity conundrum is by looking for simpler solutions to existing problems.

Optimizing developer cycles

In recent years there has been a growing realization that software companies are targeting these business problems (one might call it the “enterprise” space) by offering overly complex solutions. We can't blame them. Complex solutions sell better. It's easier to obfuscate the solution and abstract the problem to enchant executives into buying your software than to design a solid, simple solution. As more developers are realizing that these “silver bullets” solutions create more problems than they solve, the momentum is shifting towards simpler, lightweight technologies.

OPEN SOURCE

It seems that open source, with its organic development model, is able to adapt to this changing of tides better. For example, in the Java space, once can see a strong bias towards Spring and Hibernate as opposed to EJB. Many developers are defecting from a lot of spaces to Rails. Why? Because those projects aren't afraid to re-evaluate their approaches to accommodate current developer attitudes, especially since these sorts of projects are developed by the developers who use them everyday in their own work.

We like to talk about large scale systems, thousands of servers, petabytes of data, billions of requests. It's captivating, the same way we could talk about horsepower and 0 to 60 acceleration times. But in real life we often face constraints of a different scale. Can you do it with a smaller team? Can you get it done tomorrow? Can you add these new features before we go into beta? Most often in real situations businesses have to optimize not for CPU cycles, but developer cycles. It's easy to scale out by throwing more hardware at the problem, but as many businesses have found out, throwing more people at the problem just makes the project late. Sure that knowledge was captured years ago in Fred Brooks' *Mythical Man Month*, but our bosses just decided to prove it empirically.

Minimizing developer cycles is probably the single most attractive feature for dynamic languages, and Ruby in particular. Simplifying software development has been the holy grail of the software industry. Say what you will about COBOL, it's much better than writing mainframe applications in assembly language. And believe it or not, productivity was a major selling point for Java, in the early days when it came to replace C/C++. It's the nature of the software development that every once in a while we take a leap forward by changing the way we write code to deal with the growing complexity that came since the last major leap. And it's not the sole domain of the language and its syntax. One of the biggest criticisms against J2EE is the sheer size of its API, and the complexity involved in writing even the simplest of programs. EJB is the poster child of developer unfriendly technology.

The true measure of a programming language's productivity is how little code you need in order to solve a given problem. Writing less code while being able to do the same thing will make you far more productive than writing a whole lot of code without doing much at all. This is a reverse relationship compared to how many businesses view productivity: lines of code produced. Of course, if simple lines of code were the metric, Perl would win every time. Just like too much code can make your application unmaintainable, so can terse, short code that's “write only.”

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

Language features

Ruby seems to hit the soft spot and appeal to developers that value natural interfaces and have a strong contingency towards migrating away from languages that inherently promote complexity. But why? Why would developers move away from “proven” technologies to Ruby, arguably, the “new kid,” regardless of its positive aspects? Primarily, it’s because Ruby is a dynamic language that works well for applications and scripting, supports the object-oriented and functional programming styles, bakes arrays and hash literals into the syntax, with just enough meta-programming features to allow domain-specific languages. Had enough marketing? Of course, this laundry list of buzzwords is not as important as what happens when you combine all these features together. They go from buzzwords and abstract concepts to become a powerful tool.

For example, consider Ruby On Rails. Rails is one incarnation in a long heritage of Web application frameworks. Like so many Web application frameworks before it, Rails deals with user interface and remote APIs, business logic, and persistence. Unlike so many Web application frameworks before it, it does so effortlessly without taxing the developer. In its three years of existence it leapfrogged the more established frameworks to become the benchmark by which all other frameworks are judged.

All that power comes from Ruby. The simplicity of mapping relational databases to objects without the burden of XML configuration is a combination of object-oriented and dynamic styles. The ease of writing HTML and XML templates, and setting up filters comes from functional programming. Magic features like dynamic finders and friendly URL routing are all forms of meta-programming. The little configuration it needs is handled effortlessly using a set of domain-specific languages. It’s not that Rails (or really Ruby) is doing anything new, but the attractiveness comes from *how* it does things. Besides being a successful framework on its own, Rails showed the world how to use Ruby’s combination of language features to create applications that, quite frankly, rock. Dynamic features like `method_missing` and closures go beyond conceptual curiosity and help you deliver.

So, why is Ruby popular now? It can for the most part be traced to developers growing weary of complex, taxing development tools and the emergence of Rails as a definite, tangible project that shows that Ruby can be used to create production quality software that’s still developer friendly. Once you start working with Ruby, you’ll probably realize this, too. It’s about the same amount of effort to work with flat files, produce PDFs, make SOAP requests, send messages using WebSphere MQ as it is to map objects to databases, send e-mail, or parse an XML file. Wrap that into a nice, natural syntax, and you have a very potent tool for software development.

1.2 Ruby by Example

We think the best way to illustrate this idea is by example, and what better way than diving into some code? Let’s take a fairly typical situation and do something with it: you need to pull some data from a database and create some graphs with it. Perhaps you need to trace the sales performance of a range of products over across all of your sales locations. Can we keep it simple?

First, we’ll need to install three libraries: Active Record (an object-relational mapper for databases), Scruffy (a graphing solution for Ruby), and Rmagick (ImageMagick bindings for Ruby, required by Scruffy). Let’s do that using the RubyGems utility:

```
gem install active_record
gem install rmagick
gem install scruffy
```

Assuming you have all the system prerequisites for these packages (for example, ImageMagick for RMagick to bind to), you should now have all you need.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

RMAGICK

RMagick can be a beast to setup. We suggest checking the project's documentation, the mailing list, and your favorite search engine if you are having problems.

Now, let's setup our database; Figure 1.1 below shows our schema diagram for the database. You can use that as a model to create the tables (if you prefer to use some sort of GUI tool), or you can use the included SQL below it in Listing 1.1.

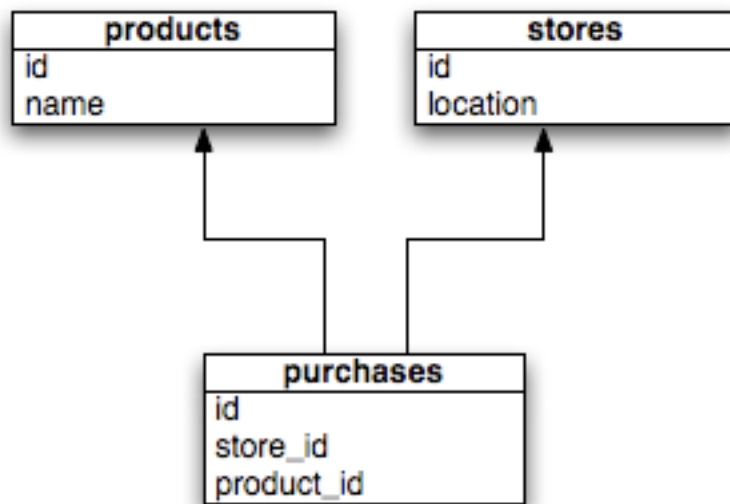


Figure 1.1 For our graph, we will build a simple domain model: Products will have purchases which belong to the stores where they happened.

Listing 1.1 SQL for graph example database

```
CREATE DATABASE `paper`;  
  
CREATE TABLE `products` (  
  `id` int(50) NOT NULL auto_increment,  
  `name` text,  
  PRIMARY KEY (`id`)  
);  
  
CREATE TABLE `purchases` (  
  `id` int(50) NOT NULL auto_increment,  
  `product_id` int(50) default NULL,  
  `store_id` int(50) default NULL,  
  PRIMARY KEY (`id`)  
);  
  
CREATE TABLE `stores` (  
  `id` int(50) NOT NULL auto_increment,  
  `location` text,  
  PRIMARY KEY (`id`)  
);
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

Now, let's set up ActiveRecord to work with the database. ActiveRecord is typically used inside of a Rails application, and since we're not using it in that environment it takes a few more lines of configuration. See our configuration and implementation code in Listing 1.2.

Listing 1.2 Setting up our database with ActiveRecord

```
require 'rubygems' #1
require_gem 'activerecord' |

# Begin ActiveRecord configuration
ActiveRecord::Base.establish_connection( #2
  :adapter => "mysql", |
  :host => "localhost", |
  :database => "paper", |
  :username => "root", |
  :password => "" |
)

# Begin ActiveRecord classes
class Product < ActiveRecord::Base #3
  has_many :purchases
end

class Purchase < ActiveRecord::Base
  belongs_to :store
end

class Store < ActiveRecord::Base
end

# Begin logic
myproducts = Product.find(:all) #4

require 'pp' #5
pp myproducts |
<#1 Require RubyGems>
<#2 Configure ActiveRecord>
<#3 Build our ActiveRecord classes>
<#4 Search for all products>
<#5 Use Pretty Printing to output our collection>
```

As you can see, it doesn't take a lot of code to get a full object-rationally mapped database connection. First, you need to import the RubyGems library (#1) so you can then import ActiveRecord. Next, you need to establish a database connection with ActiveRecord. Normally this configuration data would live in a database configuration file in a Rails application (i.e., `database.yml`), but for this example we chose to run outside Rails, so we'll use `establish_connection` directly. Next we need to create ActiveRecord classes and associations to map our database #3.

Finally, we execute a query (#4) and output its results using Pretty Printing (`pp`). Now you just need to fill in some testing data (or download the script from the book's source code to generate some for you), and run the script. You should see something like the following output.

```
[#<Product:0x639e30 @attributes={"name"=>"Envelopes", "id"=>"1"}>,
 #<Product:0x639e08 @attributes={"name"=>"Paper", "id"=>"2"}>,
 #<Product:0x639c00 @attributes={"name"=>"Folders", "id"=>"3"}>]
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```
#<Product:0x639bb0 @attributes={"name"=>"Cardstock", "id"=>"4"}>]
```

Alright, so our database is set up and our query works, so let's move on to generating a graph from the data. First, remove those last two lines from the previous listing if you'd like (they will be superfluous by the time we're done). Let's take that data that we retrieved, process it, and build the graph using Scruffy; in listing 1.3, you'll see how to do that.

Listing 1.3 Generating the graph with Scruffy

```
graph = Scruffy::Graph.new #1
graph.title = "Comparison of Product Sales" |

stores = Store.find(:all, :select=>"id, location")

myproducts.each do |product| #2
  counts = stores.collect do |store|
    Purchase.count "id", :store_id=>store.id, :product_id=>product.id
  end

  graph.add :line, product.name, counts #3
end

graph.point_markers = #4
  Store.find(:all).map! {|store| store.location} |
graph.render :as => 'PNG', :to => 'productsales.png',
  :width => 720,
  :theme => Scruffy::Themes::Keynote.new) #5
<#1 Create a new Scruffy object and give it a title>
<#2 Iterate the products>
<#3 Add a line to the graph for each>
<#4 Put in point markers>
<#5 Generate the graph with a theme>
```

First, we need to create a `Scruffy::Graph` instance and do a little bit of setup #1. Next, we iterate the products we found earlier and calculate the sales counts for each store location #2. Finally, we add a line on the graph showing the sales trends of that product across the stores #3. Before we render the graph, we need to add the markers to indicate which sales location we're looking at #4; finally, we render the graph to a PNG file using one of Scruffy's built-in themes #5. If you open the graph up and look at it, you can see that we have a very polished graph (ours looks like Figure 1.2).

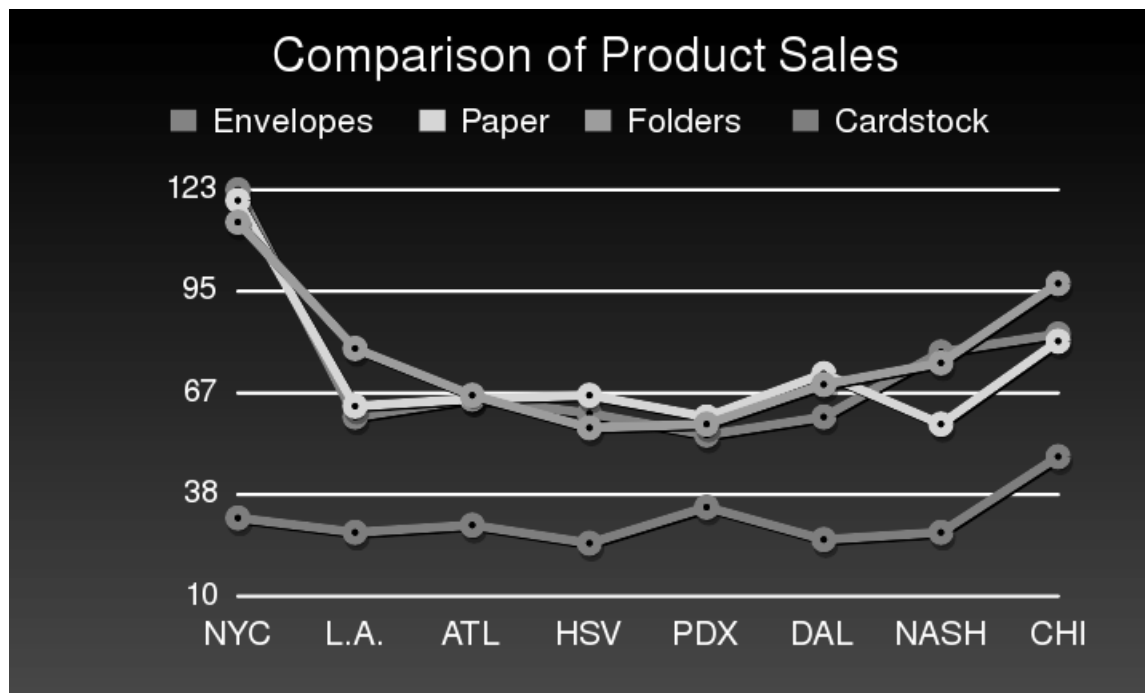


Figure 1.2 Our finished product. In about 40 lines of code we pulled data from the database, processed it, and graphed it in a rather attractive fashion.

Not bad for 40 lines of code, including whitespace, comments, and more verbose than required constructs. Sure this example isn't representative of every situation (you can't develop a full CRM solution in Ruby with 40 lines of code!), but it speaks volumes about the expressiveness of the language and the power of its toolkit. In the next section and chapters, we'll take a look at a lot of the concepts that powered this example, so you can start building applications and tools that take full advantage of Ruby's features.

1.3 Facets of Ruby

Now that we've spent some time discussing the "why" of Ruby, we'll be spending some time looking at the "how." One of the goals of this book is to make you into a truly effective Ruby developer; we want you to become Ruby crafts(wo)men, able to use Ruby to reframe problems and craft solutions. In this section, we'll discuss some Ruby concepts and unique "Rubyisms" that will help you do this and power the examples and libraries you'll see throughout the rest of the book. Our goal is that you'll come away with a grasp of these advanced concepts and how to craft code that is readable, expressive, and "good" (whatever subjective method you use to measure that). If we're successful, you'll be able to use Ruby to do your job more effectively and develop more maintainable applications.

But what do we mean when we say "reframing in Ruby"? Every programming language has its own set of idioms and best practices. Once you're comfortable with the syntax, and know your way around the libraries, you start to explore that which makes the language unique. You explore the character of the language, if you will: the way it promotes a certain style of programming and rewards you for following it. Work with the language and it will work for you.

Object oriented languages, for example, ask you to encapsulate behavior and data into objects, and reward you for that in reuse. If you're coming from Java, you know the value of using JavaBeans, the standard collections library, throwing and catching exceptions, and so on. Today we take those for granted, but in the early days of Java development, many developers would use Java as if it was C or Visual Basic. They wrote code that didn't follow Java idioms, which made it harder to maintain and use.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

Like Java, Ruby has its own set of idioms. For example, in Ruby you use blocks often to keep your code simple and readable. You can write methods that extend classes with new functionality (i.e., meta-programming). You enrich classes with common behavior by mixing in modules, fondly known as *mixins*. You can use *blocks* to abstract loops and even extend methods and reduce code duplication. For example, let's say you were writing a script to interface with an old legacy server; it accepts TCP connections and operates on very simple commands like LOGIN, GET, DELETE, and so on. Each time you start a session with the server, you need the same setup and teardown, but you want to do different things during the socket's connection each time. You could write a number of methods for each sequence of events, duplicating the setup and teardown code in each one, or you could use a block; listing 1.4 shows a simple implementation of this script.

Listing 1.4 Using blocks to reduce code duplication

```
def remote_session
  config = get_configuration_information
  sock = TCPSocket::new(config.host, config.port)
  yield sock                                #1
  sock.close
  log_results
end

# Simply send some data
def send_login                                #2
  remote_session {|sock| sock.send('LOGIN')}
end

# A little more advanced
def login                                    #3
  remote_session do |sock|                  #4
    sock.send('LOGIN')
    received = sock.recv(128)
    if received == 'OK'
      true
    else
      false
    end
  end
end
```

First, we create a method to execute our setup and teardown, with a `yield` statement inside #1. That `yield` statement tells Ruby to execute the block that is fed to the method as a parameter. Next, we create a simple method to send the LOGIN command to our server (#2); note that a block is fed to the method as a parameter. The `sock` parameter is the socket from our “setup” method (`remote_session`) that is given to the block to use. We do the same for the `login` method (#3), except this time we send the login and do something with the returned data (#4). Notice the code duplication we've eliminated by putting all of our setup and teardown into a separate method; this is just one of the many facets of Ruby that make development with it that much cleaner and easier.

Ruby is a dynamic language, and as you get to explore that facet of Ruby, known as *duck typing*, you'll notice that you don't need to use interfaces and abstract classes that often. The way Ruby allows you to extend objects and use constructors, relieves you from working juggling factories. Iteration is often done with blocks and functions, whereas in non-functional languages you tend to use *for* loops and define anonymous classes.

Once you get a hang of all of these ideas, you start reframing problems in terms of Ruby, rather than thinking out a solution in a language you're more familiar with, and then writing it in the Ruby syntax. It will change your thought process from problem to Java to Ruby, or problem to C++ to Ruby to working directly with Ruby. You will often find that reframing problems changes the way you think about code, and you realize new ways to become a better developer. For example, imagine a method that queries the database for products based on different criteria. You can call the method with any number of arguments. If your code looks like `find_products nil, nil, nil, nil, 5, 50, 250`, it's hard to understand exactly what criteria it's using. What does the 5 stand for? What about the 50?

Some languages promote the idiom of method overloading, creating several methods with different signatures, depending on the expected call. Overloading doesn't always work, for example, if all the arguments are integers, you'll want to follow a different idiom, by creating an object, populating its fields and then passing it to the method. Those are all good idioms, but not the one you'll likely to use with Ruby.

Ruby has a convenient syntax for handling hashes, which you can use to simulate named arguments. Instead of method overloading and populating objects, you can write something like `find_products :category => 5, :price => 50..250`. Duck typing even makes it possible to call `find_products :category => storage, :price => 50..250` and let the method extract the identifier from the storage object. As you practice these idioms, you'll notice certain things change. For one, you'll have fewer methods in your objects, making your APIs easier to understand. In fact, you'll have fewer classes to deal with, in this example we've eliminated the need for an object to hold all the properties and a class to define it. With less classes and methods, understanding the overall design of your software and how all the pieces fit together becomes easier.

Our discussion has barely scratched the surface. This book is not an introduction to Ruby, we only wanted to give you a taste for the language. We could go on and on about all of the dynamic features of Ruby, but it would ultimately be redundant. We'll cover some of these concepts, features, and practices in this chapter directly, but throughout the book you'll get to see more Ruby idioms in practice. We hope that through example and explanation, we can help you learn those by example. The more you invest in learning the language, in getting under the hood and really getting your hands dirty with Ruby, the better developer you'll be. There's always a learning curve, but fortunately with Ruby, there's not a lot to it. You'll see the benefit of Ruby as you're learning and practicing the language, but you'll want to take that information and go deeper to get even better.

One of the killer apps for Ruby is the Ruby on Rails Web application framework. Developers are very attracted to the magic of Rails the productivity gains therein, but, in reality, most of the "magic" is really just good Ruby programming applied to Web applications. Think about this: what if you could make the same magic work for you in other domains, to make you a better developer in any problem you need to solve? What if you could take those same productivity gains and "magic" methods and use them in *your* code? Let's take a look at some of the facets of this gem we call Ruby that can make this possible: duck typing, simplicity, efficiency, and functional programming.

1.3.1 Duck typing

Ruby uses dynamic typing rather than static typing. In a static typing system, an objects' type is determined by its class definition at compile time. Static typing forces each object to declare its heritage, you're always asking, "Where are you coming from?" In dynamic languages behavior is captured by the object not the interface. Dynamic typing only cares about merits, the question to ask each object is, "What can you do?"

You can do that same with reflection in Java or C#, but reflection hides your business logic in a haystack of type-bypassing code. With dynamic typing you don't have to declare so many interfaces and abstract classes

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

in anticipation of reuse; you don't have to write adapters and decorators as often; you don't need to choose between readability and functionality. All these help you reuse code more often.

A by-product of duck typing is that method calls do not check their recipient's type beforehand. Your code calls a method, and if it fails, raises an exception. This concept sounds a little cloudy, so let's look at a piece of code to explain it. Let's say we have a method that calls *size* and returns it in a friendly message.

```
def print_size(item)
  puts "The item's size is #{item.size}."
end
```

Our method simply calls *size* on the object without regard for its class; so if you feed it an object that responds to the *size* method, it will work.

```
mystring = "This is a string."
print_size(mystring)           # => The item's size is 17.
myarray = [1,2,3,4,5]
print_size(myarray)           # => The item's size is 5.
myfile = File::Stat.new("readme.txt")
print_size(myfile)            # => The item's size is 432.
```

It's called *duck* typing because an object that walks like a duck and quacks like a duck must be a duck. If so, then we should assume it is, in fact, a duck. Who are we to discriminate between a Mallard and a CanadianGeese? They both quack, right? Who are we to discriminate between a Man and a Woman? They both talk. We're expecting the object to do something, and we only have to ask: does it do that?

Duck typing is a trade off. You no longer have a compiler that will catch type errors upfront, but you do have fewer opportunities for errors. In a statically typed language, you'll repeat the type declaration in multiple places: class definition, variable declaration, constructor, method arguments, etc. If you need to refactor code, say splitting a class into an interface and separate implementation, maybe add a factory or write a decorator, you end up making type changes in multiple places, and you'll want a type checking compiler to help you minimize errors.

Why is that not a problem with Ruby? As you grow more familiar with Ruby you'll notice that you just don't have to declare types that often, or repeat them all over the place. You rarely need to separate interfaces and implementation classes, or to conjure factories and decorators. When you have fewer types to deal with, type checking is less of an issue. What you get in return is being able to do more with less code, which means fewer places for bugs to hide. Give it a shot, we'll doubt you'll miss type checking.

1.3.2 Keeping It simple

Ruby values simplicity. To be fair, all programming languages do, each striving for simplicity in its own way. They don't all achieve it in the same way. A language cannot enforce simplicity any more than it can keep your code bug free and as simple as possible to understand on its own terms, but it can certainly reward you for keeping things simple. It can also reward you for making things complex, often in the form of over-engineering.

As much as we hate to admit it, once you write a piece of software and release it to the world, the software becomes harder to change. Yet, it often needs to change: new features, switching databases, supporting more protocols, integrating with other systems. And so we plan for change. Each language has its own patterns that deal with change. In a statically typed language like Java, you will need to think these requirements upfront: once the implementation has been fixed, it is very hard to change. You tend to figure out the interfaces upfront,

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

use factories liberally, allow for decorators, and so on. Because those changes are hard to do later on, you're better off doing them up front, just in case. Even those cases that will never happen. In their own way, statically typed languages reward you for over-engineering.

In contrast, being a dynamic language, you'll find that it's much easier to make local changes with Ruby, without affecting the rest of your code. When change is very easy to make, you don't have to plan as hard for every eventuality. In dynamic languages there's less need to design interfaces that are separate from the implementation, because it's possible to refactor the implementation without breaking code all around. You won't need to bury the business logic in layers of factories, decorators, listeners and anonymous classes. That might seem hard to imagine if you have a strong background with statically typed languages, but as you get comfortable with Ruby you'll notice it too. Ruby will reward you for keeping things simple, and saying no to code you don't need will reward you with quicker development and easier maintenance.

1.3.3 DRY efficiency

Ruby is a DRY language. And we don't mean that Ruby is a bad date that can only tell lame jokes and talk about itself. Rather, Ruby helps you keep your code short and concise. DRY stands for: Don't Repeat Yourself. Syntactically, it's a very efficient language: you can express the same thing with less lines of code. As we know, computers are fast enough that more lines of code do not slow them down. But what about you? When it comes to debugging and maintaining, the more code you have to deal with, the harder it is to see what it does and find the problems that need fixing.

Listing 1.5 shows this with a simple example. The first style is Ruby, but you'll notice it looks similar to many other programming languages. The second style is the preferred way of doing it in Ruby: shorter and less repetitive (and even to that end, if this is the last value in a method, the return is superfluous). There's not a lot to this example, but imagine that you could do this throughout your code, eliminating thousands of lines of unnecessary cruft.

Listing 1.5 Two approaches to syntax

```
# The long way
record = Hash.new
record[:name] = "Dave"
record[:email] = "admin@net.com"
record[:phone] = "555-1235"
return record

# The Ruby way
return { :name=>"Dave", :email=>"admin@net.com", :phone=>"555-1235" }
```

Ruby's syntax has a lot of little features like this that end up giving you huge gains: blocks, iterators, open classes, and more. Part of these features, though, are due to Ruby's ability to do functional programming.

1.3.4 Functional programming

Ruby is an object-oriented language: it's objects all the way down. Like many other dynamic languages, functions are also first class citizens; couple that with outstanding support for closures, and it's easy to adopt a functional style of programming. But unlike more traditional functional programming languages, like LISP or Haskell, Ruby is easier to pick up, and can let you enjoy both worlds of functional and procedural style.

Why is functional programming so important? For one, it helps you write shorter and more concise code, some things are easier to express in functional style. Listing 2.2 shows an example of that. For another, the functional style leads to code that doesn't depend on state and has less side effects. Code like that is much

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

easier to mold and refactor, and gives you more opportunities for reuse. It also makes it easier to build applications that scale.

The easiest way to build software that scales is using the “shared nothing” architecture. The less shared state you have to deal with, the easier it is to scale. Although Ruby comes with modern libraries that support threads, locks, mutexes and other concurrency mechanisms, it helps that you don’t have to use them often. Perhaps you’re familiar with the Google MapReduce algorithm that achieves unparalleled scalability by running tasks independently of each other. Yes, it’s the same “map” as the one in Listing 1.6; the “reduce” part in Ruby is most often done using the *inject* method.

Listing 1.6 Three styles of iteration

```
# Code that iterates in order to map one array to another
application_names = []
for application in applications
  application_names << application.visible_name
names

# Code that maps one array to another
applications.map { |application| application.visible_name }

# An even shorter way to express it in Rails (and possibly Ruby 1.9)
applications.map &:visible_name
```

Code that has less dependencies is easier to run in parallel, taking advantage of modern multi-code CPUs, or when deploying on a cluster of servers. There are other ways to do it, but functional programming makes it extremely easy.

Now let’s take a look at one of the most attractive features of Ruby; in covering these various facets of Ruby, we’ve been working our way towards one of the biggest features of the language: meta-programming.

1.4 Meta-programming

We talked in the beginning about reframing solutions in terms of Ruby. Software development is the distance between your ideas, and applications you can actually use. The shorter that distance is, the more often you can travel down that path, the more you can carry with you. We’re going to stop the analogies here, but the point we’re trying to make is, you want to be able to translate your ideas into code quickly and easily.

Thinking out solutions in terms of Ruby helps you do that. But you can do even better if you bring Ruby closer to the way you imagine solutions. You can turn Ruby into a language that has a richer syntax and that is more expressive, so you can easily articulate what you want to do. We’re talking, of course, about meta-programming.

Most programming languages are content being what they are. Some languages are more generic, but they force you to deal with more details. Other languages help you deal with a specific domain problem, but tend to be simple and inflexible. Languages like SQL, RuleML, XML Schema, Ant, CSS are three sorts of languages, and they are what we call domain-specific languages (which we’ll talk about in the next section). But through meta-programming, Ruby allows you to have a mixture of both. You can extend Ruby with mini-languages that help you stay close to the problem you’re solving, yet be able to use all the expressive power of Ruby.

This book is not an introduction to Ruby; what we want to talk to are the different technologies you can use to build business applications, integrate systems, and manage data. But there is one very powerful tool that will let you crank your developer volume all the way to 11, and give you the ability to solve problems with

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

ease. And we think it's worthwhile to spend the next section introducing you to meta-programming and domain specific languages.

1.4.1 Getting started with meta-programming

"Programmers are devices for turning coffee into software" (at least the ones we know!). But jokes aside, software development is not about producing code, even if the recent trends toward outsourcing would lead you to believe that. As we've said before, software development is the distance between your problem and a usable piece of software to solve it. In fact, what will make you an even better software developer is knowing how to traverse this distance by writing less code. Not that code itself is evil, but the less code you have to deal with, the less time you spend debugging it, the less pain it is to maintain, the more time you have to solve new and interesting problems. And enjoy sunny days outside the office.

The software industry always looks for the Holy Grail, that one silver bullet that can turn ideas into code, without the "overhead" of software developers. We're not too worried about losing our jobs. For all the hype and interest, software that writes software is the impossible dream, but software that writes software is also a wonderful tool, even if it only takes on part of the workload.

The thing is that software like that exists. Compilers that turn source code into machine code do that, they let us work with higher level languages. Source code generators and IDE wizards give us a head start by writing boilerplate code and skeletons. Then, even further up the ladder, there's *meta-programming*, writing code that writes code.

Methods that define methods

If you come from a background in Java or C#, you learned that objects and classes are different in one fundamental way. Objects are mutable, so you can change them by calling methods on the objects, but in Java and C#, classes are immutable: their definition is written down in text and once compiled, cannot be changed. In Ruby, classes like objects are mutable. You can call a method on a class that will change the class definition.

Let's look at a simple example. We're dealing with a `Project` that has an `owner`, an attribute you can both get and set. Listing 1.7 shows two ways to express that. You can write a couple of methods, or you can call `attr_accessor`. By calling this method on the class, we allow it to change the class definition, and essentially define a method for getting the value of the instance variable `@owner`, and a method for setting it.

Listing 1.7 Let `attr_accessor` define accessor methods on your class

```
# Define the accessor methods yourself
class Project
  def owner()                               #1
    @owner                                  |
  end                                       |
                                           |
  def owner=(who)                           |
    @owner = who                           |
  end                                       |
end                                         |

# Let attr_accessor define them
class Project
  attr_accessor :owner                      #2
end
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

The above class definitions both do the same thing: define an attribute owner that you can get and set. But the second just uses a lot less code for us; this seems a little contrived now, but imagine having 12 or 15 attributes in a class. Then you're going from 40 or 50 lines of code down to 1. That's *huge* spread across an entire application. But how does it work? Let's take a look at an implementation of `attr_accessor` in Ruby (this is not *the* implementation from Ruby, but it has the same effect) in listing 1.8.

Listing 1.8 A re-implementation of `attr_accessor`

```
class Module                                     #1
  def attr_accessor (*symbols)                  |
    symbols.each do |symbol|                   |
      class_eval %{                             #2
        def #{symbol}                          |
          @#{symbol}                          |
        end                                    |
                                           |
        def #{symbol}=(val)                    |
          @#{symbol} = val                    |
        end }                                  |
    end                                        |
  end
end
```

Using Ruby's open classes, we re-open the class definition of `Module` (#1) and add a method, `attr_accessor`, to it. This method takes each provided symbol and executes a code block in the context of the class definition using `class_eval` #2. The `*_eval` family of methods is a family of methods you will become very familiar with in your meta-programming; check out Table 1.1 for a table of their usage.

Table 1.1: The eval family of functions allows you to dynamically generate and execute code.

Method	Usage
<code>eval(str)</code>	Evaluates a string of code. <code>eval("puts 'I love eval!'")</code>
<code>instance_eval(str)</code> <code>instance_eval { }</code>	Evaluates a given string of code or code block, changing the receiver of the call to <code>self</code> . <pre>class Hider def initialize @hidden = "Hello, world!" end end my_hidden = Hider.new my_hidden.instance_eval { puts @hidden } # => Hello, world!</pre>
<code>class_eval(str)</code> <code>class_eval { }</code>	Evaluates a given string of code or code block in the context of a class's definition. <pre>def printable_attribute(attr) class_eval %{ def p_#{attr} puts attr.to_s end } end class Printer attr_accessor :name printable_attribute name end</pre>

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```
end

my_printer = Printer.new
my_printer.name = "LaserPrint 5000"
my_printer.p_name # => LaserPrint 5000
```

Now that we have our method added to the class definition, every instance of an object has this available to it.

Now let's look at a more complex example. We're further along the development of our application, and we realize there will be a lot of projects to manage. We want to help users by giving them a way to tag projects, so they can then find projects through tags and through related projects. So we need a separate table to hold the tags, and we need to add methods to retrieve projects by tags, delete tag associations when a project is deleted, and so forth.

That's quite a lot of work: a lot of database work with inserting new tags, editing and deleting tags, not to mention search. We decide to re-invent the wheel some other time, and instead we download a plugin called ActsAsTaggable. Listing 1.9 shows how we use it.

Listing 1.9 Using ActsAsTaggable to get a lot of features in one line of code

```
class Project < ActiveRecord::Base
  acts_as_taggable
end
```

Why is this a more complex example when it's not longer than the previous one? Because `acts_as_taggable` adds a lot of functionality to our `Project` model that would take us several days to do right. Now our `Project` model has access to a set of tags in the database, along with methods to search with (e.g., `find_tagged_with`) baked right in. We can call `Project.find_tagged_with(:all => 'ruby')` and get an array of models that have the tag 'ruby.' It required very little code on our part: that is the power of meta-programming.

Ruby is flexible enough that in addition to defining new methods, you can extend existing methods. Think of it as Aspect Oriented Programming baked into the language. It can also let you define complete classes. From the simple `Struct` which declares a bean-like class in one method call, to more complicated mechanisms, for example, creating a collection of model classes from your database schema.

Implementing methods dynamically

This style of meta-programming happens at class definition, but meta-programming can also be used to dynamically alter objects. With languages that have static- (or early-) binding, you can only call a method if that method is part of the class definition. With languages that have dynamic- (or late-) binding, you can call any method on an object. The object then checks its class definition to decide how to respond to the method call.

This means that in Ruby, if the method is not part of the class definition, the object falls back by calling `method_missing`. Typically, `method_missing` will simply complain by throwing an exception, but you can override it to do more interesting things. Let's try to create an XML document (let's say an RSS feed for our projects) using `XML::Builder`; take a look at Listing 1.10.

Listing 1.10 Building an RSS feed for our projects

```
xml = XML::Builder.new
xml.rss "version"=>"2.0" do
  xml.channel do
    xml.title "Projects"
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```

xml.pubDate CGI.rfc1123_date(Time.now)
xml.description "List of all our projects"
for project in projects do
  xml.item do
    xml.title project.name
    xml.guid project_url(project), :permalink=>true
    xml.pubDate CGI.rfc1123_date(project.created_on)
    xml.description project.details
  end
end
end
end
end

```

If you're looking for the step where we use a DTD or XML Schema to generate source code, you won't find it. Builder uses `method_missing` to create elements based on the method name, attributes from the method arguments, and text nodes from the first argument, if it's a string. So, a very simplified version of that method might look like listing 1.11.

Listing 1.11 A simplified look at XML::Builder's use of `method_missing`

```

def method_missing(sym, *args, &block)
  root_element = sym #1

  args.each do |arg| #2
    case arg
      when Hash #3
        build_elements(root_element)
      when String
        add_element(root_element)
      end
    end
  end
end
end

```

Using Ruby's open classes, Builder overrides the method `method_missing`. Typically this method will just raise an exception since it is called when a method called on an object doesn't exist, but when overridden, can add interesting functionality. In this case, Builder takes the name of the missing method that is called #1 and the value provided #2 and makes elements out of them depending on the value's type #3. You'll likely find coding like this sprinkled throughout your favorite and most used libraries, including ActiveRecord.

Builder is one of those staple libraries you'll no doubt use quite often in your projects and an interesting library to analyze if you're interested in meta-programming. Doing so can help you come to a place where meta-programming becomes a tool that can help you build mini-languages that you can use to reframe business logic in terms of the problem domains rather than a language's syntax. We're talking, of course, about *domain specific languages*.

1.4.2 Domain specific languages

Functions, objects, libraries and frameworks all help you work at a higher level of abstraction, closer to the problem. If you look at the software landscape, you'll see a lot of different specialty domains. You can imagine languages, each of which is designed to solve a specific set of problems, by expressing solutions in a way that's easy and natural for that specific domain. Perhaps you never heard of domain specific languages (or DSLs for short) before reading this book, but you certainly have used them. Do any of the following names

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

sound familiar? SQL, regular expressions, HTML, Make, WSDL, .htaccess, UML, CSS, Ant, XSLT, Bash. These are all domain specific languages.

Within the domain of relational databases, there needs to be a way to define new tables. There needs to be a way to query these tables and return the results, to create new records, to update and delete existing records. You could do those things by delving into low-level database APIs, creating the table structures directly, iterating over B-Tree indexes to fetch records, performing joins in memory and sorting the data yourself. In fact, many system developers used to do just that; nowadays, we use SQL.

But domain specific languages have a limit: it's very hard to create a new programming language that has good support for variables, expressions, functions, objects, and all the tooling around it. So domain specific languages tend to be very simple, static and inflexible. HTML, for example, can express rich multimedia content, including text, images, audio, and video. But if you want to do anything dynamic, like pull down menus, partial updates, drag & drop, you need a more generic programming language. You'll want to use JavaScript. On the other hand, because JavaScript is a generic programming language, it will take a mountain of statements to create a page using the DOM API. But what if you could mix a generic programming language with a domain specific language?

Let's look at XML Schema definitions for example. The XML Schema language was originally designed to validate XML documents. With it, you can express what a valid XML document looks like, so you can check XML documents against these rules before deciding whether or not to process them. Granted, it's much easier to use than iterating over the DOM and checking whether the current element is allowed to follow the previous element, and whether it has all the right attributes.

But, like most domain specific languages, XML Schema has its limits. For example, you can validate that a customer element has optional contact data, such as e-mail, phone or IM handle. But there's no easy way to require at least one of these elements to exist. There's no easy way to validate that all US addresses have a state, or that the zip code matches the actual address. But creating a DSL using a powerful host language (like Ruby), we could create something that looks like listing 1.12.

Listing 1.12: A validation domain specific language example

```
contact_information_verification do |the_persons|
  the_persons.name.is_required

  the_persons.address.is_required
  the_persons.address.must_be(10).characters_long

  the_persons.phone.must_be(10).characters_long

  the_persons.im_handle.must_not_be(in_existing_accounts)
end
```

Using this, we have a nice set of readable rules that can be updated by anyone (probably even your secretary!), which will generate or execute the validations we need. DSLs can bring the same simplicity and abstraction to a lot of your specific problems. Think about code that expresses business rules or composes tasks, and you'll find many opportunities to simplify and reduce noise. DSLs don't just help your secretary or business manager keep your code up, but it can also help you as a developer keep yourself sane. Using Ruby's metaprogramming capabilities, you can build these sorts of solid, literate tools and fluent interfaces.

EMBEDDED DSLS

What we've created here is not *technically* a DSL in the purest, scientific sense, but an embedded DSL (EDSL) – a domain specific language that's embedded inside a host language, in this case Ruby.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

Throughout this book, we'll show different Ruby libraries and tools that include their own mini-languages, all of which are EDSLs.

1.4.3 Refining your meta-programming

If you never worked with meta-programming before, you're probably asking some of the same questions we asked ourselves: Where do I start? What's a good meta-programming style? Should I worry about methods that are not explicitly defined? How do I know my code works, can I even test it?

The place to start is with the problem you solve over and over again. When you see a lot of repetition in your code, when you find yourself getting bogged down by details, that's the right time to simplify things. Meta-programming helps you simplify by creating an easier, more expressive way to write code. There's no point in writing a DSL that you will only use once, but it pays several times over when you use it repeatedly.

You could let your imagination run wild and use meta-programming to solve problems you think you'll have someday, but you'll quickly realize those are just mental exercises. Instead, look for patterns you have in your code right now, for practices and idioms you use often, and use meta-programming to simplify those. It's called extraction. Extraction comes from what you need and use, not what you could imagine doing some day. You'll notice that Rails, Rake, RSpec and many of the other frameworks we cover in this book all came from extractions, from looking at existing solutions and finding better, easier ways to write those.

The best way to write a DSL, is to practice "intentional programming". Step away from the code for a minute and ask yourself, "If I were not limited by the language I use right now, if I could express the solution itself in some other language, what would my code look like?" Try and write that code as if that language already exists. You've just defined a DSL. Now you have to go and implement it, but the hardest part is behind you.

So how do you know you're successful? A good DSL has two interesting qualities. The first, you'll want to use it. You know you're successful when, out of all the possible things you can do that day, just after coffee, you decide to write code using that DSL.

The second quality of a good DSL is that you can throw away the documentation and still use it. It's called "the element of least surprise". When the code comes naturally to you, it's easier to write and less painful to maintain. When we work with Rails, we never have to stop and ask "How do we access the `order_on` field?" We know that in Rails, it's accessed with the `order_on` method. We never have to ask "How do I find an employee record by their name?" We know to use `find_by_name`. The rule of least surprise is one of those "I'll know it when I see it": just ask your team mates what they expect to happen. And if you follow our advice for intentional programming, you'll find that it's an easy principle to follow.

Using `method_missing` and a few other fun meta-programming tricks, you may work yourself into a position where you have a lot of methods that aren't explicitly defined. How do you deal with that? You document, of course. And if you follow the rule of least surprise, you have little to worry about: other developers will intuitively know which methods to expect, when it all makes sense.

So, what about testing? Don't the dynamic features make it harder to test? Testing meta-programming is no different from testing any other piece of code. There's nothing magical about it, no hidden traps. You can use any faculty available for testing, in fact, in the next section we're going to talk about testing at length, and cover a testing framework called Rspec, which uses meta-programming to help you write tests efficiently by expressing the expected behavior of your software.

1.5 Summary

Ruby has gained a lot of popularity in recent times for being a simpler, expressive alternative to other contemporary languages. This success is partially due to the emergence of Rails as a tangible tool that

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

developers can use to develop production quality software that's still developer friendly. Rails owes this friendliness to Ruby, which offers a lot of features, such as meta-programming, that make Ruby more expressive and even fun.

As more and more developers pick up Ruby, they will want to use their current programming paradigms like aspect-oriented programming or declarative programming and development practices such as Scrum or test-driven development. Ruby has a number of libraries to help the process for many of these (such as Ruleby or AspectR). In the next chapter, we'll concentrate on the most prevalent of these, test-driven development, and look at testing your Ruby code using the built-in testing library and some related third-party tools and libraries.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

Licensed to Keith Steward <paypal1@keithsteward.com>

2 Testing Ruby

How do you currently test your code? From talking to people, it seems the more popular testing methodology is “Works Now, Works Later”: test your software now by simply playing around with it and if it works now, assume it always will. If this is your approach to testing, then you’ve surely encountered many of the problems we did when we used that method: changes to one part of the system breaking another, unexpected edge cases (or not so edge cases!) causing the software to behave unexpectedly, and the list goes on. Then comes the inevitable debugging after one of these issues crops up. The whole business is a huge time and effort sink.

You can avoid the entire situation and gain some more assurances along the way by simply using automated testing. Ruby makes it easy by including a unit testing library in with its standard distribution (and more tools via third party libraries). But before we get to use that library, let’s take a look at the basics of software testing.

2.1 Testing principles

Automated testing is a process in which code that is used to test the behavior of other code is written. So, for example, if you wrote a shopping cart web application, then you would write code that would call the various pieces of the application and verify its results (e.g., ensure the `calculate_total` method gave you a correct total), test the all the “moving parts” of you application (e.g., call a piece of code that uses the database to ensure your database code is behaving properly), and so on. These sorts of tests are called automated because, well, they are. You write the test code, then you run it (typically inside of a container specifically meant for testing), and it tests the code it’s written to test. This means no more clicking around on your application for 10 minutes to ensure that one feature is functioning properly; no more writing that fake input file and hoping it treats other input files the same; and no more wasting hours of time trying to nail down what part of your big system is causing that little error to pop up (OK, so maybe not “no more,” but at least a lot less!).

So what exactly do we mean by a “test”? A test at its core is essentially a piece of code that ensures that another piece of code does what it’s supposed to. In testing, you’re seeking to setup a test state in your application and then check for (the right) changes to that state after the tests run. You write tests that will verify the results of a method or other value-bearing code to ensure that in most cases, the input is handled properly. So, for example, if you wanted write tests for your method `get_results_as_string`, you would write a test to ensure it returned a string. You would also write a test that made sure it raised exceptions with the wrong input as a safety guard: you want it to complain when it’s improperly used (you want to test that which works, and that which doesn’t). You are testing the behavior of your code in an isolated environment to ensure it behaves as it should. This may sound abstract now, but we’ll take a look at what these tests look like in Ruby a little later in this chapter.

2.1.1 Why bother with testing?

Many developers avoid testing because of all that extra code, and we’ll certainly give it to you honestly: testing creates more code. But don’t let that scare you! While testing may cost you some keystrokes, you’ll appreciate it later when it saves you some brain power and time. It keeps your other code in line; it lets you know if it’s still behaving the same rather than having to either think really hard about whether it is or not or, even worse, finding out it’s not when you’ve deployed it and it crashes hard.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

And so the first benefit of testing is obvious: you know that the code works. Of course you could just run the code and see for yourself, there's no need to write automated test cases. But then consider the huge time sink that would become: spending hours a day poking at your application, making sure each piece is working properly. Using automated tests gives you peace of mind and a productivity boost because if the tests pass (and they're well written tests) you can be assured your code is behaving properly.

But as we all know, code has a sensitive personality: you fix things in one place, and it may get upset and break in another place. Testing helps you maintain the behavior of your application; if you add code or refactor, how can you be sure that your application will really behave the same unless you test it? And what better way to test it than with consistent, automated tests? Let's say you refactor a class's methods and cut them down from 13 to 6. Can you be sure that the methods that are exposed to the rest of the application still behave the same? If you've written a solid test suite, it should be easy. Just run the tests, and if they all pass, it should presumably work in the same fashion as it did before.

Another benefit of automated tests is that they help with debugging. Some bugs are notoriously hard to find when you have to repeatedly run the code, each time going through manual steps like creating and deleting data to make the bug surface, fixing and testing again to see if you got to the root of the problem. Test cases are more granular, and tell you exactly what they're doing, so by running the test cases you can find bugs and fix them easily.

TESTING AND DYNAMIC LANGUAGES

Many critics of dynamic typing say that debugging dynamic languages is harder because the compiler doesn't help you with debugging through type checking, but testing can help you get over a dependence on this static compiler behavior. Since Ruby objects are defined by behavior rather than a static definition you can use tests to verify their behavior more effectively than by relying on the compiler. Compilers don't test for nulls, string values, integers out of range, partially constructed objects, and are easily fooled by type casting. Testing gives you the same level of granularity with a much wider test spectrum, and as such, more information when trying to debug.

Now that we've looked at some of the basic principles of testing, let's take a look at testing in Ruby. The "burden" of testing is much lighter in Ruby thanks to its built-in testing libraries and third party tools. In this section, we'll take a look at these tools and how they can help you effectively write tests and code faster and easier.

2.1.2 Types of testing

When most people discuss testing, they are usually talking about *unit tests*. Unit tests test your code in small, discrete units: methods or classes or whatever small units of code that you care to test. These tests the core of all testing, since they test your code at its lowest level and verify its basic, core behaviors. Unit tests are the "bread and butter" of testing; typically these will be the first tests you write and act as the foundation that other test types are built from. You wouldn't want to write tests for a database application and all of its queries without first writing a test to ensure it can connect and make a query, right? Unit tests should be very concentrated; that is, they should test the smallest unit possible: test a single method at a time, a single call signature at a time, a single class at a time, a single module at a time, and so on. In doing this, you create a set of tests that depend only on what's being tested, thus making their results more accurate. This way the `foo` method can't make the test for `bar` fail.

ISOLATING UNITS

One of the first difficulties most developers have with testing is gaining this separation of concerns. Don't worry! There are plenty of techniques and libraries to help you out. Further in this chapter, we discuss some of these strategies.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

The second type moves a little further up architecturally; these tests are known as *integration tests*. These tests involve larger pieces of code; you could think of them being something like a compound unit test. As the name suggests, they are tests that test the integration of your application's pieces. For example, Rails' integration tests allow you to test an entire application flow, from logging in, through filling a form to getting a page with search results. Other integration tests could test a full application operation, like an e-commerce transaction or a sequence of actions like downloading, extracting, and using data. The purpose is to make that all of the "moving parts" of your application like your code, the database, any third party code, and so on, are all moving in the right direction.

There are many other types of testing (acceptance testing, regression testing, and so on), but these are beyond the scope of this book (and it could be that you will never need to write any of these sorts of tests because you have a testing/QA team). The focus of the rest of this chapter is to practically show you how to unit test your code using Ruby and its libraries, and we'll start in the next section by walking you through a basic testing workflow.

2.1.3 Testing workflow

Developers typically don't test as much as they should. The excuses usually range from "It's too much code!" to "We don't have time for tests!", and this temptation to skip testing is especially prevalent when tests are simply an afterthought to development. This tendency to ignore testing (along with a few more reasons we'll discuss a little later) is why we advocate a *test first* approach to development. In a "test first" approach, you write your tests before you write your code, using your tests as a way to describe the behavior of the code you will write. For example, if you were writing an application to process XML files, you would want to write tests that describe how the application would behave when fed a valid XML file, when fed an invalid XML file, when the process is complete, and so on, and then you would go back and write the code to make those tests valid.

The "test first" methodology can follow one of two paths: plain *test-first* or *test-driven development*. We've described test-first development above: write your tests, then write your code to make the tests pass. Test-driven development is essentially the same idea, except that it adds an element of code refactoring to the mix. The basic process can be broken into five steps: add, fail, code, test, refactor (see a visual in Figure 2.1).

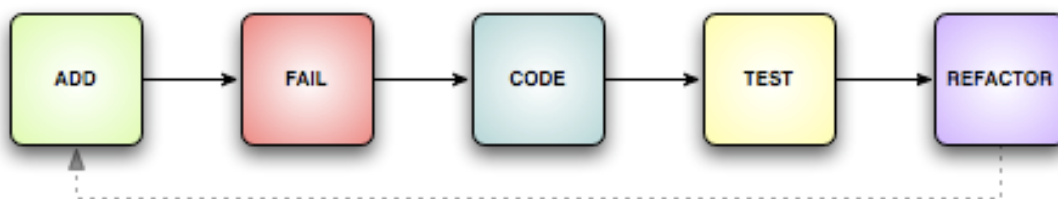


Figure 2.1 Test-driven development is a five step process: add, fail, code, test, refactor.

In the first step, you want to add tests for any new code you want to write *before you write the code*. You write a test that tests for the right behavior of the method; if it takes a string and returns its length, write a test that calls it with "hello" and test for a return value of 5; if it should raise an exception when you pass it a `nil`, test for that. As tests are added, you create a specification of the behavior that your code should conform to.

Once the specification is laid out, you move on to the fail step. This step is to simply run the tests and ensure they fail. And for a reason. We use tests to deal with errors we make in the code. But like all software developers, we can make errors while writing test cases. We can setup an object with all the right values, but forget to call the method we're supposed to test, or call a method but forget to check its result. Such a test will

always pass. Until you can make the test fail, at least once, you have no certainty it's actually testing your code.

Next, you simply write code to make the tests pass: you implement the specification you've created with tests. Write the method, module, or class; if the tests pass, great! If the test fails, then go back and debug your code to make it pass. When they all pass, go back refactor your tests and code so they're more effective (i.e., shorter but accomplish just as much or more). But why bother with all this business?

2.2 Test-Driven development with Ruby

The built-in Ruby testing library is very similar to the libraries of other languages such as Java and Python. The general architecture consists of test suites, built from methods, which makes assertions about your code. These test suites are run by test runners, which invoke the tests and report the results. See figure 2.2 for a visual.

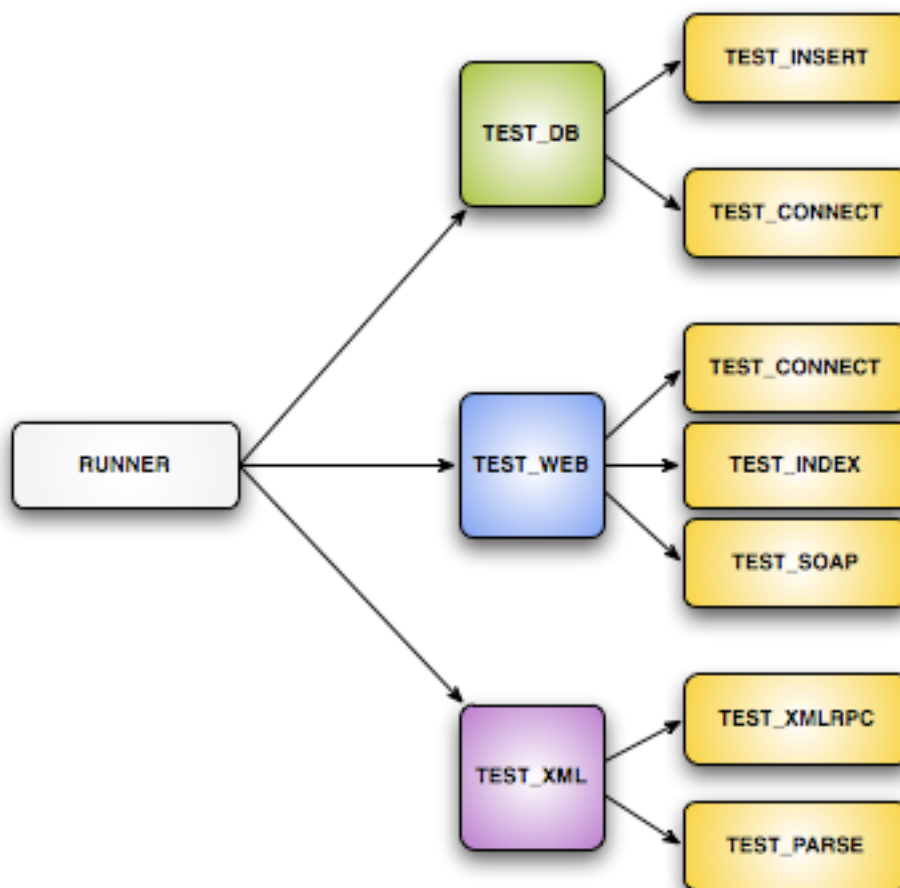


Figure 2.2 Test suites are composed of a collection of unit tests which are run one by one by a test runner.

So a single runner may invoke a number of suites, which may hold a whole lot of tests (e.g., the ActiveRecord tests for MySQL have 1,035 tests with 3940 assertions!). So, let's get going on actually writing some tests. We'll start with a very basic test: testing the length of a string. First, we'll need to create a test

harness, which in Ruby is a class that inherits from `Test::Unit::TestCase`; listing 2.1 shows our test case.

Listing 2.1: A simple test using Test::Unit

```
require 'test/unit'                                     #1
|
class MyFirstTest < Test::Unit::TestCase                 |
  def test_length                                       #2
    my_string = "Testing is fun!"
    assert_equal 15, my_string.length                  #3
  end
end

<#1 Require the Test::Unit library>
<#2 Test methods start with test_>
<#3 Assertions prove the code works, or fail the test>
```

First, we need to require the `Test::Unit` library (#1) and create a class that inherits from `Test::Unit::TestCase`; we can then create methods that test the code #2. Within these methods, we will make assertions (#3) about the code. Save this code to a file (e.g., `my_first_test.rb`) and run it. You should see output that looks something like this:

```
Loaded suite my_first_test
Started
.
Finished in 0.00058 seconds.

1 tests, 1 assertions, 0 failures, 0 errors
```

You can see that we had one test (i.e., one method that contained tests), one assertion, and no failures. In our example, we had one `assert_equal` statement that evaluated to true, so we had no failures. The `assert_equal` method isn't the only assertion available; there are a number of assert methods, which are listed in Table 2.1.

Table 2.1: Ruby's built-in testing library offers a large number of assertions baked right in.

Assertion	Description
<code>assert(boolean)</code>	Passes if <code>boolean</code> (i.e., a boolean expression) is true.
<code>assert_equal(expected, actual)</code>	Passes if <code>expected == actual</code> or <code>expected != actual</code> .
<code>assert_not_equal(expected, actual)</code>	
<code>assert_nil(object)</code>	Passes if <code>object == nil</code> or <code>object != nil</code> .
<code>assert_not_nil(object)</code>	
<code>assert_match(pattern, string)</code>	Passes if <code>string</code> matches or doesn't match <code>pattern</code> (i.e., a regular expression).
<code>assert_no_match(pattern, string)</code>	
<code>assert_raise(exception...) {block}</code>	Passes if the block raises or doesn't raise the provided Exception(s).
<code>assert_nothing_raised(exception...) {block}</code>	
<code>assert_same(expected, actual)</code>	Passes if <code>actual.equal?(expected)</code> or if <code>!actual.equal?(expected)</code> .
<code>assert_not_same(expected, actual)</code>	
<code>assert_respond_to(object, method)</code>	Passes if <code>object</code> can respond to the given method.
<code>assert_throws(expected_sym) {block}</code>	Passes if the block throws the provided symbol.
<code>assert_nothing_thrown {block}</code>	<code>assert_nothing_thrown</code> passes if the provided block doesn't throw any symbols.
<code>assert_instance_of(class, object)</code>	Passes if <code>object.class == class</code> .
<code>assert_operator(first, operator, second)</code>	Passes if the first object compared to the second object is true when using the provided operator.
<code>assert_kind_of(class, object)</code>	Passes if <code>object.kind_of?(class)</code>
<code>assert_in_delta(expected, actual, delta)</code>	Passes if <code>actual - expected <= delta</code>
<code>assert_send(send_array)</code>	Passes if method sent to object returns true.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

Using these assertions, you can build sets of tests that check your code effectively, but, like your code, tests don't run in a vacuum. Often times you need a little bit of setup before execution or tear-down afterwards, such as reading in files or creation of objects. **Test::Unit** provides a way to do that using the xUnit architecture. For example, if we wanted to test our string for length, emptiness, and its hash value, it wouldn't make sense to instantiate the string in each method, so we would use the **setup** method. Take a look at Listing 2.2 to see what we mean.

Listing 2.2: Using setup and teardown to prepare tests

```
require 'test/unit' #1
                    |
class MyFirstTest < Test::Unit::TestCase
  def setup          #2
    @my_string = "Testing is fun!"
  end              |

  def test_length
    assert_equal 15, @my_string.length #3
  end

  def test_emptiness
    assert_equal false, @my_string.empty?
  end

  def test_hash
    assert_equal "693492896", @my_string.hash.to_s
  end

  def teardown      #4
    @my_string = nil
  end
end

<#1 Require the Test::Unit library>
<#2 Setup the context in which these tests run>
<#3 Assertions prove the code works, or fail the test>
<#4 Teardown the context in which these tests run>
```

We start out the same way as last time: require the library and create a class that inherits from **TestCase** (#1). But this time, we do something a little different; we create a **setup** method to create an instance variable to hold our string (#2). Now we refer to **my_string** as instance variable throughout our tests (#3 and other places). Finally, we (superfluously) set **my_string** to **nil** in **teardown** #4. The **teardown** method is special like **setup**, except that it's run after tests rather than before. We know this example is a little contrived (especially since we didn't really do the whole testing cycle), but it's important to get the concept before we take it into practice. And now that you have the concept, let's move on to an example that's a little more real world.

Let's say you're building an application to grab some XML reports from a remote server, pull some data out, and then process the data you pull out. The reports, which are output from your payroll server, are simply catalogs of employees and the hours worked; they are used by the application you're building to build departmental expense reports. One of these reports might look something like this:

<payroll-report>

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```

<department name="Graphics">
  <employee name="Janice Partridge">
    <week id="1">40</week>
    <week id="2">38</week>
    <week id="3">30</week>
    <week id="4">40</week>
  </employee>
  <employee name="James Jones">
    <week id="1">33</week>
    <week id="2">23</week>
    <week id="3">30</week>
    <week id="4">25</week>
  </employee>
</department>
<department name="IT">
  <employee name="Andrea Lantz">
    <week id="1">40</week>
    <week id="2">41</week>
    <week id="3">45</week>
    <week id="4">39</week>
  </employee>
</department>
</payroll-report>

```

Fairly simple, right? Departments have employees which have weeks and hours. So let's look at the class that we've built to handle these reports. Now we need to write some tests to get our code started. Take a look at listing 2.3.

Listing 2.3 Tests for our to-be-implemented XML reporter

```

require 'test/unit'
require 'reporter'

class ReporterTest < Test::Unit::TestCase
  def setup
    @my_reporter = PayrollReporter.new('test.xml') #1
  end

  def test_department #2
    assert_equal nil, @my_reporter.department(1234)
    assert_equal nil, @my_reporter.department("Development")
    expected = {"Andrea Lantz"=>
      {"week"=> [{"id"=>"1", "content"=>"40"},
        {"id"=>"2", "content"=>"41"},
        {"id"=>"3", "content"=>"45"},
        {"id"=>"4", "content"=>"39"}]}}
    assert_equal expected, @my_reporter.department("IT")
  end

  def test_employee #3
    assert_equal nil, @my_reporter.employee(234323)
    assert_equal nil, @my_reporter.employee("Mr. Mustache")
    expected = {"week"=> [
      {"id"=>"1", "content"=>"40"},
      {"id"=>"2", "content"=>"41"},
      {"id"=>"3", "content"=>"45"},

```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```

        {"id"=>"4", "content"=>"39"}]}}
    assert_equal expected, @my_reporter.employee("Andrea Lantz")
end

def test_get_hours_for #4
    assert_equal nil, @my_reporter.get_hours_for("Miguel de Jesus")
    assert_equal 165, @my_reporter.get_hours_for("Andrea Lantz")
end
end

<#1 Create a PayrollReporter that tests can use>
<#2 Test department data was parsed from XML>
<#3 Test employee data was parsed from XML>

```

First, we create an instance of a class **PayrollReporter** #1 in the **setup** method; the constructor takes a path to an XML file (we'll feed it a test file here). Next, we create a test for a method named **department**, which will retrieve the details (e.g., employees) of a department #2; we follow this with a test for a method named **employee** which will basically do the same thing but for employees (i.e., it will retrieve a hash of their work hours rather than employees) #3. The last test we create is for a method named **get_hours_for**, which retrieves the total hours worked by an employee. Now we run the tests and get a load of errors. If we create a stub file (i.e., a file that has all classes and methods defined but with no method bodies), then we should see failures rather than errors. We've added some tests and they are properly failing, so let's implement the code (which you can see in listing 2.4).

Listing 2.4 Our PayrollReporter implemented

```

require 'rubygems'
require 'xml-simple' #1

class PayrollReporter
  def initialize(report_path = 'foo.xml') #2
    @report = XmlSimple.xml_in(report_path, { 'KeyAttr' => 'name' })
  end

  def department(name) #3
    if @report['department'].keys.include?(name)
      @report['department'][name]['employee']
    end
  end

  def employee(name) #4
    my_person = nil
    @report['department'].keys.each do |dept|
      if @report['department'][dept]['employee'].keys.include?(name.to_s)
        my_person = @report['department'][dept]['employee'][name.to_s]
      end
    end

    return my_person
  end

  def get_hours_for(name) #5
    my_employee = employee(name)
    return nil if my_employee == nil
  end
end

```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```

    total_hours = 0
    my_employee['week'].each do |week|
      total_hours += week['content'].to_i
    end

    total_hours
  end
end

<#1 Use the XmlSimple library>
<#2 Parse report from XML document>
<#3 Return the named department>
<#4 Return the named employee>
<#5 Calculate work hours for employee>

```

We'll talk more about using and manipulating XML data later on in the book, but for now we'll use a library named XmlSimple to do some basic data extraction #1. Now we implement each piece of the code one by one; first the constructor, which takes a file path and opens it for parsing #2. Then a **department** method to get the employees in a department #3, and an **employee** method to get the work hours of an employee #4. Lastly, a method **get_hours_for** to get the total work hours of an employee #5. Now we should run the tests to ensure they pass.

```

Loaded suite test_reporter
Started
...
Finished in 0.028279 seconds.

3 tests, 8 assertions, 0 failures, 0 errors

```

Excellent! Now that all of our tests pass, we should move on and refactor the current code and add new code (maybe a method to get an employee's work hours for a certain week or a whole department's work hours). As we add this code, we'll use the tests to ensure the applications behavior is consistent. If the language of assertions and tests doesn't appeal to you, though, there is an alternative testing method.

2.3 Behavior-Driven Development with RSpec

Test cases help you ensure the code behaves as it should, but what exactly should it behave like? We like to base our development on a written specification. The specification tells us what the code should do, the tests measure if the code behaves according to the specification, and the code itself implements the expected behavior. A written specification makes it easier to work in a team, to work against in-development libraries and services, to communicate to stakeholders what the software does, and to track progress as we get each feature implemented.

In this section we're going to talk about Behavior-Driven Development, a development approach that focuses on specifying, testing and implementing behavior – in this particular order, and we'll show you how to do that using RSpec.

2.3.1 What Is Behavior-Driven Development?

Specifications takes many forms, from high level architecture diagrams through story boards and screen mocks, and all the way down to capturing the behavior of each functional unit. The high level specification is most likely to remain accurate, it doesn't capture a lot of details, and wholesale changes to the architecture and

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

major components don't happen frequently. It's much harder to keep the functional specification synchronized with the code. If you ever had to develop against a specification written as a Word document you know what we're talking about. The specification starts out as the best idea we have about the software we're about to write, it's only during development that we find out it contains wrong assumptions, and fix those in code. In a matter of days the code starts to diverge, and the document turns into a work of fiction,

Behavior-driven development (or BDD for short) was developed by Dan North to address this issue. To do BDD right, we'll start by writing a specification of the code, but instead of creating yet another static document, we'll do that in the same repository that holds our source code. The specification forms a template that describes the expected behavior, which we then fill up with actual test cases. As we make changes, we'll reflect those back into the specification and tests, changing it alongside the code it specifies.

If all this sounds a bit conceptual, don't worry, in practice it's easy to do. Initially, a specification document may look something like this:

```
describe "A new string" do
  it "should have a length of zero."
  it "should be empty."
end
```

This is actually Ruby code, we'll explain what it does in just a moment. It's also a specification that's easy to read and understand, the behavior stands out so we don't have to guess it from the code. When we run it through a tool, we get a report that looks like Figure 2.3.

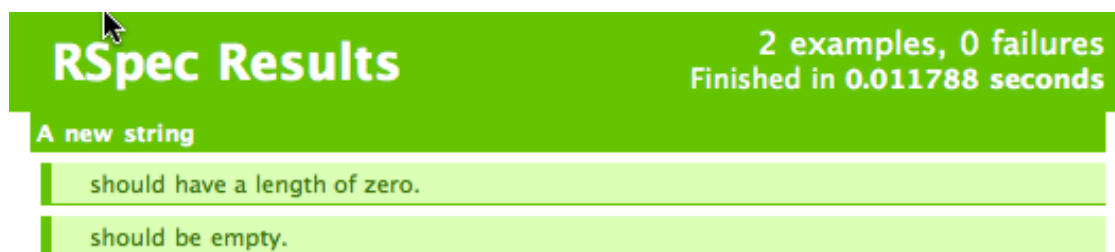


Figure 2.3: HTML report generated from our specification.

To go from specification to working code we'll start by adding test cases:

```
describe "A new string" do
  setup do
    @my_string = ""
  end

  it "should have a length of zero." do
    @my_string.length.should == 0
  end

  it "should be empty." do
    @my_string.should be_empty
  end
end
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

From here on we're following traditional TDD practices: write a test that fails, write the implementation to make the test pass. We'll encounter the difference between BDD and TDD again when we decide to make a change to our specification. Maybe we realized we need to use a different message format, present a task to the end-user in a different way, simplify a set of methods that are part of the public API. Whatever the reason, we're going to return to this file, change the specification to describe the new behavior, fill it up with a test case to check the new behavior, and work out the implementation to conform to both. Now we have a way to manage change, keeping the code and specification synchronized.

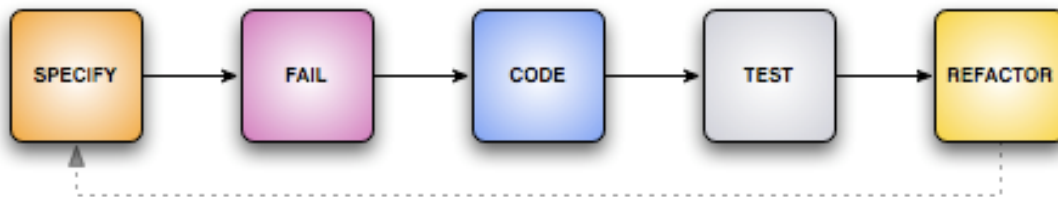


Figure 2.4:The behavior-driven development process starts with specification, adds tests and builds an implementation that matches the specification and passes the tests.

The report we generate has three colors: green for test cases that pass, identifying behaviors we implemented correctly, red for failing tests, and yellow for anything we specified but didn't get to implement yet. It tells us where we stand so we can track progress on our project.

So now that we covered the basics of BDD, let's look at a particular framework, one that's used extensively in the Ruby world and also influenced BDD frameworks for languages as diverse as Java and FORTRAN. We're talking, of course, about RSpec.

THE ROLE OF SPECIFICATION IN AGILE DEVELOPMENT

Agile development practices put emphasis on progress through small, incremental iterations, with a close feedback loop between developer and end-users. They're antithesis to BDUF (Big Design Up Front) and the waterfall model. What agile practitioners discovered is that having one stage to cement the specification, followed by a marathon to implement it, lead to disaster. No matter how well your intentions are, it's simply impossible to predict everything up front. Business needs change as you're building the software, technical difficulties require adjustments, and your customer may think they know what they want up until they see working code and realize they need something different. To create software successfully you need the ability to evolve its design as you're building it, working closely with end-users to make sure it meets their expectations.

To make agile practices work you still need a specification, what you don't need is a specification that's cast in stone before the first day of development. It has to be a living, changing document that can adapt to evolving requirements. Even better when it documents not what the software should have looked like, but what the delivered software actually does, and can be used to track progress. That makes BDD an important tool for agile development.

2.3.2 Testing with RSpec

So let's take a look at how to build a suite of BDD tests using RSpec as we look at a very simple spec for a class we'll create. We'll start by installing RSpec (`gem install rspec`) which provides us with the spec library we need to require in our code, and the `spec` command line tool we'll use to run these specification files. You can also use the command line tool to generate reports like the one we showed above, and to

repeatedly run only failing tests. In chapter 3 we get to talk about Rake and we'll show you how to setup your environment to automate these tasks.

Let's return to our string testing from before to get a grasp on BDD; listing 2.14 shows a basic spec for a new string object.

Listing 2.5 A context for an empty queue object

```
require 'rubygems'           #1
require 'spec'               |

describe "A new string" do   #2
  # Our specs will go here
end

<#1 Require RSpec>
<#2 Establish a new context>
```

First, we require the needed libraries #1. Now we have the methods that we need available to us, so we create a context (#2) for our string object: "A new string." This creates the context we'll need to describe the expected behavior of a new string. We can just list those behaviors without writing any code to test them (yet), or we can go straight ahead and start adding code that will test these behaviors. Since this is a trivial example, we'll move straight ahead and add some real test cases, which you can see in listing 2.15.

Listing 2.6 A few specs for a string object

```
describe "A new string" do
  before do                               #1
    @my_string = ""                      |
  end                                    |

  it "should have a length of zero." do   #2
    @my_string.length.should == 0        |
  end                                    |

  it "should be empty." do               #3
    @my_string.should be_empty           |
  end                                    |
end

<#1 Setup the context before running a test>
<#2 Test the length method>
<#3 Test the empty method>
```

The general setup of the contexts and specifications may look familiar to you since they are somewhat similar to those in the TDD library. First, we specify a **before** block (#1) which will create an instance variable for us to use in our specifications (cleaning up after a test is done using an **after** block). Next, we create a specification that a new string should have a length of zero #2. Note the use of the method **should**; the **should** and **should_not** methods are the primary verbs used in RSpec. These methods combined with various predicates (like **empty** above #3) allow you to specify various conditions. You use these predicates to describe how it should behave (**should throw_symbol**), exist (**should be_instance_of**), respond (**should respond_to**), and so on. Check out table 2.2 to see a list of the specification predicates that are available to you. You can find more details on the RSpec Web site (rspec.info).

Table 2.2 RSpec has a large number of specifications for you to use to verify the behavior of your application.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

Specification

```
should_be_predicate [args]
should_not_be_predicate [args]

should_be_close(val, tolerance)
should_not_be_close(val, tolerance)
should_have_something [args]

should <operator> value
should_not <operator> value
should_include(item)
should_not_include(item)
should_match(regex)
should_not_match(regex)
should_be_an_instance_of(class)
should_not_be_an_instance_of(class)
should_be_a_kind_of(class)
should_not_be_a_kind_of(class)
should_respond_to(symbol)
should_not_respond_to(symbol)
should_raise_error([ex], [mesg])
should_not_raise_error([ex], [mesg])

should_throw_symbol [:symbol]
should_not_throw_symbol [:symbol]
should_change(receiver, :method)
should_change { block }

should_have(number).things
should_not_have(number).things

should_have_at_least(number).things
should_have_at_most(number).things
should_eql(arg)
should_not_eql(arg)
should_equal(arg)
should_not_equal(arg)
should_satisfy {|arg| block }
should_not_satisfy {|arg| block }
```

Description

Uses the question mark form of the method provided as predicate (e.g., `simple_object.should_be_empty?` calls `empty?` on the `simple_object`).

Allows a value tolerance for floating point specifications.

Calls `has_something?` on the receiver (e.g., `should_have_key` calls `has_key?`).

Uses Ruby's operators to test (e.g., `should_be < 3` or `should_be =~ /hi/`).

Specifies that a collection should not include `item`.

Specifies that the receiver should match the provided regular expression.

Uses `instance_of?` to test the receiver's class.

Uses `kind_of?` to test the receiver's class ancestry.

Test whether the receiver responds to the given symbol using `respond_to?`.

When called on a `Proc`, specifies that a given exception should not be raised; if no exception is given, checks for any exception thrown. The parameter for `mesg` is a string or regular expression that is matched against the exception's message.

When called on a `Proc`, tests whether a given symbol was thrown; if no symbol is given, it tests for any symbol.

When called on a `Proc`, this tests that the value of `receiver.method` has changed after the `Proc` has executed; this method has a number of forms (check the latest documentation to see them all).

Tests the receiver's count (`should_have`), lower limit (`should_have_at_least`), and upper limit (`should_have_at_most`) of the collection named for `things` (e.g., `my_array.should_have(3).items`).

Specifies that the receiver should be equal to `arg` in object identity (`eql`) or merely value (`equal`).

Tests whether the block evaluates to true when the receiver is passed as the only argument.

Back to our example, run the specs, and if they pass (which they should!), let's move on looking at BDD with something a little more realistic. Let's say you want to test an `IntranetReader` object, which is a wrapper for your intranet's RSS feed; let's create a spec for all the behaviors of the library. Remember, since we're using behavior-driven development, we want to start with a spec and need to specify behavior whether or not the object has stories in it, whether it properly connected, whether the RSS is parsed correctly, and so on rather than asserting values and conditions. Let's start with the most basic spec, which you can see in listing 2.7.

Listing 2.7 A basic spec for our `IntranetReader` class

```
require 'rubygems' #1
require 'spec'      |
require 'intranet_reader' |

describe "A new IntranetReader" do #2
  before do
    |
    @my_rss = IntranetReader.new |
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```

end

it "should be empty." do
  @my_rss.should be_empty
end

it "should not have 'raw'." do
  @my_rss.raw.should be_nil
end
end

specify "A populated IntranetReader" do
  before do
    @my_rss = IntranetReader.new
    @my_rss.process
  end

  it "should have entries." do
    @my_rss.entries.should_not be_nil
  end

  it "should have 'raw'." do
    @my_rss.raw.should_not be_nil
  end
end

<#1 Require RSpec>
<#2 Specify behavior of an empty IntranetReader>
<#2 Specify behavior of a populated IntranetReader>

```

This is very similar to our specification from before, except this time we've specified two contexts. First, we require our essential libraries (#1), then proceed to create a context for an empty **IntranetReader** #2. Inside the context we have specifications to check for the emptiness of the new object we've created. The second context holds specifications for a populated **IntranetReader** #3. These specifications ensure the object was populated properly. If you run these specifications now (using the spec command like: **spec intranet_reader_spec.rb**), you will get some errors (especially if you don't create an intranet_reader.rb file to require); it should look something like the following when they're failing "correctly."

```
FFFF
```

```

.
.
.

```

```
Finished in 0.002936 seconds
```

```
4 specifications, 4 failures
```

Once you've reached this point, all you have to do is go back and implement code to make the tests pass, just like test-driven development. Listing 2.8 shows our implementation using the SimpleRSS gem, but yours may look somewhat different.

Listing 2.8 Our implementation of IntranetReader

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```

require 'rubygems'
require 'simple-rss'
require 'open-uri'

class IntranetReader
  def initialize(url='http://yourintra.net/feed/')
    @feed_url = url
  end

  def process
    @raw = open(@feed_url).read
    @rss = SimpleRSS.parse @raw
  end

  def entries
    @rss.items
  end

  def empty?
    @rss.nil? ? true : @rss.items.empty?
  end

  def raw
    @raw
  end
end

```

If you run the specifications and there were no failures, now it's time to add and refactor tests and/or code. Perhaps you'd like to implement a method to validate the RSS input before it's processed, or maybe you'd like to allow the user to search the RSS feed. Whatever you add, just remember: specify, code, test, refactor. But what if your testing environment doesn't have access to the RSS feed? How do you mimic your production environment for testing? The next section will discuss a few strategies to deal with this problem and a few techniques to setup an environment for your tests.

2.4 A Testing Environment

As you read this chapter and start playing with tests, you may be thinking, "What if my test environment doesn't have access to certain parts of my application like networked components or third party services?" Fortunately for us, the pioneers of software testing devised a number of techniques to handle this sort of thing. In this section, we'll take a look at three of these techniques: fixtures, stubs, and mocks.

2.4.1 Setting up a baseline with fixture data

One of the earliest difficulties you'll encounter with testing is replicating the environment in which your code will run. Many times it's not so much replicating the relationship among your code, but the external environment from the operating system or network. Many developers use *fixtures* as a first "crutch" to get this sort of environment up and going. Fixture data (or fixtures) creates a fixed, baseline environment for your tests to run in. For example, like Ruby on Rails, you might use a YAML file to setup the database with an initial set of products so you can test your inventory management code against known data. Or, you might create a few data files that are pre-populated with data that would normally be generated by another application so you can test how your code reacts to existing data.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

RAILS AND FIXTURES

In the Ruby development realm, you will probably most often hear the term “fixture” applied to Rails and its testing fixtures for databases. Don’t be confused: the concept of a fixture existed before Rails was around and is useful outside of Rails.

There is no special technique necessary to “use” fixtures, simply because you are constructing a baseline environment for testing. For example, let’s say you were testing a component of your application that received the output from `ps` to check for the existence of a process. See listing 2.9 for the original class and test.

Listing 2.9 A process checker class and test

```
require 'test/unit'

class ProcessChecker
  def initialize
    @ps_output = `ps -A` #1
  end

  def rails_server?
    @ps_output =~ /ruby script\/server/ #2
  end

  # Other process checking methods...
end

class TestProcessChecker < Test::Unit::TestCase
  def test_initialize
    my_checker = ProcessChecker.new

    assert my_checker.instance_variables.include?('@ps_output') #3
  end
end
<#1 Get the output from ps>
<#2 Check for the Rails server script>
<#3 Assert the output is in an instance variable>
```

As you can see, we have a `ProcessChecker` class that grabs the output from `ps`, stores it in an instance variable #1, and then checks that variable for various processes (in this case, we just check for the Rails `server` script to be running) #2. We then create a test that ensures that variable is actually set when the object is instantiated (#3), but now we’re at a testing impasse. How do you test the results of the `ps` call? Will you have to start the Rails `server` script every time this test is run? What if the deployment environment doesn’t necessarily even have Rails installed?

This is one situation where implementing a fixture makes sense. If you create some standard data for the object to use instead of a call being made to `ps` every time, then you can ensure your tests will run the same everywhere (even Windows!). Take a look at a revised version of the test in Listing 2.10.

Listing 2.10 A revised test with fixture data

```
class TestProcessChecker < Test::Unit::TestCase
  def test_initialize
    my_checker = ProcessChecker.new

    output = <<PS #1
PID  TT  STAT      TIME COMMAND |
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```

1  ?? S<s    0:06.84 /sbin/launchd
21 ?? Ss     0:13.90 /sbin/dynamic_pager
25 ?? Ss     0:05.37 kextd
35 p1 S+     0:06.21 ruby script/server
PS

```

```

my_checker.instance_variable_set('@ps_output', output) #2

assert my_checker.instance_variables.include?('@ps_output')
end
end
<#1 Creating the fixture data>
<#2 Setting the data in the object>

```

In this instance, we fill in some fixture data, which has static output from a run of the `ps` command #1. This data is then fed into the object via `instance_variable_set` (yes, we are injecting the data in a way that makes our test case fragile, we recommend adding methods to your classes and modules that will assist in testing, but we wanted to show what's possible when you don't have full control of the objects you're testing). Now we can run this test on any platform (even though Windows will complain it can't find `ps`) and it should behave the same since we now have a set of fixture data to work from. You could then go back to your tests to refactor and add to them like listing 2.11.

Listing 2.11 Our refactored and expanded tests

```

class TestProcessChecker < Test::Unit::TestCase
  def setup #1
    @my_checker = ProcessChecker.new

    output = <<PS
PID  TT  STAT      TIME COMMAND
  1  ?? S<s    0:06.84 /sbin/launchd
 21  ?? Ss     0:13.90 /sbin/dynamic_pager -F /private/var/vm/swapfile
 25  ?? Ss     0:05.37 kextd
 35  p1 S+     0:06.21 ruby script/server
PS
    @my_checker.instance_variable_set('@ps_output', output)
  end

  def test_initialized
    assert @my_checker.instance_variables.include?('@ps_output')
  end

  def test_rails_server
    assert @my_checker.rails_server? #2
  end
end
<#1 Using the setup method to create an object to test>
<#2 A test for the check method>

```

Now that we've refactored to use the setup method (#1), it's trivial to add tests for the methods on the class (#2). Fixtures make it easy to set a baseline environment for your tests, but they can't cover everything. For example, if you're using a database, often times fixtures do not scale easily and become a hassle to maintain (you can find plenty of blog posts about the subject on Google). If you plan on running your software

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

on multiple platforms, then you may have to maintain a fixture for each platform in cases like the above example. There are alternative methods to get around these problems, though, and with a little work and abstraction you can take advantage of them.

2.4.2 Faking components with stubs

Stubs are essentially “fake” objects or methods that mimic the behavior of “real” objects. These stubbed objects will create a façade of the real objects, seemingly behaving like the real object, when in actuality it fakes the real object’s logic. This technique is useful for mimicking parts of your application that may not be available or are very performance intensive. For example, let’s say you have an application that orders products from one of your suppliers, but you don’t want to order a room full of products while running your tests. In that case, you would create stubbed objects to fake the supplier service so your tests can still be run without actually buying anything. These objects would still act like, expose the same API as, and return the same values as the real objects which interacted with the remote service, except they wouldn’t actually be interacting with the remote service. For a visual of the concept, see figure 2.4.

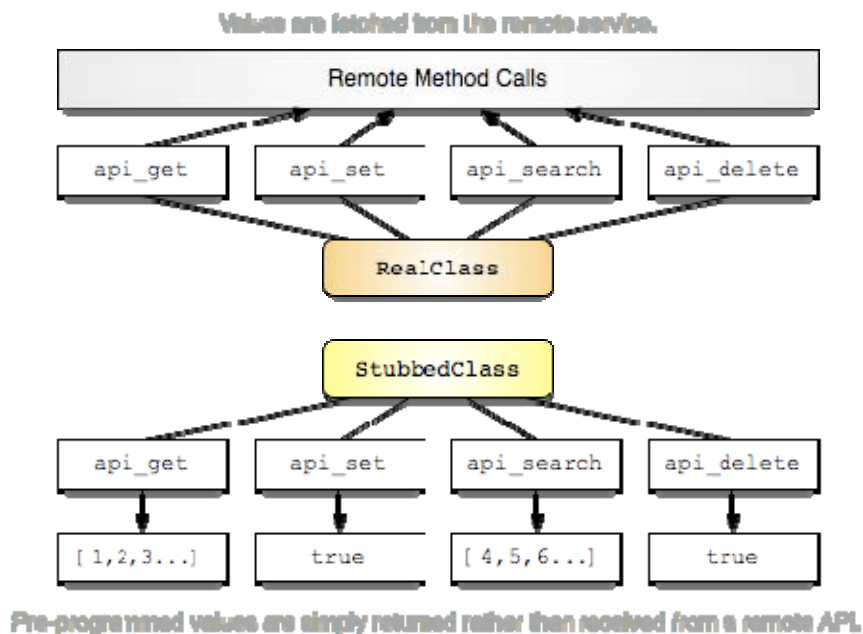


Figure 2.4 A stubbed class will simply seem to act like the real object rather than actually behave like it. In this case, it doesn’t actually grab data from a remote service, but to the code consuming the API, it appears to.

Let’s look at this example in code. Let’s say the supplier exposes a web service API to you and has included a very natural Ruby wrapper for this service. It exposes a **search** method for finding products, an **add_to_cart** method for adding products to your cart with them, and a **purchase** method for finishing your order. Your application receives the purchasing data in XML form, which is parsed then given to a class unique to each supplier for purchasing. Each supplier class has an **execute_purchase** method, which takes an array of product names and quantities that are taken from the purchasing data XML. Our test for a simple purchase looks like listing 2.13.

Listing 2.13 Building tests for our supplier class

```

class TestSupplierInterface
  def test_execute_purchase
    data = [
      { :item_name => 'Red Rug', :quantity => 2 },
      { :item_name => 'Set of Pens', :quantity => 17 }
    ]
  end
end

```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```

    my_supplier = MaterialSupplier.new
    assert my_supplier.execute_purchase data
  end
end

```

As you can see, we simply instantiate our class, call the `execute_purchase` method, and assert it returns `true`. If you run the tests, it should fail, so we're ready to implement the class. The class implementation for this particular supplier looks like listing 2.14.

Listing 2.14 A class for handling purchasing from a supplier

```

class MaterialSupplier < Test::Unit::TestCase
  def execute_purchase(purchase_data)                                #1
    purchase_data.each do |purchase|                                #2
      product_id = SupplierInterface.search(purchase[:item_name]) |
      SupplierInterface.add_to_cart(product_id, purchase[:quantity]) |
    end                                                              |

    SupplierInterface.purchase                                     #3
  end
end
<#1 The method takes purchasing data>
<#2 Iterate each data>
<#3 Finalize the purchase>

```

As we mentioned before, the `execute_purchase` method takes purchasing data from another part of your application. The data is then iterated, where the product name is searched, its `id` found, and then it is added to the cart. After the cart has been populated, we finalized the purchase. Now that we have a test and a class, how do we reliably test this? We can use the remote system to implement it, but we shouldn't depend on that all the time, especially if we want our tests to be consistent and reliable in every environment. Even if it is available, you probably don't want to (or shouldn't) send real requests to the remote server. So, we need to stub it.

To stub the `SupplierInterface` class, we'll simply need to create a class of the same name and input methods that return the right results without actually making any requests to the remote service. See listing 2.15.

Listing 2.15 A stubbed SupplierInterface

```

class SupplierInterface
  @@products = {                                                    #1
    1234 => 'Red Rug',
    5678 => 'Ergonomic Chair',
    9012 => 'Set of Pens'
  }
  @@cart = {}                                                       |

  def self.search(product_name)                                     #2
    @@products.index(product_name)
  end                                                                |

  def self.add_to_cart(id, quantity)                                #3
    @@cart[id] = quantity
  end                                                                |
end

```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```

def self.purchase                                     #4
  true if @@cart.length > 0                           |
end                                                    |
end
<#1 Our initial data>
<#2 Search for products by name>
<#3 Add to the @cart>
<#4 Finish the purchase>

```

We've create a stubbed version of **SupplierInterface** that works from data that we've statically created (#1). We use class level hashes to implement searching (using the **index** method) #2 and adding to a cart #3. Since all we are testing at this point is the fact that we add items to the cart and purchase them, we'll just have the **purchase** method mimic the remote system and return **true** if the order was successful (i.e., there are items in the cart) #4. Now our tests should execute in the same manner, except using our local stubbed class instead of the remote service interface.

Stubbing is a really useful technique for getting your tests up and running, but at the end of the day, stubs are still dumb objects. They simply pass back the value you ask for, but they don't actually verify that the right method calls are being made to them. This deficiency can make your tests weaker than they would if you were to test those call to the objects you're stubbing since you could be missing out on a missed method call or a wrong parameter. Now let's take a look at another similar technique: mocks.

2.4.3 Setting behavior expectations with mock objects

Mock objects are very similar to stubs, except that they set expectations about your code's interactions with them. For example, in our previous example, a mock object wouldn't have just stupidly taken the input for the search or add_to_cart methods, but it would have verified that it received the right parameters or that the method was even called in the first place. In doing this, it not only tests your code's actions on the returned values, but it also tests your code's interactions with these objects: parameters passed, methods called, exceptions raised, and so on.

In this section we'll take a look at a couple of ways to create mocks objects. The first method we'll discuss is creating your own mock object patterns, no library required. We'll then take a look at using libraries such as Mocha to create mock objects.

Creating custom mocks

Before there were fancy mocking libraries, developers had to get their hands dirty and build their own mock objects. Of course, unlike programming on punch cards, creating your own custom mock objects is still a valid practice. Sometimes you there is a very specific order or logic to how your application runs, and trying to shoehorn that into some mocking libraries' interfaces can sometimes be awkward. Building your own mock objects is just easier.

Let's take our example from before. If we take our stubbed class and add in expectations for each method, we can make it into a mock object. See listing 2.16 for a revised version of the **SupplierInterface** stubbed class we created earlier.

Listing 2.16 Tests for a shipping components

```

class SupplierInterface
  @@products = {
    1234 => 'Red Rug',
    5678 => 'Ergonomic Chair',
    9012 => 'Set of Pens'
  }

```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```

@@cart = {}

def self.search(product_name)
  $expectations[:search] = true if product_name.is_a?(String)      #1
  @@products.index(product_name)
end

def self.add_to_cart(id, quantity)
  $expectations[:add_to_cart] = true if id.is_a?(Integer) &&      #2
  quantity.is_a?(Integer)
  @@cart[id] = quantity
end

def self.purchase
  $expectations[:purchase] = true                                  #3
  true if @@cart.length > 0
end
end
<#1 We expect a string argument>
<#2 We expect two integers>
<#3 We expect this method to be called>

```

Here we created have expectations about parameters (#1 and #2) and the method (#3) placed into a global variable (not the best practice, but for the sake of brevity, we'll keep it simple). These expectations can then be evaluated in an updated test, as seen in listing 2.17.

Listing 2.17 An updated test to use our custom mock

```

class TestSupplierInterface < Test::Unit::TestCase
  def setup
    $expectations = {}
  end

  def test_execute_purchase
    data = [
      {:item_name => 'Red Rug', :quantity => 2},
      {:item_name => 'Set of Pens', :quantity => 17}
    ]

    my_supplier = MaterialSupplier.new
    assert my_supplier.execute_purchase data

    [:search, :add_to_cart, :purchase].each do |meth|
      assert $expectations[meth], "#{meth} was not called or not called
properly."
    end
  end
end

```

As you can see, we added a small piece of code to the end of the method to iterate over our known expectations (one for each method we expected to be called) to make they were set. If they weren't, the test fails. Using this techniques, our tests are more robust, and if used consistently, can make full test coverage a reality a lot quicker than using simple stubs or nothing at all. The only problem with custom mocks is they aren't formal and as a result, end up being pretty time consuming to maintain. Fortunately there are a few libraries to formalize your mocks and speed up the development of your tests.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

Creating mocks with Mocha

As mocking has become a more popular practice, a number of libraries have been created which makes the process much easier. In this section, we're going to discuss Mocha, one of the newer mocking libraries. Its mocking interface is much more intuitive and powerful than that of some of the other mocking libraries, so it is the one we would advocate to you were you to ask (and you are asking, since you are reading this book!).

OTHER LIBRARIES

If in reading this section you find that Mocha doesn't really work with your testing practices or is "too magical," then you may want to look into FlexMock, Jim Weirich's mocking library, or RSpec's mocking capabilities.

Mocha works via a mechanism it calls expectations (hence why we called our hash in the earlier examples **\$expectations**). Mocha sets these expectations on object behavior; so, for example, you might set an expectation that an object will have its **process** method called or that an object's **save** method will be called and it should return **true**. This is just like what we did before, except that Mocha makes the process much cleaner. Mocha expectations are defined by a single method, **expects**, and can be attached to any object. When an expectation is attached to an object, a stub is automatically created and an expectation setup. For example, Listing 2.18 is our previous example but refactored to use Mocha.

Listing 2.18 The previous mocking example, rewritten using Mocha

```
class SupplierInterface #1
end

class TestSupplierInterface < Test::Unit::TestCase
  def test_execute_purchase
    data = [
      { :item_name => 'Red Rug', :quantity => 2 },
      { :item_name => 'Set of Pens', :quantity => 17 }
    ]

    my_supplier = MaterialSupplier.new

    SupplierInterface.expects(:search).with('Red Rug').returns(1234) #2
    SupplierInterface.expects(:add_to_cart).with(1234, 2) |
    |
    SupplierInterface.expects(:search).with('Set of Pens'). |
    returns(9012) |
    SupplierInterface.expects(:add_to_cart).with(9012, 17) |

    SupplierInterface.expects(:purchase).returns(true) #3

    assert my_supplier.execute_purchase data
  end
end
<#1 An empty class>
<#2 Setting up expectations for methods>
<#3 The expectation that the purchase will complete>
```

In this version of the code, we still verify behavior, but we don't have to spend the time to implement an entire class and we get the benefit of being able to verify specific calls to a method. In our example, we create a blank class since all we're doing is attaching stubs to a dummy class #1; in a "real world" setting, this would be unnecessary because there would be a real class to attach the stubs to. Next, we define expectations for each

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

method and calls to that method #2. The final expectation is set and the method tested #3; when this test suite is executed, you should see something like the following:

```
Loaded suite purchasing
Started
.
Finished in 0.001637 seconds.

1 tests, 6 assertions, 0 failures, 0 errors
```

As you can see, assertions are created for each expectation that we set; these assertions verify the behavior of your code when calling the stubbed methods.

MOCKS OR STUBS?

The answer is: Yes. If you're not using a mocking library, often times it's more effort than it's worth to make full on mocks, especially if you're already testing the code you're mocking because you wrote it. Even further, if the code or service you're faking doesn't return any values that are used in your code, stubbing just makes more sense.

Ruby offers a number of utilities to take all sorts of measurements of your testing code. Now that we've taken a look at testing, specifying, stubbing, and mocking, we need to look at a few of these techniques that test your tests.

2.5 Testing your tests

No one is perfect; if testing helps to smoke out your imperfections, then who's to say you won't create some in the process of writing test code? In other words, "Who's testing the testers?" In this section, we'll look two metrics that you can take of your testing code: coverage and quality.

2.5.1 Testing code coverage

As your applications grow in size, so you shall your test suites, and as your test suites grow in size, so shall the possibility of not testing a method or class here or there. If you miss writing tests or specs for some code, that's alright, because Ruby offers a few tools to help you ensure your code is properly covered.

ZENTEST

The ZenTest tool from Ryan Davis is another way to avoid skipping tests. If you run it over your code, it will generate stubs for code that you've written tests for or generate tests for code that doesn't have it. It's excellent if you're writing tests for the existing code or if you're doing TDD and want to save some effort.

Typically, there isn't an issue with skipping a test here or there unless it's a very important method which you forgot to test or a class that is used quite often. Fortunately, Ruby offers a couple of tools that can help you ensure your code is covered with good quality tests 100%, and we'll get to talk about them next

Testing coverage with rcov

The rcov utility from Mauricio Fernandez allows you to test the C0 (i.e., only the top level items like methods, not any optional branches of execution) coverage of your code. It can be installed using RubyGems (e.g., `gem install rcov`) and executed using the `rcov` command. When executed with a test file as a parameter (e.g., `rcov test_db.rb`), it will crawl the code and look at the test coverage for each class. Then, rcov collects this data and produces HTML/XHTML (or, optionally, text).

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

TEXT RESULTS

Rcov can also show your results in a simple text format if you feed it the `-T` option on the command line.

Rcov works by running over test suites and checking the line coverage of the code. You run it by invoking the `rcov` command followed by the list of files to test. As an example, we'll run `rcov` on a portion of Rails using the command `rcov test/*.rb -o coverage/` (the `-o` option allows you to specify an output directory). Figure 2.5 shows the generated HTML page.

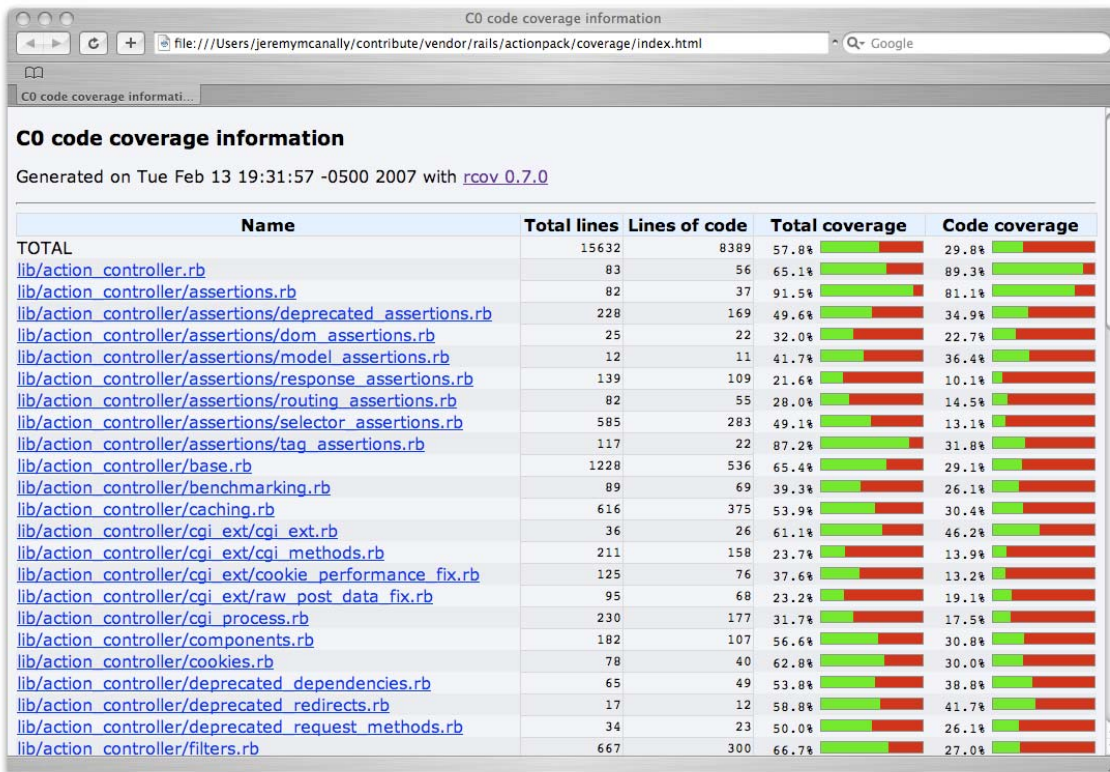


Figure 2.5 The `rcov` tool presents its results as HTML or text; the HTML view has nice graphs which illustrate code coverage and individual pages for each file tested.

As you can see, for each code file, there is a percentage given with a small bar graph. Of course, running this on a smaller codebase will yield less spectacular results, but it still a very useful tool for making sure your code is completely covered. Once you've mastered code coverage, it may be beneficial to ensure your tests are of good quality and coverage; Heckle is a tool to test just that.

2.5.2 Testing quality with Heckle

Heckle from Ryan Davis is a different kind of tool. Think of Heckle as your tests' worst nightmare. It tests the *quality* of your tests using a technique called fuzz testing. The problem with test cases is that, merely passing doesn't guarantee you're actually putting the code to the test. Perhaps you're calling one method, but not testing for the changes it makes to the object. These kinds of errors happen on occasion, and you'll notice them when you change the test code slightly, and the test still passes. But who does that?

Heckle takes your tests, messes with the test code, and then runs them again to ensure that they fail when they should. If your tests don't notice crazy changes to your code, then either that code isn't covered or isn't covered very well.

BDD AND HECKLE

Heckle supports RSpec also, all you have to do is invoke RSpec with the `--heckle` option.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

Think of Heckle as a way of “testing your tests,” checking them for coverage and quality. For example, let’s say you have the class in listing 2.17.

Listing 2.17 A simple class to Heckle

```
class Customer
  def initialize(name = nil)
    @name = name
  end

  def tag
    tag = "Customer: "
    tag += @name.nil? ? "<unknown>" : @name
    tag
  end
end
```

This class will create a pretty simple object to represent a customer and hold their name. Let’s look at a test for this class in listing 2.18.

Listing 2.18 Our tests to Heckle

```
require "test/unit"
require 'customer'

class TestCustomer < Test::Unit::TestCase
  def test_tag
    @customer = Customer.new('Mike Stevens')
    assert_equal 'Customer: Mike Stevens', @customer.tag
  end
end
```

Simply for an example’s sake, we have one test that tests the value returned by the tag method. If we were to run rcov on this, it will tell you that we have 100% coverage, but if we run Heckle, we see a little different story.

Initial tests pass. Let's rumble.

```
*****
*** Customer#tag loaded with 3 possible mutations
*****

3 mutations remaining...
2 mutations remaining...
1 mutations remaining...
```

The following mutations didn't cause test failures:

```
def tag
  tag = "Customer: "
  tag = (tag + if @name.nil? then
    "\037B\0e|H\020B\027\022W3_q\027\025G\0f?bZMJ&p/P&\nP\016\036-#\031"
  else
    @name
  end)
end)
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```
tag
end
```

One of the mutations that Heckle ran didn't cause a test failure when it should have. Note that it changed the value that is returned from tag from nil to a long string of text. Since we didn't test for this, it didn't fail. So, now we need to go back and add or refactor tests to catch the mutation; look at listing 2.19 to see how we did it.

Listing 2.19 Our updated tests that can stand up to even a strong Heckling!

```
class TestCustomer < Test::Unit::TestCase
  def test_tag
    @customer = Customer.new('James Litton')
    assert_equal 'Customer: James Litton', @customer.tag
  end

  def test_tag_empty
    customer = Customer.new
    assert_equal 'Customer: <unknown>', customer.tag
  end
end
```

We added a test to test for an empty name attribute, and if we run Heckle again, we should see a different result.

Initial tests pass. Let's rumble.

```
*****
*** Customer#tag loaded with 3 possible mutations
*****

3 mutations remaining...
2 mutations remaining...
1 mutations remaining...
No mutants survived. Cool!
```

No mutants survived! Great! So, now we've fixed our tests to stand up even to fuzz testing. Now it's time to go back and more tests, more code, and heckle again and again. Even though Ruby makes this cycle easier, it can get very annoying to make a change, run the tests, make a change, run the tests, and so on ad nauseum.

TESTING TOOLS

This chapter covers a number of tools to help you with testing, but Appendix C covers a few more peripheral tools that didn't make it in here.

2.6 Summary

Ruby's testing facilities almost make it too hard not to test! In this chapter, you've been exposed to test-driven development and how to do it with Ruby. You should be able to build test suites for new and old code, and expand those tests over time. You were also shown how to do behavior-driven development using the RSpec library, starting with the specification and working your way to implementation.

We also discussed how to mimic a production environment while testing using techniques like fixtures and fake objects. You learned how to build stubs and mocks (both from scratch and using Mocha) in order to

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

mimic code functionality. In the last stretch, we looked a couple of secondary tools that can help you out when writing tests. The rcov tool will ensure your code is covered, while the ZenTest tool will help you ensure you code is covered well.

In the next chapter, we'll look at using Ruby with other technologies. The rest of this book is about practical techniques, many of them using other technologies, but before we dig into that content, we need to take a look at the basics of integration and automation with Ruby.

3 Integration with Ruby

What's the first thing you would use Ruby for? For many developers we know that would be integration, from the simple automation tasks to building new applications by integrating existing ones. Before we move into the rest of the book where we are talking about specific libraries or Ruby-related technologies, we need to take some to look at using Ruby to integrate existing technologies with other existing technologies or new Ruby code since many of you will be using the information in these chapters to do integration. There are a lot of places you can use Ruby today, adding new features and composing applications, without having to write existing code, but why use Ruby and not other languages like Java or PHP? The answer is part language features, part toolkit, but for the full the full answer, let's take a closer look.

3.1 Glue languages

It's common knowledge that the universe is held together with glue languages and duck tape.

What exactly is a "glue language"? There are fully fledged applications that do a lot of interesting things, like balancing accounts, managing inventory, processing orders. They tend to rank high on the level of importance, in part because they're more expensive to produce. And then there are simple programs that solve smaller problems. They don't get as much street cred, but they do have a special place in the world: they keep large systems from falling apart. Behind every big system you'll find tens, sometimes thousands of these glue programs. They backup the database and rotate log files, they synchronize between multiple systems, they send e-mail alerts when things break, they patch little holes that would otherwise leak like a sieve.

Dynamic interpreted languages are easy to work with: it only takes a few lines of code to accomplish simple tasks. You write the code and run it, without the overhead of setting up projects, managing dependencies, compiling and linking, deployment procedures. So armed with a dynamic language, it's easy to write these simple scripts. To distinguish them from full blown applications, these little programs are called "scripts" or "glue scripts," and languages that excel at writing these programs are known as *scripting languages* and *glue languages*. Well known glue or script languages include Perl, Python, Ruby, LUA, JavaScript and VBA.

There's no clear, "official" distinction between glue, scripting and general purpose programming languages. Some languages are only useful as scripting languages, for example VBA in the Microsoft Office world and Tcl in the realm of UNIX. Some languages work best for gluing programs together, in particular command line language like Bash and PowerShell, Expect and AppleScript.

At the other end of the programming language spectrum are languages that are an overkill for writing simple programs. You don't see a lot of glue code written in Java or C++. It's possible, but it takes too much energy to do simple tasks. The simple things are hard to do: that's kind of ironic, don't you think? But that's the nature of low-level programming languages; they can scale up but they don't scale down. And then, there are programming languages you can use to write complete applications in all their complexity, but also to quickly scribble out a glue script or two. We're talking of course about Ruby, but also Perl, Python, Tcl and others.

Throughout this book we'll talk about Ruby as a general purpose programming language for building fully fledged applications, but in this chapter, we want to cover a different side of Ruby: using Ruby to script applications and glue programs together.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

3.1.1 Ruby as scripting language

Generally speaking, a scripting language is embedded inside a larger application or system, and is a different language from the one used to write the hosting application or system. We use different languages to solve different problems. For example, most Web browsers are written in C++ since rendering HTML requires a lot of processing power, but use JavaScript for scripting dynamic behavior in Web pages. JavaScript is much easier to develop with than C++ or Java, that's why you see more JavaScript out there than ActiveX or Java Applets. Similarly, Microsoft Office is written in C++ for performance and access to low-level Windows APIs, but most extensions are scripted using VBA. Scripting languages exist to make these little scripts' development as painless as possible.

If you're building a large scale application and you want to make it easy to develop new features and change its functionality, you may consider using a scripting language. If you're working with Java, you can use the scripting support in Java 1.6 and above in combination with JRuby, an implementation of Ruby that runs in the Java Virtual Machine. You'll be able to use Ruby as a scripting language for your Java application. Why? If you're delivering an application that your customers want to customize, Ruby will make their life easier, and breathe new life into the application by adding scripting or plugin capabilities in an easy to learn and readable programming language. The Web-tier and business logic are specifically ripe for scripting languages. JSP is already script-like, and you can simplify it further by using a simpler language. Business moves fast, and with it, your business logic. As such, your business logic is the one place you'll want to make changes often, by making it easier to change using a simpler, more nimble scripting language like Ruby, you can let your application keep up with your business, easier and faster than you could with something with a longer development cycle.

As an example, let's look at an implementation using a project that Jeremy worked on as a prototype. Let's say you are building an accounting application in Java that has a lot of rules that need to change often: budgets fluctuate so resource requests need to be automatically moderated differently at different times, departments were allowed to make different requests every quarter (sometimes they were given "perk requests" if they performed well), and the size of the department governed much of the accounting business logic. Would it be easier to define the rules in Java?

```
public bool PhysicalResourceRequest(Resource res)
{
    bool ret = null;

    if (res.type == 'Office')
    {
        ret = false;
    } elseif (res.type == 'IT') {
        ret = true;
    }

    if ((BudgetManager.query(this.dept_id).type_budget(res.type) > res.amount)
    && ret == null)
    {
        ret = true;
    }

    return ret;
}
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

Of course these are very simple, but you could see as the rules pile up, they could get more and more ugly and complex. You could use Java in this case, but should you use something easier to write? You could use Ruby to create a very simple domain-specific language to manage these rules.

```
def physical_resource(res)
  return false if res.type == 'Office'
  return true if res.type == 'IT'
  return BudgetManager.find_by_dept_and_type(dept_id, res.type) >= res.amount
end
```

As you can see, the Ruby implementation is cleaner, leaner, and certainly more expressive than the previous Java implementation. Who doesn't like that, right? Using Ruby as a scripting language can allow you to use existing systems with the power and expressiveness of Ruby. This is just a quick example, but we'll cover these concepts deeper in Appendix (***). Let's move on to looking at Ruby as a glue language.

3.1.2 Ruby as a glue language

Coming in to this section, you may be asking yourself, "What makes Ruby such a great glue language?" We can think of several reasons. For one, Ruby is a high level language, you can do a lot of things quickly, with a few lines of code. If you need to glue together some code to migrate your database from one server to another, a program you'll use today and throw away tomorrow, you don't want to spend a lot of time developing it. Using Java to develop this sort of thing would likely take too long, but Ruby's brevity and libraries make it quick. This is not a research effort: you want to get it done and out of the way. And you already know Ruby's killer feature is productivity.

Glue scripts do a little bit of work, but most of the stuff happens in other applications. A glue program may take the report from the sales system's web application, convert it to PDF, upload it to the content management system, and then e-mail everyone on the sales team. Let's stop for a second at the step where we convert an HTML page into a PDF. What options have we got?

- Use the Ruby doc_convert library.
- Run another program, say htmldoc or html2pdf
- Use an on-line service, like the one provided by Adobe.

We'll chose the simplest options that works. Sometimes we'll be using the Ruby library, generally because it gives us more control and can do more detail work. Sometimes we'll run another program, especially if we want to do it in one line of code. And sometimes we'll use a service because it's already there, and if we just use it we won't have to work late into the night. It's tasks like this one that you can perform that are not part of an existing application or system, but rather than write something new and reinvent the wheel, you can use other applications to quickly implement these tasks.

A glue language gives you that ability, by letting you run the program directly, and control other applications. You can glue together a few applications to make something all new: think of it as an application mash-up. Think about when you were a child. Let's say you had a lot of pieces of paper, some cut into specific shapes or made of specific colors, Apart, they were merely loose pieces of paper, but glued together they became a new creation: possibly a dinosaur or a panda bear! It's the same concept for glue languages; get a visual for this from Figure 3.1.

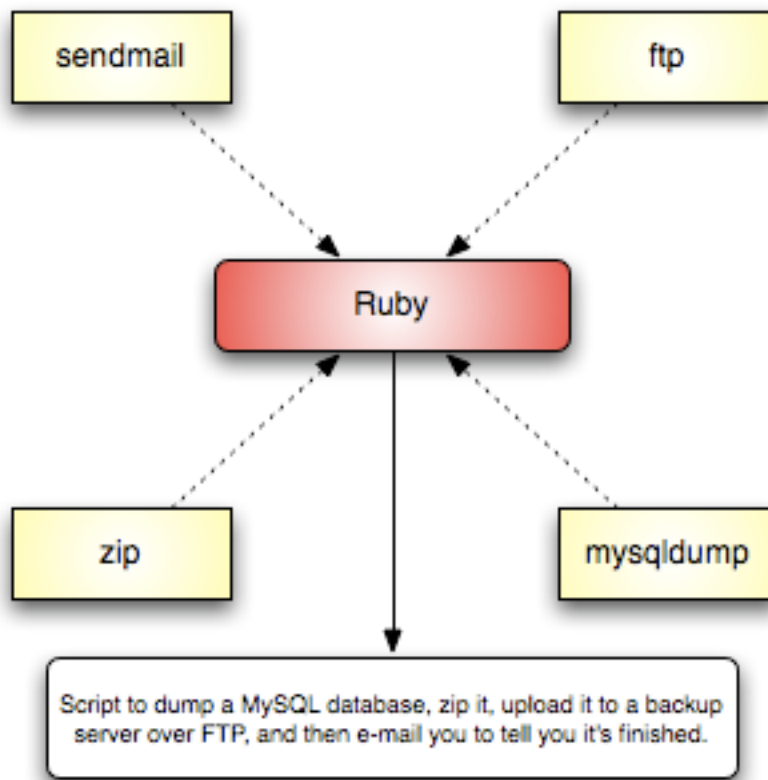


Figure 3.1: Glue languages allow you to glue components together quickly and easily to form something all new. Here we see Ruby gluing together a number of Linux applications to form a new script which will backup your database.

Use Ruby to glue together `mysqldump`, `zip`, `ftp` and `sendmail`, and now you have a script that can create a backup of your database, compress it, FTP it over to a backup server, and e-mail to let you now it ran successfully. You'll quickly recognize that Ruby has libraries for doing all of that, talking to the database server, creating ZIP files, moving files via FTP, sending and receiving e-mails. And if you need more control and capabilities you'll want to use those. But most often, calling existing programs is much easier and will get the job done in no time.

Let's take a look at a few examples. Assaf has one application where he uses Rails' `ActiveMailer` to send e-mails alerts to users, with rich HTML content created dynamically from the database. But he also wants the server logs sent to him every morning, they show any abnormal activity, like slow queries and deadlocks. Since it's just a plain text file that doesn't need any bells and whistles, he chose to use simple *sendmail*. The daily script that does some other maintenance task includes this code:

```
task "send_log" do
  log = File.join(RAILS_ROOT, "log", "production.log")
  sh "sendmail", "assaf@labnotes.org", "<", log
  task("clear_logs").invoke
end
```

Ruby lets you call programs, fork and kill processes, pipe data between applications, work with the file system, download from and upload to HTTP servers, open SSH sessions, send and receive e-mails, process text, and much more. Doesn't sound like much? Well, the simple ability to run programs and fork processes is

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

enough to backup and restore a database, login to and reboot a remote server, check code from version control, reconfigure a Web server, replicate data between two disks, and much more. Gluing together simple components can make something much more complex and useful.

Let's say you needed a script to zip and rotate your log files if they're getting too big. You could write a Bash script or some such to do that for you, or you could use Ruby for the same effect, with even better syntax and possibility for extension. Look at Listing 3.1 for an example script.

Listing 3.1: A script to zip log files and roll them over to an archive

```
LOGS_PATH = '/home/myuser/logs' #1
ARCHIVE_PATH = '/mnt/arc' |
MAX = 5000000 |
date_stamp = #{Time.now.strftime(%m%d%y)}.to_s |
holding_path = File.join LOGS_PATH, date_stamp |

puts "Starting up log archiver"

%w( http.log ftp.log access.log ).each do |log| #2
  log_file = File.join LOGS_PATH, log
  if File.size(log_file) > MAX #3
    puts "Archiving #{log}"

    if (!File.directory?(holding_path)) #4
      `mkdir #{holding_path}` |
    end |

    `cp #{log_file} #{File.join holding_path, log}` #5
    File.truncate log_file, 0 |
  end
end

`zip -r #{date_stamp} #{holding_path}` #6
`cp #{date_stamp}.zip #{ARCHIVE_PATH}` |

puts "Logs archived"
<#1 Setup file paths and other variables>
<#2 Create the collection of log file names>
<#3 Check the log file's size>
<#4 Check the existence of the holding directory and create it if not found>
<#5 Copy the log file to the holding directory>
<#6 Zip the log files and copy the zip into the archive>
```

First, we define some setup variables (file paths and such) #1. Then we tell Ruby to iterate over a collection of log files (#2), check if they're over a certain size (#3), and if so, create a directory to hold all of the "too big" log files (#4), copy that file to the holding directory, and then truncate it (that is, make it zero length) #6. In this one script, we've glued together `cp`, `mkdir`, and `zip` to make a pretty solid log archiving solution.

Let's take a look at another quick example. Let's say you need to consolidate a number of SQL, each creating and populating a different set of tables. These SQL statements update one database, but you want them to update a different database. Listing 3.2 is a quick and easy way to do this.

Listing 3.2: A script to merge SQL files and change the database name

```
DATABASE = "midas_test" #1

['users.sql', 'data.sql', 'codetables.sql'].each do |file| #2
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```

`cat #{file} >> combined.sql`
end

`cat combined.sql | sed -e 's/'*.\\.(.*)'/'#{DATABASE}.$1"/g' > combined.qml`
#3
`mysql -utest < combined.sql`
#4
<#1 Set a variable for the database name>
<#2 Concatenate files to new file>
<#3 Use sed to replace database name>
<#4 Feed it to mySQL>

```

First, we create a variable for the new database name #1, and then loop through a set of SQL files, concatenating each one to a new SQL named combined.sql #2. Finally, we echo the file into sed, which replaces any row/column name with the new database name and writes it back to combined.sql. We glued together cat, bash, and sed to make a very fast script.

If you need more flow control or input, you start writing more Ruby code to shore it up. Maybe you might want to make the sure SQL files exist before you merge them, and if not, log an error. Maybe you'll want to find the IP address of a remote database server using Ruby's network libraries. Remember that Ruby is a general purpose programming language, so there's no limit to what you can do. Call a Web service using SOAP, make queries to the database, edit PDF and Word documents, manipulate images, do number crunching: Ruby is a general purpose language that can do general purpose tasks.

This general purpose is part of what makes Ruby compelling as a glue language: you can easily from a simple little glue script, all the way to a full fledged application. In the Ruby world, they often live side by side. For example, say you're building a new Web application using Rails, and most of the code is in the Web server in front of the users. But there's a lot of management tasks you need to take care of: deploying new releases, backup up the database, rotating log files, repairing tables, deleting user accounts, monitoring the server, and so forth. Most of these tasks are easier to do if you have a nice GUI front-end. But who wants to sit in front of their computer every night and backup the database? Or rotate log files? Or to go through five Web console pages in order to deploy a new release? So you write a little script that does the job, and fire it up.

If you're not using a glue language today, you'll realize something amazing when you start. Because glue languages are so easy to work with, you'll want to automate a lot of tasks that are painful today. You'll also find that some tasks that would have taken you quite a while, can actually be handled by existing applications faster, easier, and better. All of a sudden you'll find yourself more organized, getting more stuff done, and using computers in ways you couldn't imagine before.

Now that we've looked at how Ruby glues together pieces into components and applications, let's broaden our view and look at how Ruby glues, or integrates, components and applications together.

3.2 Integration strategies

Think back again to your childhood. When you were child, how did you glue things together? Were you a "dab will do" sort of kid, using just a little bit (sometimes not enough), or did you really slather it on, so much so that it ran off the edge? Of course, as you grew older you learned to use just right the amount: some things only required a little glue but other tasks required half a bottle. We're working toward that with Ruby; after looking at Ruby as a glue language, it will be beneficial to take a look at how Ruby can glue larger pieces together, or integrate them. Just like when you were a kid, Ruby lets you use a little or a lot of glue to glue your systems together, but it's important to weigh the benefits and costs of each approach. There are two overarching approaches we will look at: tight integration and loose integration.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

3.2.1 Tight Integration

Tight integration is the integration of components that directly interface with each other. You tightly integrate C++ libraries into your .NET application; you tightly integrate that J2EE component into your Java application. Tightly integrating components is about directly interfacing two unlike technologies and making them seem at least somewhat native to one another.

ALTERNATIVE IMPLEMENTATIONS

The ultimate way to get a native feel for non-native code is to have it natively interpreted one way or the other. That's what a lot of Rubyists have been working on by implementing Ruby on things like the JVM with JRuby and the .NET CLR with RubyCLR.

There are typically two methods for doing tight integration: a bridge and an interface generator. The bridge approach works by creating bridge methods that give non-native methods a close to native appearance. For example, using Yet Another Java Bridge, you can use Java classes in an awkward but somewhat native way. See Listing 3.3 for a quick example of showing a dialog message with YAJB.

Listing 3.3: A quick YAJB (Yet Another Java Bridge) example

```
require 'yajb/jbridge' #1

include JavaBridge
jimport "javax.swing.*" #2

:JOptionPane.jclass.showMessageDialog( nil, "Ruby likes Java!") #3
<#1 Import the YAJB library and include the module to get its methods>
<#2 Use jimport to import a Java class>
<#3 Show the message dialog>
```

As you can tell, it's easy to use. First, we do some setup by requiring the library and mixing in the JavaBridge module #1. Then we use the jimport method from the JavaBridge module to import a Java class #2. At this point, we can use JOptionPane or any other Swing class right in our Ruby code, just as we do here (#3) to show a message dialog that says "Ruby likes Java!"

Using an interface generator, like SWIG (the Simplified Wrapper and Interface Generator), you can generate a native interface to non-native code. It's a fairly complicated process, but it's worth it for the benefit of not having to rewrite tons of native code that you need to use. Tight integration is a more advanced topic that we want to cover in this book, but if you're interested in learning more, head over to Appendix B. There you can find more information about JRuby, RubyCLR and other works in that area.

But what about components and systems that aren't "touching"? What if they're not on the same system or if they're networked? That's where loose integration comes into play.

3.2.2 Loose Integration

Loose integration is the integration of components that do not "touch." These components may be networked or simply independent processes on the same physical machine, but regardless of their proximity, they are not closely related systems. The job of loose integration, then, is to pull these components "closer" and forge them into a working system. You may ask, "When should I use this sort of integration? Does it fit my situation?"

When should I consider loose integration?

There are a number of situations where a loosely integrated architecture is ideal; seemingly, the most obvious situation is when the system is very large or disparate. Loose integration also helps with reuse and breaking down application silos, as we'll see when we get to talk about SOA and Web services.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

If you have one component living on ServerA and another component on ServerB, it's impossible to have components directly interface, and as a result, it's impossible to use tight integration schemes. It in this realm of disparate systems where many of these loose integration architectures came into being and still remain the dominant players.

The second most apparent reason to use loose integration is that you have something you can't readily fix, but by adding another layer to it, you can expand its functionality. That's a reality in the business world, and one place where you'll find that Ruby can breath new life into old code.

For example, Jeremy recently finished working on a project to integrate donor data for a non-profit, sharing data between an older system and a newer one. The portion of the old C-based system system they needed to access wouldn't change very much, but they still needed to be able to pull a lot of data from it and push minimal data back. Rather than trying to rewrite the functionality or using an automate dump and reload script, Jeremy used Ruby to create a REST web service to let the new system access the old system's data. The service sat on the server, and when a request came in, executed the requested action: a POST request created a new data row, GET returned data, DELETE deleted a data row, and so on. It was impossible to access the database's proprietary format externally or expand the old system to allow for networked access, so a new service had to be created that would locally interface with the application and expose the data to the new system. When a system can't be expanded in itself, you can always add layers to it to expand the functionality.

Another place where loose integration is well-suited is a heterogeneous environment. Let's say you need a C++ legacy application and a Java application to communicate or share data. Neither are written to do so and both live on separate physical machines. You can rewrite the code to all run in the same language, but you'll find that it's much easier to write a little bit of code to glue these applications together. That way you can make systems of different languages, eras, protocols, and conventions to communicate with each other, all facilitated by a glue script. In fact, anytime you have little or no access to the innards of the application you need to interface with is an opportunity to write a glue script to integrate the applications.

Imminent reuse by another system is another reason to look at looser integration. If you plan on reusing a component in a number of scenarios, turning it into a service or some other exposed resource may serve you better than trying to duplicate the code in a number of places or turning it into a library that you have to deploy in multiple machines. This, of course, depends on the nature of what is being used, performance issues, your network, and so on, but in general, a service-oriented architecture is the best solution in these situations. You know better than us which type of integration is best for each of your projects. Let's take a look at the first type of this loose integration: shared resource integration.

Shared resource integration

The crudest form of integration between applications who may or may not share the same physical machine is resource sharing. In this setup, two applications interact (typically asynchronously) through the sharing of a resource such as a database database server or the file system. See Figure 3.2 for a simple visual of what we're talking about.

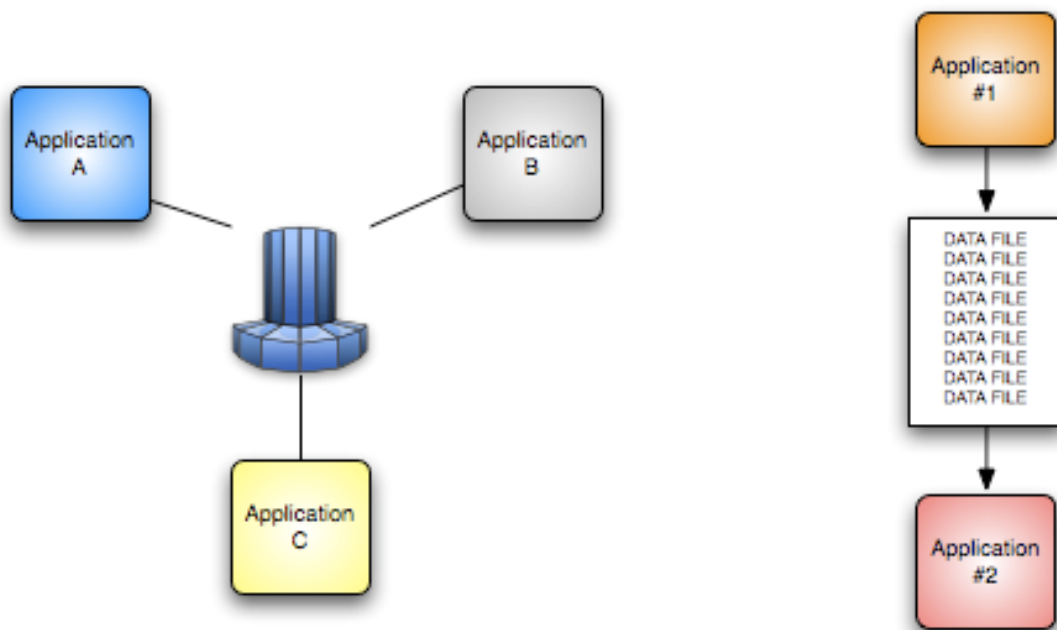


Figure 3.2: Shares resource integration uses a database or data file as a data bus for integrating applications.

As you can see, two or three applications may use the database to communicate among themselves or text file dumps as data buses. This is probably one of the most common ways to integrate legacy applications. Jeremy built a system once that needed to integrate data from a very old directory system with a new identity tracking system the client was implementing. The suggestion for a migration to LDAP was denied because of some secondary functionality of the system and some data that wasn't exportable (which means the guy that wrote it still worked there and he didn't want to work on the app any more to make it migratable), so the decision was to use Ruby build a service that used text dumps of the data from the old system which were exposed to the new system through a web service. Then, actions and data from the new system were made into text dumps and fed back into the old system by a scheduled script. It wasn't the most ideal solution, but it got the job done. Jeremy could have spent a lot of time trying to reverse engineer the database format or hook into the application somehow, but using Ruby and its glue abilities allowed him to get the job done faster and just as well.

But this setup isn't necessarily only for legacy applications. While crude, it is the best option when simple is more important than robust. Rather than muck around with protocols and libraries, you can use the native file system or database server. This simplicity affords you a degree of freedom in that you don't have to worry much about protocol implementation inconsistencies or serialization issues. Granted, we recommended you choose to develop services for any new application you're building, but sometimes that's not an option. There's existing code and you have to live with it.

Shared resources do place hard limits on what can be done. For example, concurrency is an issue. If two components attempt to update the same data at the same time, then the data will likely be compromised or out of date for the next access. A little corruption and some component out of sync and you've got a pretty big mess to clean up. Most applications that dare use this architecture rely on database transactions to deal with concurrency, or use locks files to serialize access to the data. If this approach seems too rudimentary for you or you're worried about the shortcomings of this approach, shared resource integration's shortcomings are, for the most part, been overcome in other solutions such as asynchronous messaging.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

Messaging mechanisms

The shortcomings of shared resource integration have caused developers to look at other solutions for integration, one of which is *asynchronous messaging*. Even though messaging emerged in the 1980s as one of the earliest answers to application integration, it has seen recent revival and reinvention. The basic concept is this: there are message producers and message consumers. The message producers create messages, which hold a payload (i.e., the information that needs to be sent to the consumer). A message queuing system receives this message, and a message consumer will take the message from the queue and react to the payload (e.g., if the payload was new account data, the consumer creates a new account with that data). Figure 3.3 shows a simple visual of a typical messaging setup.

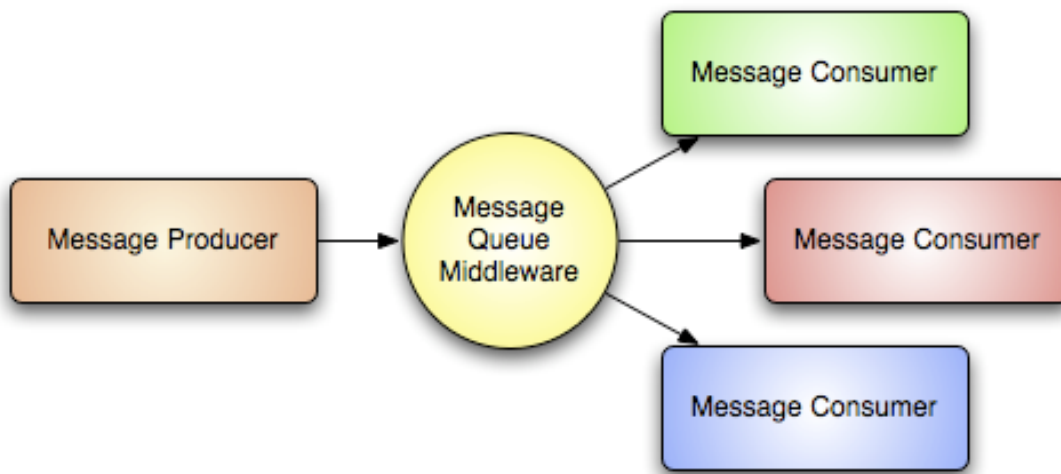


Figure 3.3: Messaging is a fairly simple architecture, but encapsulates a lot of advanced functionality that solves the problems of many other approaches.

Messaging fixes many of fundamental problems with application integration. Unlike shared resource integration, it grants you a degree of robustness and security. Unlike point-to-point protocols, it doesn't assume that both endpoints are up at the same time, and it sidesteps the issue of serialization since data sent in a payload doesn't necessary have to be serialized. The middleware assures proper routing, persistence, and delivery of messages, taking the pain of implementing your own algorithms to deal with message loss.

How does Ruby stack up in that department? Ruby offers a fairly large number of asynchronous messaging solutions. You can choose native Ruby implementations, like *reliable-msg* (written by Assaf Arkin, one of the authors of this book) . If you prefer using ActiveMQ, you can use the Stomp library to talk to your ActiveMQ servers; we know of companies using that to integrate Ruby front-ends with Java back-ends. There's also work on implementing the Advanced Message Queuing Protocol (AMQP), that lets you talk to many more MoM products, check the Qpid product. If asynchronous messaging won't work with your current architecture or you would prefer to take an alternative route, you may want to consider using Ruby's abilities to work with service-oriented architectures

Service-oriented architectures

One of the most popular architectures for large-scale systems is a *service-oriented architecture* (or SOA). If you're building a new application, you're probably looking at the role services play in helping you develop large applications out of loosely coupled components. Buzzword compliance aside, there are some real benefits to developing services as part of your overall architecture. Think of services as a portfolio of features

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

that you can use in one application today, and then reuse in another application tomorrow. You're doing all the work upfront, so you don't have to spend development cycles retrofitting integration after the fact. And because these services are available, the barrier to using them are lower, and you end up with more interesting ways of using them.

If you want a compelling example, look no further than the proliferation of mashups on the Web. What happens when you want to see real estate prices across different geographical areas? You can grab the data, copy it into Excel, organize it into geographical areas, and try to make sense of the resulting data. Or you can mashup a real estate listing site with an online map for a quick visual overview that gets updated in real time. Why so many mashups? Because they make life easier. But also because they're so easy to do. The information is readily available through open and standard protocols, like HTTP, HTML, RSS, JSON to name a few.

Service Oriented Architecture works like that. You may not know that you need these services right now, but once you have them, and the barriers to using them are low, you'll find new and interesting ways of using them. And it changes how you think of large scale systems, from talking about integration to talking about composition. We often hear the question "how do I integrate my new Ruby code with my existing Java code?" We talked about some of the options, like using tight integration (JRuby, JRB), and loose integration (ActiveMQ, HTTP, shared resources). But this might be a good opportunity to switch gears and talk about composition instead.

Let's say you're building your applications by composing services. You have a service to manage the inventory, another service to handle billing and invoicing, another to handle the company's personnel information, and so on. You have another service for handling accounts, which is a composition of the other two services with additional logic, an application in itself. It's also a service in that someday you may pull it into a large application. As you can see, service composition is recursive and you can always build larger and larger applications, much like building things out of Lego. One day the system we just described may look like Figure 3.4.

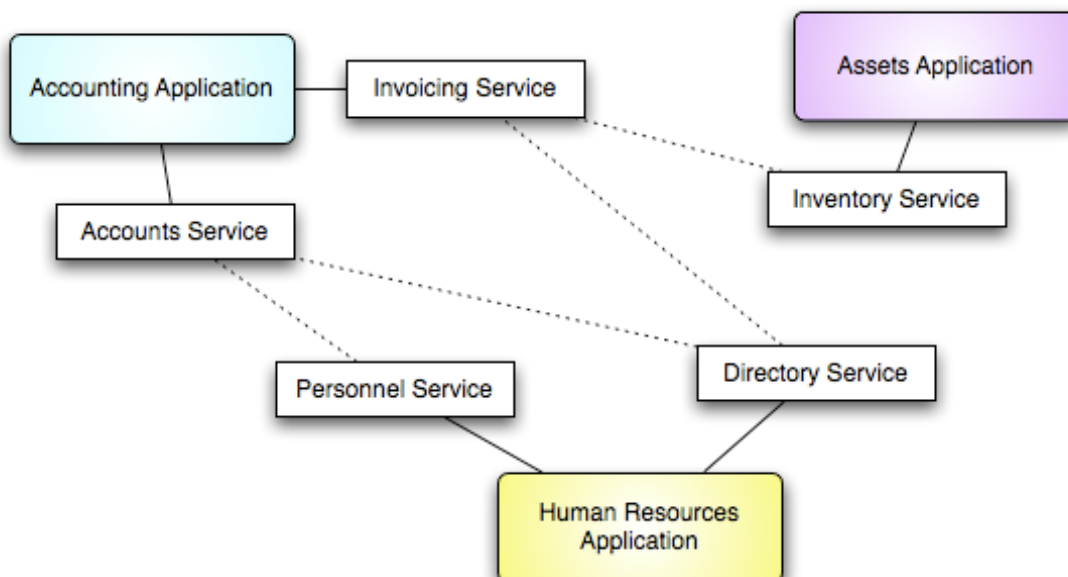


Figure 3.4: A service-oriented architecture involves breaking your application into distinct services reusable by other applications.

When you look at your software in terms of services like this, you no longer ask: "How do I integrate Ruby with Java?" Instead, you compose services together. One may be written in Ruby, another may be written in Java. Sometimes you'll want to change the implementation, maybe rewrite the entire thing to improve quality

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

and performance, and in doing so change the implementation language. Maybe a service was provided by one system today, and you're switching to another system tomorrow.

Going back to the real estate listing and mapping mashup, do we really care which languages were used to implement either of these? Does it matter if the real estate listing service is written in Perl or C++? If the mapping service is written in Python or Java? If your mashup is written in Ruby or PHP? So think about composite applications as the way to build new applications, and services as ways to expose existing functionality. Obviously not everything is a service as good services are coarse grained, but if your architecture is focused on services, you'll get more bang out of your development money.

To make this happen, though, you need to choose protocols that work across different languages and platforms. We mentioned open and standard protocols: RMI in the Java world, DCOM in the Windows world, DRb in the Ruby world, all tie you up to a specific language or platform. On the other hand REST, SOAP, JSON and their likes help you work across languages and platforms. They allow you loosely couple components together and build new systems, to create mashups regardless of language or platform.

So how well does Ruby fit in this world of loosely coupled services? Ruby has great support for SOAP and WSDL, which typically go hand in hand. You can use the SOAP4R library to call SOAP services implemented in any other language, and to expose Rails services to other applications. If you're working with Rails, you can add SOAP services to your Rails application with the same ease. We'll show you how to do all of that in Chapter 5.

Typically the messages you'll be working with are composed of XML, and Ruby has several excellent choices for dealing with XML documents. Most people start with REXML, the Ruby equivalent of DOM, but another, faster option is libxml. There are also libraries that use the meta-programming pixie dust to make like easier, like XmlSimple and Builder. You can also use something like ROXML to marshal Ruby objects to and from an XML string. We'll talk more about those in Chapter 10.

SOAP works well in many situations and is widely used, but sometimes you'll want to use other protocols. XML-RPC is a lightweight alternative that will work better, especially with legacy services that predate SOAP. And then there are protocols like ebXML and RosettaNet, that you can easily use with the simplicity of Ruby's built-in HTTP and XML libraries.

Ruby can also work well with REST web services. The beauty of REST is that, as a way to design Web services, it works better for front-end applications like the one you build using HTML and JavaScript, but you can also add machines processed messages, so you get a single service that offers both a remote UI and a remote API. We'll show you how easy it is to extend you existing Rails application to offer an API based on XML and JSON (JavaScript Object Notatio) in Chapter 5.

Why would you want to do all three? You already know that the easiest way to power users is with a Web-based application, one that combines HTML and JavaScript. You'll want to make it a rich UI application, where AJAX (Asynchronous JavaScript and XML) excels. JSON works very well with AJAX. Not only that, you can easily compose new applications in JavaScript, using your existing JSON services. Add a touch of XML, and you can also use these services from back-end applications. That may sound like a lot work, but you'll be surprised to find out how easy it is to pull off, especially with Rails.

Reading where JSON works well, you might be tempted to use JSON in places you currently use XML, and you actually may be on to what the rest of world is beginning to realize. While JSON sounds like a solution only good for AJAX applications, it is in fact supported by all modern programming languages. Ruby has support for serializing data to and from JSON. And if you look at the www.json.org Web site, you'll find libraries for Java, C#, PHP, Python and just about any language you can think of. JSON is quickly becoming a lightweight alternative to XML.

Let us offer a word of caution, though. While REST has more uses than SOAP: it gives you a single way to build applications that have a rich user interface, the ability to compose applications in the browser, and a

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

way to interconnect distributed systems. And with Ruby, especially Rails, you'll wonder why you didn't use REST before. But not all platforms have such an excellent support for REST just yet. At the time of this writing, work has just started on improving REST support in Java, since the existing libraries are hard to work with and non-standard. Likewise, languages like C# and PHP are catching up. Right now, in many cases, you'll find that REST is tempting but SOAP is your preferred choice.

Now that we've taken a good look at the options available, it will be beneficial to take a look some extended examples of how Ruby can be (and is being) used to integrate systems.

3.3 Case Studies

Now that we've looked at the concepts underlying integration with Ruby, let's take a look at some extended examples of how Ruby can be (and is being) used to integrate systems. These case studies are mildly anonymized real projects that we've worked on, and we thought it would be good to take these concepts and show two things: they do work and Ruby can make it happen.

3.3.1 Messaging: The Grant Foundation

The Grant Foundation needed a system to handle signups for their program, and then automatically place them based on some business rules that they had laid out in their program: if a person made a certain amount of money, they qualified for Program A but not Program B; if a person lived in certain state, they didn't qualify for Program C like everyone else but did qualify for Special Program 1. The Foundation decided to bring in some very "professional" consultants to build this system. The consultants got to work and produced something 5 months later.

The First System

At first glance, The First System seemed to be pretty good. On the front-end, they had built a Java application that would process signups, and post them to an ActiveMQ messaging server. A backend application would grab those signups and place the people into programs based on certain criteria and business rules. A reporting application would periodically grab data from the database and generate a report. Figure 3.5 gives you a basic visual of the system they had built.

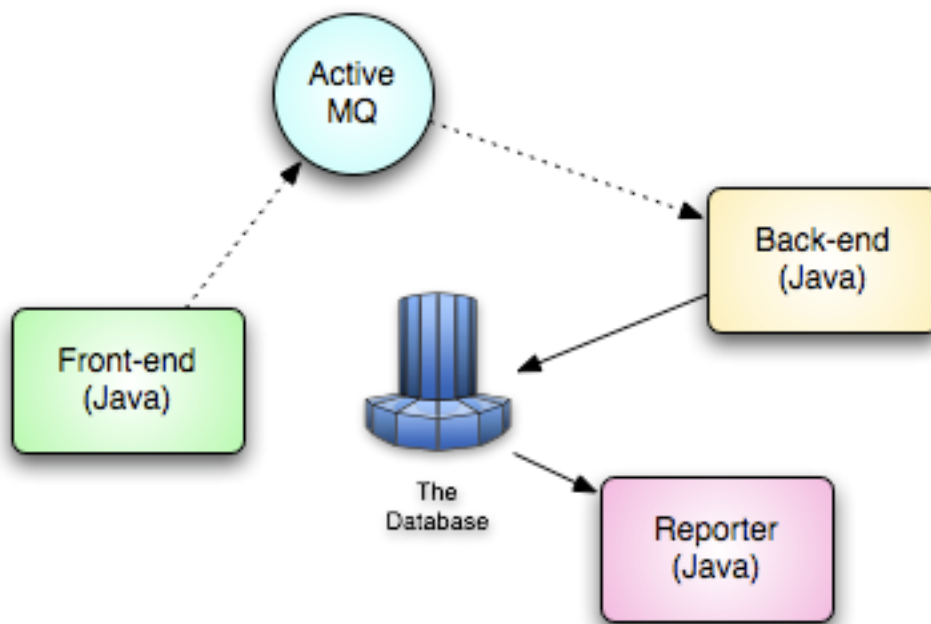


Figure 3.5: The original design for The Grant Foundation's system was purely Java, but Jeremy soon changed that.

That seems like a decent system design, but there were a few underlying problems. First, the backend application that actually placed people crashed at least two or three times a day. The database code didn't use transactions and had a bunch of queries, so the anomalous data left in the database by a failed database operation made the reporting script crash constantly. The consultants didn't know what the problem was, but were going to charge overtime to find out. They actually switched from using web services to asynchronous messaging because of the constantly crashing application. A cron job had to be set up to run every half hour to make sure the process was still running, and if it wasn't, restart it. Rather than pay the original consultants to maybe or maybe not fix the problem, Jeremy and a colleague were brought in to see if they could fix the problem and salvage the system.

Second, the code for the backend system turned out to be a complete mess: tons of spaghetti code, methods that were pointless or duplicated built in functionality (*two* methods to replace substrings in a string), incredibly massive database queries. Jeremy's only guess is that one team built the front-end and one team built the backend because the front-end code was decently written. There was very little they could do to save the system as is without billing a lot of unnecessary hours (which isn't helpful to a new non-profit organization whose purpose is to give money away), so they went for Plan B.

Plan B: Ruby

It was decided that the backend application would be rewritten in Ruby. Jeremy and his colleague could write a new one faster than they could debug the old one, while still being able to fully integrate with the existing code. The stomp library was used to talk to the ActiveMQ server; changing to a web service or a different middleware wouldn't have been a productive alternative regardless of the functionality gains. For the database, they used ActiveRecord; this made the database code not only quicker to write but also easier to maintain. Any future changes to the schema required no change to the code (unless columns were removed).

Using Ruby would also make it easier to change business rules (another reason the system needed to be updated). When the rules were written in Java, it was painful to write new ones and the company workers had

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

no clue how to edit the code (which they didn't really understand anyhow) then compile and deploy it. Since Jeremy was using Ruby, he could use its dynamic features to develop a small DSL to write business rules with; see Listing 3.4 for a quick example of what that looked like.

Listing 3.4: A small sample of the business rule DSL from The Final System.

```
rule "Income Requirement", "Program B" do |person|
  if person.income < 10000
    approved
  end
end
```

The system would work like before except it would be new and improved, with a stable backend and business rules that, due to Ruby's largely readable syntax, even the company workers could change with little training. The Foundation was happy with the plan, so they proceeded to build it.

The Final System

The Final System took about 6 weeks to write and setup. The code for the backend application ended up being one quarter the size of the First System's and ran much more stable (it has crashed once in 4 years). Figure 3.6 shows a visual of how the system ended up looking.

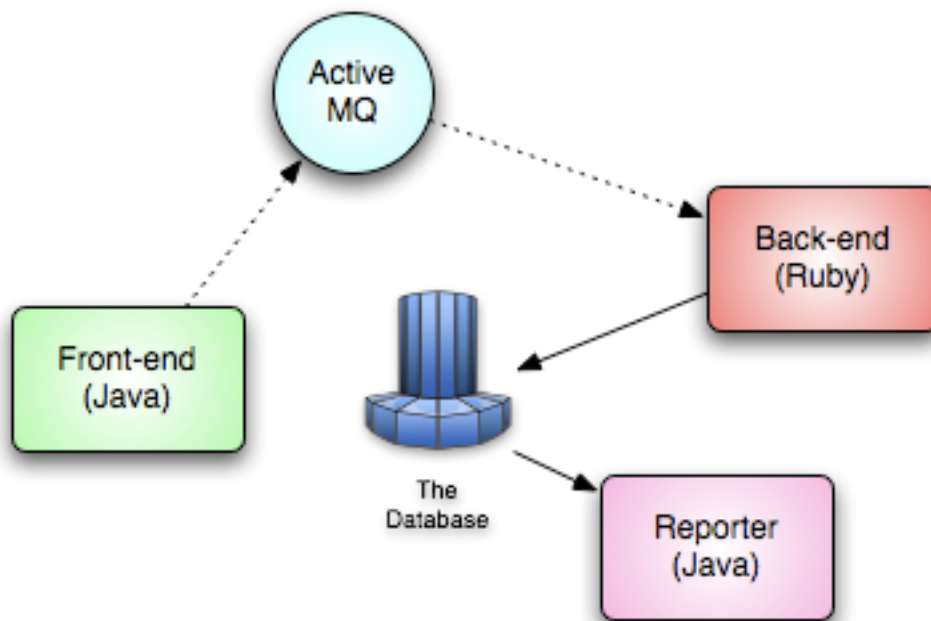


Figure 3.6: The final result of Jeremy's revisions to The Grant Foundation's system. Nothing changed architecturally, but the Ruby system was a perfect functional equivalent to its broken Java predecessor.

The Ruby application dropped right in place and was a perfect functional duplicate of the Java version (without the crashes and such). The restart script could finally be retired, and the system ran (and continues to run) rock solid. In this case, Ruby was able to function in the same manner as a previously existing solution, but in the next case study, we'll look at creating a solution from scratch using Ruby.

3.3.2 XML-RPC: Person Manager

A large organization contact Jeremy about integrating some of their people management software. Their system had become large and complex, but it had little to no cohesiveness. They wanted Jeremy to integrate all of their existing applications into a single control and access console that could easily be used by their administrative staff.

The existing system

The existing system had an ActiveDirectory domain that was used as the main authentication backbone, but attached to that it had a number of stray applications that held different data. This data could be anything from how many times this person had accessed certain resources (e.g., requested software, requested technical support, and so on) to their departmental assignments to their printer quotas: all in separate applications. Figure 3.7 gives you an idea of the logical layout of the system.

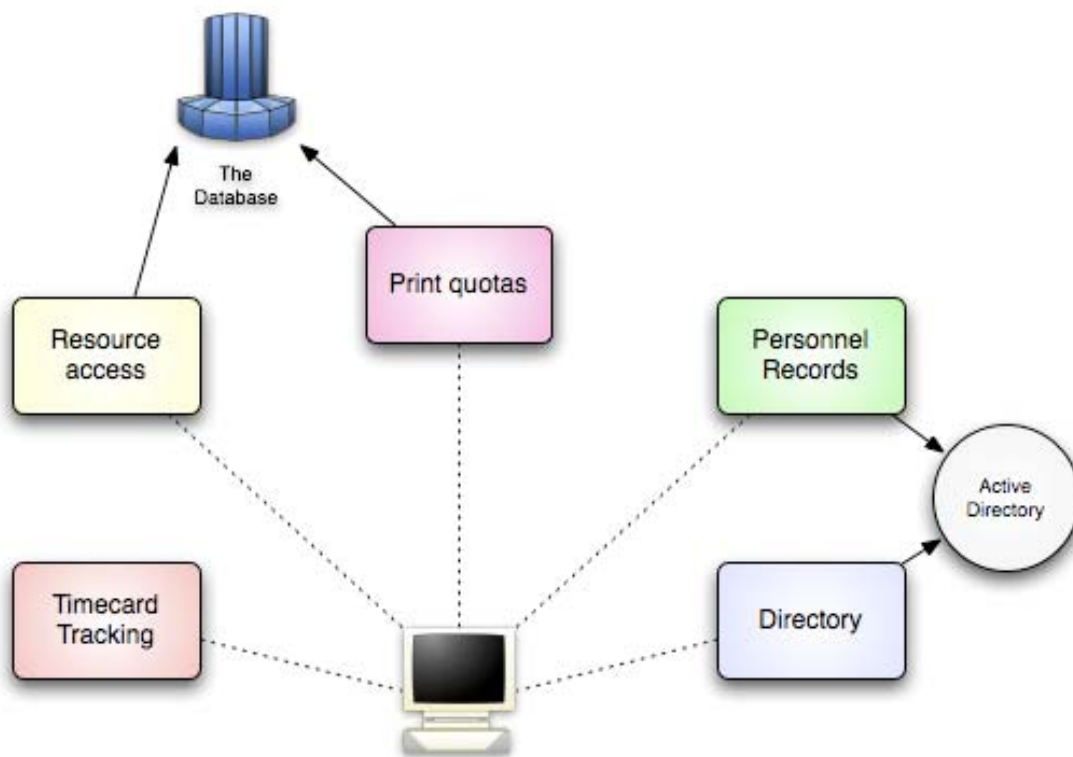


Figure 3.7: The original design for the people management system: a lot of disparate applications that required different clients to access.

The existing system was basically a bunch of information silos that needed to be connected together.

Of course, all of this information living in silos made accessing it needlessly complex, especially for the somewhat computer illiterate administration staff. Using five different, rather old, command line applications to access one person's data was far too complex for them to manage, so they decided it was time for a change.

Gluing it all together with Ruby

Jeremy decided to build XML-RPC interfaces for each of the applications. XML-RPC was chosen not only because of its lightweight implementation, but it is also built-in to the Ruby library. Most applications' source

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

code wasn't available (i.e., only one of them had its source code readily available), so each remaining application would have a small Ruby script that sat next to it on the server; every time a request came into the interface, it would crank up the application (or talk to an already running instance) and get the data requested.

Jeremy would then construct a single console to access all of that data at once, reducing the complexity of their operations five-fold. It would still be a console-based utility, but at least it would be *one* utility rather than five. The client approved, and so Jeremy began work on putting everything together.

The end result

The system took about 3 months to develop and another 2 months to fully test before being deployed (the staff had a party on the day of deployment). The final system ended up looking something like Figure 3.8.

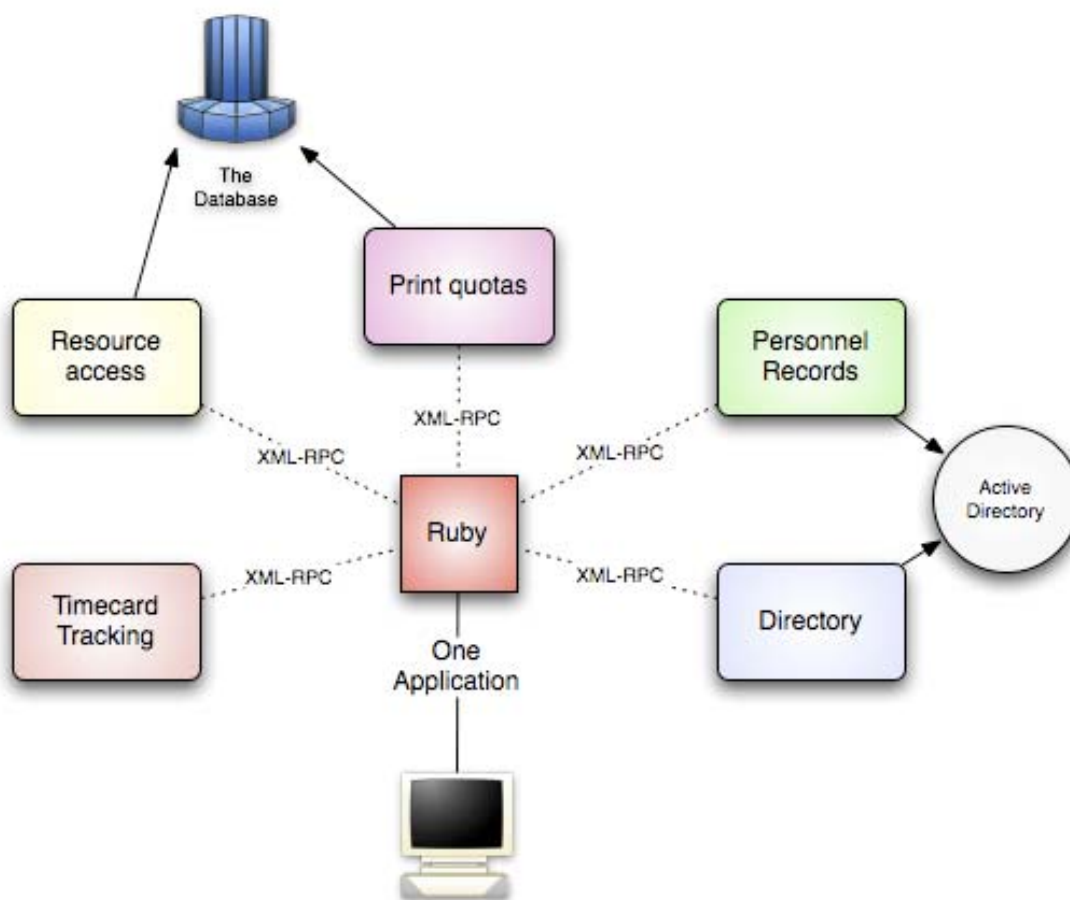


Figure 3.8: The final product: XML-RPC services exposed to a single application. This approach greatly simplified staff workflow.

All of the services were accessed by a single console interface presented to the staff members, which allowed them to ditch the four other applications they were using.

In the console's first incarnation (which most of the staff preferred), it was a command line application, and even though the application was still a text-based console application, it was light-years ahead of the previous setup. Before the new system, they had to use complex system commands (e.g., to lookup a person by their identification number, they had to type "DAT LP @ID=123456"), but the new system allowed simple

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

commands (e.g., “lookup 123456”) or a menu based system. The second incarnation of the console was a small Rails application (which the new staff tends to prefer), that encapsulated all of the behavior of the console application in pretty Rails pages.

3.4 Summary

In this chapter, we took a slightly different look at Ruby (i.e., how it can be used for the specific purposes of scripting and “gluing”) and how to use Ruby in these ways to integrate heterogeneous systems. Ruby can be used for both tight integration, which refers to integrating components that directly interface, and loose integration, which refers to integration that is more disparate, such as asynchronous messaging. Ruby has robust support for both types of integration, and we’ve taken a look at both of them, both in an overview fashion and through a couple of case studies.

So far, we’ve covered a great deal of strategy that surrounds Ruby development. We’ve discussed why you should (and should not) choose Ruby, where to use Ruby, but only a little about how to actually use Ruby. As we move into the next two parts of this book, strategy will take a slightly more minor role and we’ll get more into practical techniques for Ruby usage.

4 Ruby on Rails Techniques

This chapter is not an introduction or tutorial on Rails; rather, it is a discussion of various techniques related to Ruby on Rails. This chapter is meant to fill in just a few of the gaps that many other books leave open when dealing with Rails.

4.1 Extending Rails

A lot of Rails developers see the magic going on in Rails and wonder how to capture that same syntactic sugar in their own helpers or other Ruby scripts. As we've found out, the trick is to use meta-programming (remember that from Chapter 1?) along with a few of Rails built-in mechanisms to gain maximum syntactic sugar while also making it easy to develop.

There are three essential types of extensions to Rails: helpers, libraries, and plugins. Helpers are view-specific extensions that expose new functionality to views or expose deeper functionality from controllers or models to views. Libraries are just that: libraries that you can use to extend Rails' functionality. Plugins are essentially libraries that are in a redistributable format.

4.1.1 Using helpers to expose Ruby libraries

Surprisingly, many new Rails developers seem to misunderstand the relationship between Ruby and Rails. Rails is written in Ruby, and as such can take advantage of any library that Ruby can. The difference between the two arises when either the library places a big tax on system resources (you don't want to add 3-4 seconds on to each request just for a library unless you really have to) and some libraries' syntaxes simply aren't geared towards Rails usage. You can work around this by creating a library of your own.

Problem

You have an existing Ruby library that you would like to expose to a Rails application in a natural way.

Solution

There are three main mechanisms for creating a library or component for Rails: a helper that lives in `app/helpers`, a library that lives in `lib/`, and plugins. In this instance, let's say you want to build a source code browser and employ syntax highlighting. We're going to create a helper to expose the Ultraviolet syntax highlighting library to a Rails application. Ultraviolet's syntax is pretty nice, but you don't really want to put something like the following in your views or controllers if you can help it.

```
result = Uv.parse(code, "xhtml", "ruby", false, "amy")
```

The first parameter is the code you want to parse, followed by the format to output, followed by the syntax file to use (Ultraviolet lets you use Textmate syntax and theme files), followed by `true` or `false` to indicate whether we need line numbers, and finally the name of the theme we want to use. Putting that all over your application will not only junk up the code a bit, but it will also cause a lot of duplication (if we're not using

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

line numbers, only Ruby code, and the same theme all over the place, we shouldn't need to type it in everywhere). We need to figure out a way to make it a little friendlier.

To get started, we'll first need to install Ultraviolet using RubyGems, using a command something like the following:

```
gem install -r ultraviolet --include-dependencies
```

Now we need to start looking at how call to Ultraviolet would ideally look when we call it from a view. We want to simplify as much as possible, perhaps only requiring the code and making the other parameters optional would be the best. Of course, we'll want it to be semantically correct, so the `Uv.parse` call has to go in lieu of a better name method like `highlighted_code` or simply `code`. Perhaps it should look something like the following:

```
code("puts 'hello!' if code.ruby?")
```

So how do we do it? First, we need to either create a helper by creating a new file (we recommend something like `uv_helper.rb`) or edit `application_helper.rb` in the `app/helpers` directory. Inside we need to define a `code` method (shown in `ApplicationHelper` here).

```
require 'rubygems'
require 'uv'

module ApplicationHelper
  def code(code)
    end
end
```

Now we need to look at putting the call to Ultraviolet in place. We could take the code as the sole argument and hard code the other values, but it would be pretty nice to be able to specify a different theme or syntax if we ever needed to (and trivial to add). Listing 4.1 is what our method should look like with these parameters in place.

```
def code(code, format="xhtml", syntax="ruby", line_numbers=false,
  theme="amy")
  Uv.parse(code, format, syntax, line_numbers, theme)
end
```

As you can see, we give the parameters that won't change often default values, so when it's called in a view, the only required argument will be the code we want to highlight.

So, now we need to create a controller, action, and a view and place the following code inside the view.

```
<html>
  <head><title>Syntax highlighting!</title></head>
  <body>
    <%= code('@items.each {|item| puts item }) %>
  </body>
</html>
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

If you view the action in a browser, you should see that the code is monospaced, but it doesn't have any color! That's because we didn't tell Ultraviolet to copy its CSS and other related files to the right place or tell our view to include that CSS file. First, let's add a method to copy the files if we need to.

```
def copy_files
  Uv.copy_files "xhtml", "#{RAILS_ROOT}/public/stylesheets"
  File.rename "#{RAILS_ROOT}/public/stylesheets/css/",
    "#{RAILS_ROOT}/public/stylesheets/syntax/"
end
```

First we tell Ultraviolet to copy the files to our stylesheets directory, and we then rename the directory it creates to something more sensible. Now we need to go back and add a few lines to our `code` method. It should look something like the following.

```
def code(code, format="xhtml", syntax="ruby", line_numbers=false,
  theme="amy")
  unless File.exist?("#{RAILS_ROOT}/public/stylesheets/syntax")
    copy_files
  end

  Uv.parse(code, format, syntax, line_numbers, theme)
end
```

Now the helper checks for the existence of the theme CSS files, and if they aren't there it copies them. This should only happen once, so the performance hit is negligible.

Now we need to create a method to include the stylesheet in our view; we could just use `stylesheet_link_tag` directly, but then we'd have to worry about which theme we were using. In the interest of abstraction and making our lives easier, let's create a custom method.

```
def theme_stylesheet(theme='amy')
  stylesheet_link_tag "syntax/#{theme}"
end
```

Now if we add that to the `<head>` of our view, the code in the view should be properly highlighted, and looking something like Figure 4.1.

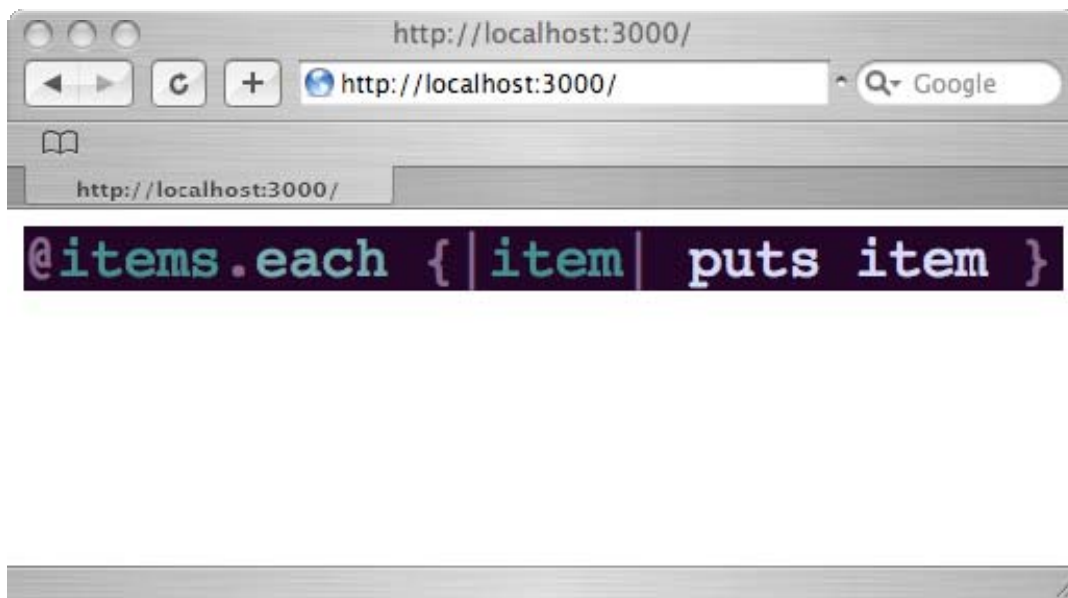


Figure 4.1: The output from our syntax highlighting library.

The final version of our syntax highlighting helper looks like Listing 4.1.

Listing 4.1: The whole Ultraviolet helper

```
require 'rubygems'
require 'uv'

module ApplicationHelper
  def code(code, format="html", syntax="ruby", line_numbers=false,
  theme="amy")
    unless File.exist?("#{RAILS_ROOT}/public/stylesheets/syntax")
      copy_files
    end

    Uv.parse(code, format, syntax, line_numbers, theme)
  end

  def copy_files
    Uv.copy_files "html", "#{RAILS_ROOT}/public/stylesheets"
    File.rename "#{RAILS_ROOT}/public/stylesheets/css/",
      "#{RAILS_ROOT}/public/stylesheets/syntax/"
  end

  def theme_stylesheet(theme='amy')
    stylesheet_link_tag "syntax/#{theme}"
  end
end
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

Discussion

Helpers are a great way to expose third party functionality to your application or to abstract your own logic to avoid duplication or a lot of unnecessary code. In this instance, we could potentially be saving ourselves a lot of keystrokes while also preserving the flexibility of the original API from Ultraviolet. We built-in the most common use case for our needs, and left the rest to be explicitly specified. This “operation by common convention” is how Rails manages to stay so simple and useful: the most common usage is dead simple. Think about how easy it is to map a database to a class or to create really nice AJAX effects.

Of course, this isn’t the only Ruby library you could expose to Rails nor are helpers only useful exposing Ruby libraries to Rails. There are a great number of Ruby gems that would be (or are) very helpful to a Rails application when exposed properly: RedCloth which parses Textile into HTML, any of the loads of time handling libraries such as Chronic, and the list goes on. Helpers are also useful for abstracting away parts of your views that are common: setting a special page title on each page, building your navigation menu, building common layouts and widgets, and so on. But helpers are only really useful to your views; what about the rest of your application?

4.1.2 Meta-programming away duplication

A big part of the Rails design philosophy is DRY, or Don’t Repeat Yourself, but it’s very easy to repeat yourself in a Rails application: putting the same validation in a number of models, repeating that bit of markup on nearly every page, using that same idiom over and over in your controllers. It takes a keen eye to see when to extract pieces of code into reusable pieces, but once you see what needs to be extracted, you need to know what to do with it.

Problem

You have a lot of code that is very similar in structure and function that you would like to consolidate as much as possible.

Solution

When you are building more complex models, your validations and other related calls can become quite large. One of Jeremy’s recent projects sported some models whose validations, plugin calls, and callbacks added up to over 30 lines at times. One way to shorten a lot of the code that builds up in your models is to meta-program and abstract it away.

For example, let’s say you were working on a project that had a lot of URL fields: links for feeds, links for web pages, links for product pages. Your models just have a lot of fields that are URLs. You need to validate the fields, so you write something like the following.

```
validates_presence_of :url
validates_length_of :url, :minimum => 12
validates_format_of :url, :with => /^(http|https):(([A-Za-z0-9$_.+!*(),;\/?:@&~=-])|[%[A-Fa-f0-9]{2}]{2,}){2,}#([a-zA-Z0-9$_.+!*(),;\/?:@&~=-]*)?([A-Za-z0-9$_.+!*(),;\/?:~=-])/, :message => "isn't a valid URL."
```

As you can see, there are just a few simple validations for the field. This doesn’t seem major, until you take a step back and realize that you’re doing these validations somewhere between 25 to 30 times in your application on differently named attributes, sometimes two or three times per model. If you decide to add to one of them or remove one of them, that’s a lot of unnecessary work.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

The better way is to meta-program a little library to help you clean them up. Listing 4.2 shows one such library. Place the code in a file (ours is `named validates_url.rb`) in the `lib/` folder.

Listing 4.2: A URL validation library

```
module ValidatesUrl

  def validates_url(attribute = :url)
    instance_eval do #1
      validates_presence_of attribute
      validates_length_of attribute, :minimum => 12
      validates_format_of attribute, :with => /^(http|https):([A-Za-z0-9$_.+!*(),;\\/?:@&~=-])|[%[A-Fa-f0-9]{2}){2},}#([a-zA-Z0-9][a-zA-Z0-9$_.+!*(),;\\/?:@&~=-]*)?([A-Za-z0-9$_.+!*(),;\\/?:~=-])/, :message => "isn't a valid URL."
    end
  end

end

ActiveRecord::Base.send(:extend, ValidatesUrl) #2
<#1 Evaluate this block in the context of the instance it's called...>
<#2 Add this to every model>
```

Basically what we've done here to wrap the validations in a method, and in that we method wrap them in an `instance_eval` block #1. This means that the code will be executed in the context of the instance from which it's called (in this case, our model), and in doing so, will setup the validations on the model that calls the method. Now we put the validations in, substituting our previous `:url` attribute for a variable which can be set (but defaults to `:url`). Finally, we extend `ActiveRecord::Base` with our method so it's available to all of our models #2.

So, the `validates_url` method is now available to our models. Of course, we'll remove the validations from before so they don't look like the model in Listing 4.3.

Listing 4.3: Our model, pre-meta-programming

```
class Blog < ActiveRecord::Base
  validates_presence_of :url
  validates_length_of :url, :minimum => 12
  validates_format_of :url, :with => /^(http|https):([A-Za-z0-9$_.+!*(),;\\/?:@&~=-])|[%[A-Fa-f0-9]{2}){2},}#([a-zA-Z0-9][a-zA-Z0-9$_.+!*(),;\\/?:@&~=-]*)?([A-Za-z0-9$_.+!*(),;\\/?:~=-])/, :message => "isn't a valid URL."
  validates_presence_of :feed_url
  validates_length_of :feed_url, :minimum => 12
  validates_format_of :feed_url, :with => /^(http|https):([A-Za-z0-9$_.+!*(),;\\/?:@&~=-])|[%[A-Fa-f0-9]{2}){2},}#([a-zA-Z0-9][a-zA-Z0-9$_.+!*(),;\\/?:@&~=-]*)?([A-Za-z0-9$_.+!*(),;\\/?:~=-])/, :message => "isn't a valid URL."
end
```

Instead, they're cleaner and read much better, as you can see in Listing 4.4.

Listing 4.4: Our meta-programmed model

```
class Post < ActiveRecord::Base
  validates_url
  validates_url :feed_url
end
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

end

Discussion

Meta-programming in Ruby is a powerful concept, and it's meta-programming that really gives Rails its power. Methods like `acts_as_list`, Active Record's attribute mapping, and nearly every model-related plugin depend on meta-programming to function. In this example, though, we've really only shown one side of meta-programming. That is, we've shown you how to evaluate generated code in a given context, but meta-programming can also be generated for later use.

Take, for example, the RequiresApproval plugin, which adds the `published!`, `pending!`, `draft!`, and so on methods to any model that you call `requires_approval` in, allowing you to set their status at will. Or ActiveRecord's associations which define accessor methods for the associated objects, which you can then use later on as much as possible. In both of these instances, meta-programming was used to generate methods.

4.1.3 Turning your code into reusable components

After a while, you will build a considerable toolkit of helpers and libraries, but what if you want to share those with your friends, family, fleeting acquaintances, and business colleagues? The easiest way to do that is to extract the code into a plugin.

Problem

You'd like to share your recently meta-programmed code with others as a plugin.

Solution

In this section, we'll show you how to extract your URL validation library into a plugin that you can distribute. First, generate a plugin using `script/generate`.

```
$ script/generate plugin validates_url

create vendor/plugins/validates_url/lib
  create vendor/plugins/validates_url/tasks
  create vendor/plugins/validates_url/test
  create vendor/plugins/validates_url/README
  create vendor/plugins/validates_url/Rakefile
  create vendor/plugins/validates_url/init.rb
  .
  .
  .
  create vendor/plugins/validates_url/test/validates_url_test.rb
```

It should have generated a number of files for you, including a Rake file, a code file, and a few installation necessities. Next, copy the code from your library in `lib/` to the file `validates_url.rb` file in the plugin's directory.

Now that the code is basically in the right place, we need to move some of the pieces around. Pull the call to `extend` out from the bottom of the main code file and place it in `init.rb`. The code in your `init.rb` file is executed in order to initialize the plugin and execute any code hooks (such as extending a class) that your plugin needs to work. Fill out the README with some basic information, and you've just written your first Rails plugin!

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

Discussion

As you can probably tell, Rails plugins are basically libraries and helpers that have an explicit places for code hooks, an install/uninstall script, and a README. Their simplicity, though, makes them very easy to write while still being useful. This example covers the core of writing a plugin, but there are a few little pieces that it doesn't cover.

First, plugins are given install and uninstall scripts, which should be used for any sort of setup/teardown that your plugin needs to do. For example, if we were to port our Ultraviolet helper to a plugin, we would probably want to call `copy_files` in the install task and delete them in the uninstall task rather than doing it on the first request. This would not only knock down the small performance hit that you take from doing it on the first request, but you can also deal with any errors that might arise before your application is running rather than after. Many plugins often use the install hook to show further instructions for installation, like the `restful_authentication` plugin that shows you the README file with further installation and generation instructions.

Plugins also have their own set of Rake tasks, which can be used to do repeated tasks related to the plugin. For example, if you had a plugin that worked with Amazon's S3 and you wanted to clear off your testing bucket, you might have a `rake:bucket:clear` task. These tasks sit right alongside Rails' tasks in the `rake -T` listing, so they're easily accessible to developers using your plugin.

4.2 RESTful Rails

The REST paradigm has been all the rage since David Heinemeier Hansson introduced the Rails community to RESTful Routing/Active Resource in Rails. Now, a lot of developers are attempting to move to the REST paradigm, but for those uninitiated, it can be confusing as to what exactly needs to change in an application to make it RESTful.

4.2.1 A quick REST Introduction

Switching a non-REST Rails application to a RESTful one is mostly a change in perception. The method names stay the same (see Figure 4.2), but when using REST, they can do a lot more tricks.

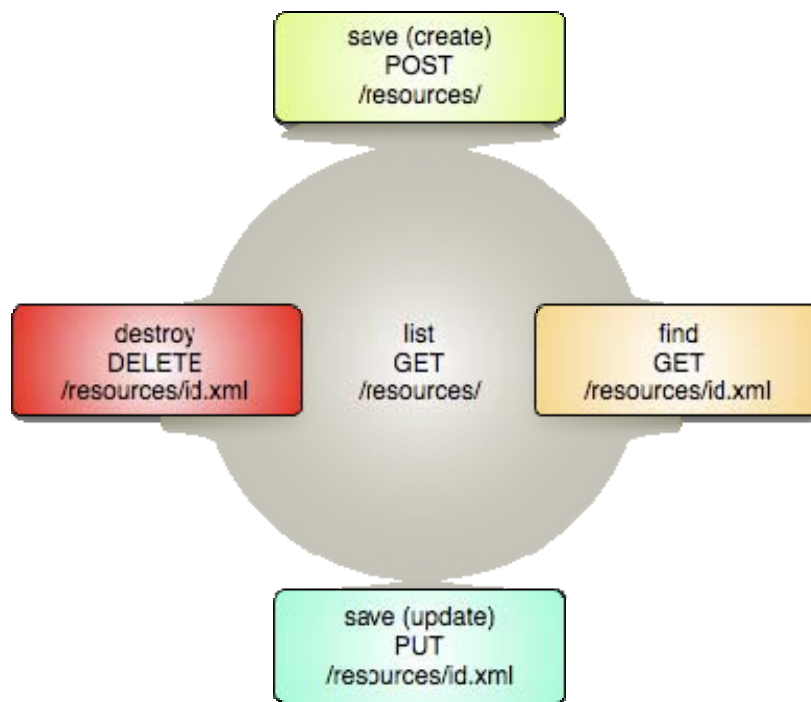


Figure 4.2: Method names in controllers stay the same with REST.

The term REST is an acronym for *Representational State Transfer*; REST is a web application architecture for manipulating web resources using the standard HTTP verbs **GET**, **POST**, **PUT** and **DELETE**. When we speak of REST resources, we mean entities that we can manipulate over HTTP (in our case, Rails models) and that can be rendered in a number of ways depending on what the client requests.

In Rails, using REST changes an application's URLs from the standard `controller/action/id` setup to simply `resource_collection/predicate`. So, for example, you would have `resources/1` instead of `resources/show/1` (see Figure 4.1 for the rest of the changes). But the URL is only part of the equation. The HTTP verbs are just as important: the index and show actions require GET, the update action requires PUT, create requires POST, and the destroy action requires DELETE. Using these pre-chosen URLs allows clients that conform to the REST protocol to interact with your application.

REST CLIENT

Ruby on Rails also has a REST client library named `ActiveResource`. It can interact with your own REST services or other REST enabled services such as Basecamp's remote API. To learn more about it, check out the Rails documentation for the library at <http://api.rubyonrails.org/>.

4.2.2 Refactoring to REST

Refactoring a controller to REST is fairly simple when it comes down to it since Rails takes care of the great majority of the work for you. On the developer's part, it's primarily the shift in frame of mind from actions and objects to verbs and resources. Once you have that, it's usually a smooth transition to REST.

Problem

You need to refactor a Rails application to conform to REST principles.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

Solution

You decide you'd like to refactor the voting application you're working on to use REST. Currently, you have one controller (OptionsController) and two models (Option and Vote) that compose your application. The models are standard ActiveRecord models with no extra methods, and the controller looks like Listing 4.5.

Listing 4.5: Our controller before RESTification

```
class OptionsController < ApplicationController
  def index
    list
    render :action => 'list'
  end

  def list
    @option_pages, @options = paginate :options, :per_page => 10
  end

  def show
    @option = Option.find(params[:id])
  end

  def new
    @option = Option.new
  end

  def create
    @option = Option.new(params[:option])

    if @option.save
      flash[:notice] = 'Option was successfully created.'
      redirect_to :action => 'list'
    else
      render :action => 'new'
    end
  end

  def edit
    @option = Option.find(params[:id])
  end

  def update
    @option = Option.find(params[:id])

    if @option.update_attributes(params[:option])
      flash[:notice] = 'Option was successfully updated.'
      redirect_to :action => 'show', :id => @option
    else
      render :action => 'edit'
    end
  end

  def destroy
    Option.find(params[:id]).destroy
    redirect_to :action => 'list'
  end
end
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```

end

def vote_for #1
  new_vote = Vote.new
  new_vote.option = Option.find(params[:id])
  new_vote.save!

  redirect_to :action => :index
end

def unvote #2
  delete_vote = Vote.find_by_option_id(params[:id])
  delete_vote.destroy

  redirect_to :action => :index
end

def vote_statistics #3
  @count = Vote.count
end
end

```

The **OptionsController** is a pretty standard CRUD controller, except that we have a few extra methods at the end: a method to vote for an option (#1), a method to remove a vote for an option (#2), and a method to gather some statistics on the votes (#3). If you already have a solid CRUD base to work from, REST enabling your application is primarily a three-fold action: resources, routes, and responses.

The general rule of thumb for REST is “one controller per resource.” Upon first examination of the code we’re working with, that seems like what we’re doing here: nothing seems unrelated to **Options**. But, notice the methods that manipulate **Vote** objects. Since they are manipulating a different type of resource (**Vote**), they don’t need to be in the controller responsible for manipulating **Option** objects and need to be in their own **VotesController**. Also, in this separation, we need to consider the method names. What are these methods really doing? The **vote_for** method is essentially creating a vote, the **unvote** method is essentially deleting a vote, and the **vote_statistics** method...well, it gather statistics (and that’s okay). The other methods, though, should be renamed to match Rails’ convention (otherwise the routing won’t really work properly), so our **VotesController** should look like the following.

```

class VotesController < ApplicationController
  def create
    new_vote = Vote.new
    new_vote.option = Option.find(params[:id])
    new_vote.save!

    redirect_to :controller => :options, :action => :index
  end

  def destroy
    delete_vote = Vote.find(params[:id])
    delete_vote.destroy

    redirect_to :controller => :options, :action => :index
  end
end

```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```

def vote_statistics
  @count = Vote.count
end
end

```

The routing portion handled by a set of routes generated by Rails for you. To create RESTful routes, you need to map the resources in your routes file doing something the following:

```

map.resources :options
map.resources :votes

```

The `map.resources` call tells Rails to generate a number of named routes for you, which you can and should then use instead of explicit routes. These named routes will make your redirects and links more semantically natural (e.g., `option_path(id)`, `votes_path`, and so on) and are very convenient (especially since they are generated for you). See Table 4.1 for all of the routes that are generated for you.

Table 4.1: Rails generates named routes for your RESTful enjoyment.

Named route	HTTP Verb	URL	Action
resources_path	GET	resources/	index
	POST		create
resource_path	GET	resources/1	show
	PUT		update
	DELETE		destroy
new_resource_path	GET	resources/new	new
edit_resource_path	GET	resources/edit	edit

But what about the `vote_statistics` method? Rails makes it easy to create routes for custom methods on your REST controllers using parameters to the `map.resources` call. See following snippet for our routes with the declaration for `vote_statistics`.

```

map.resources :options
map.resources :votes, :collection => { :vote_statistics => :get }

```

The `:collection` parameter says that a route without an `id` parameter should be generated (you can also use `:member` to create routes to methods that act on a single resource instance and therefore need an `id` parameter). The routes are specified by giving the method name as the key and the allowed HTTP method name as the value.

Now that we have our routes generated, we need to implement them. Listing 4.6 shows a controller that has been refactored to use these named routes rather than explicit ones.

Listing 4.6 A controller that has been refactored to use named routes

```

class OptionsController < ApplicationController
  def index
    @option_pages, @options = paginate :options, :per_page => 10
  end

  def show
    @option = Option.find(params[:id])
  end

  def new
    @option = Option.new
  end
end

```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```

def create
  @option = Option.new(params[:option])

  if @option.save
    flash[:notice] = 'Option was successfully created.'
    redirect_to options_path #1
  else
    render new_option_path
  end
end

def edit
  @option = Option.find(params[:id])
end

def update
  @option = Option.find(params[:id])

  if @option.update_attributes(params[:option])
    flash[:notice] = 'Option was successfully updated.'
    redirect_to option_path(@option) #2
  else
    render :action => 'edit'
  end
end

def destroy
  Option.find(params[:id]).destroy
  redirect_to options_path
end

class VotesController < ApplicationController
  def create
    new_vote = Vote.new
    new_vote.option = Option.find(params[:id])
    new_vote.save!

    redirect_to options_path
  end

  def destroy
    delete_vote = Vote.find(params[:id])
    delete_vote.destroy

    redirect_to options_path
  end

  def vote_statistics
    @count = Vote.count
  end
end

```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

Replacing the explicit routes with their named equivalent is a subtle change, as you can tell. The route the index method is now `options_path` (#1), the route to a single `Option` is now `option_path` (with the id still given as an argument, just as before) #2, and so on. The change becomes a little less subtle when you need to specify a different HTTP method, such as when deleting or updating. For example, a link to delete an `Option` would require a call to `link_to` that looked like the following.

```
link_to 'Delete', option_path(option.id), :method => :delete
```

Since we're using HTTP verbs to express what we want to do, they have to be specified in certain places, typically using a `:method` parameter like we did here. It's not a hard thing to remember, but it is required.

As for the “responses” portion of a REST refactoring, that's not necessary here since we're concentrating on HTML. In the next section we'll take a look at responding with different formats.

Discussion

As you can see, refactoring to REST from a CRUD setup isn't necessarily a difficult process but simply a different way of thinking. The most difficult part of this shift is focusing on resources and their behaviors rather than actions and controllers. Many Rails developers have developed code like what you saw in this section, where a number of models interact in the same controller, and that's not necessarily a bad thing in CRUD. But when using REST, it's important to understand who your actors are, what they're doing, and then design your application around that. REST is so resource intensive that doing that wrong essentially defeats the point of using it in the first place.

In this section you saw how to create RESTful routes, but we didn't mention one other small detail related to those. We used `map.resources` to map routes to resources that are plural: we have a lot of votes and options. But you can also use `map.resource` for a singular resource. This is useful for things like a login session (`map.resource :session`) or a user profile (`map.resource :profile`). The controller and model setup remain the same, but it makes your URLs a little more “correct” (`session/new` and `profile/1/edit` rather than `sessions/new` and `profiles/1/edit`).

Another routing option you have is to nest your routes. For example, an `Option` has a lot of `Vote` objects attached to it, so you could nest them this way in `routes.rb`:

```
map.resources :options do |options|
  options.resources :votes
end
```

This nesting does two things. First, it makes your URLs a lot more readable. Secondly, it changes the named route for votes. Rather than `vote_path`, it becomes `option_vote_path` (and `option_votes_path`), and rather than simply specifying a `Vote` object's id, you'll have to also provide an `Option` object's `id`. This seems like a lot of work, but it really isn't and it gives your resources more structure if you need it.

Just like non-REST controllers, Rails will automatically look for `action_name.html.erb` and render that for each action, unless another response type has been defined. In the next section, we'll look at how to respond with different formats based on what the client requests.

4.2.3 Exposing a remote web service

Once you have your controller transitioned to REST, it's nearly trivial to have it expose an XML-driven web service interface. Thanks to Rails' `respond_to` method, it's dead simple to expose web services for RESTfully routed controllers.

Problem

You need to expose a RESTful web service to a remote client.

Solution

Using the `respond_to` method you can easily respond with different types based on what the client requests. For example, let's take the `index` action from our previous `OptionsController` example; Listing 4.7 shows a refactored version that will render XML when it's requested.

Listing 4.7 A refactored index action version that will render XML when it's requested

```
def index
  @option_pages, @options = paginate :options, :per_page => 10

  respond_to do |format|
    format.html
    format.xml { render :xml => @options }
  end
end
<#1 Open respond_to block>
<#2 Allow XML requests>
```

The method functions the same except, that now we have this `respond_to` block #1. Inside, we have a specification of the request types that this method can respond to. In this case, we have HTML and XML #2. In the case of HTML, we'll just render the template `index.html.erb` since there is no explicit render here, but in the case of XML, we'll render an XML version of options (which is produced by calling `to_xml` on it).

TEMPLATE NAMING

Rails template naming isn't completely random; `index.html.erb` tells us that this template is for HTML requests and will be rendered by Embedded Ruby/Erubis. You could just as easily have `index.xml.erb` or `index.js.erb` if you wanted.

So, if we apply this `respond_to` block to all of our methods, our controller should look like Listing 4.8.

Listing 4.8 Applying `respond_to` block to methods

```
class OptionsController < ApplicationController
  def index
    @option_pages, @options = paginate :options, :per_page => 10

    respond_to do |format|
      format.html
      format.xml { render :xml => @options }
    end
  end

  def show
    @option = Option.find(params[:id])
  end
end
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```

    respond_to do |format|
      format.html
      format.xml { render :xml => @option }      #1
    end
  end

  def new
    @option = Option.new

    respond_to do |format|
      format.html
      format.xml { render :xml => @option }
    end
  end

  def edit
    @option = Option.find(params[:id])
  end

  def create
    @option = Option.new(params[:option])

    respond_to do |format|
      if @option.save
        flash[:notice] = 'Option was successfully created.'
        format.html { redirect_to(@option) }
        format.xml { render :xml => @option, :status => :created, :location =>
@option }      #2
      else
        format.html { render :action => "new" }
        format.xml { render :xml => @option.errors, :status =>
:unprocessable_entity }      #3
      end
    end
  end

  def update
    @option = Option.find(params[:id])

    respond_to do |format|
      if @option.update_attributes(params[:option])
        flash[:notice] = 'Option was successfully updated.'
        format.html { redirect_to(@option) }
        format.xml { head :ok }      #4
      else
        format.html { render :action => "edit" }
        format.xml { render :xml => @option.errors, :status =>
:unprocessable_entity }
      end
    end
  end

  def destroy
    @option = Option.find(params[:id])
  end

```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```

@option.destroy

respond_to do |format|
  format.html { redirect_to(options_url) }
  format.xml { head :ok }
end
end
end

```

The XML rendering can take a few different shapes depending on what's being rendered; if you're rendering a single resource (#1), it's basically the same (i.e., `render :xml => object`). For creating a resource, though, there's a little more work involved to satisfy REST clients #2. First, the status code needs to be 201 Created rather than 200 OK (like we use at #4). This tells the client that the request resulted in a new resource being created. Second, the `:location` parameter tells Rails to redirect the request after it's completed, in this case to the `show` action for the newly created resource. If there is an error, the client needs to be notified. Since the client may or may not be a web browser, we need to utilize HTTP as much as possible to tell them that there was a problem. Using the 422 Unprocessable Entity header to indicate an error, we return the error array as XML to the client.

Discussion

Rails doesn't just support HTML and XML as the only valid request types; it actually has built-in support for plaintext, JavaScript/JSON, ICS, RSS, ATOM, and YAML. Convenience methods like `to_json` and `to_yaml` are provided to help you render your response easier. You can also define your own response type; for example, if you wanted to use RDF, you could enable that by adding something like the following to `environment.rb`.

```
Mime::Type.register "application/rdf+xml", :rdf
```

Now you should be able to do `format.rdf` and render your RDF response in the block.

TO SAVE YOU SOME EFFORT...

Now that you've seen the "long" way to do things, check out a Rails plugin named `make_resourceful`. This plugin will generate much of the code for RESTful programming for you (as a testament, Jeremy recently used it in a project and it reduced many of his controllers from around 120 lines to just 5 lines!). We didn't show you this plugin from the start because we wanted you to know what was going on behind the scenes rather than employing cargo cult programming and taking the easy way out!

4.3 Rails performance

One of the sticking points for some developers when it comes to Rails is the performance; of course, performance is a rather relative term when you have so many moving parts in a framework: "the database performance," "the rendering performance," "the dispatch performance," and so on. Any or all of these parts can play a part in your performance problems, but you can't fix the problem until you know what it is, right? In this section, we'll look at how to profile a Rails application and find out your problem spots.

The simplest way to do it is to take a look in your log files. When you see something like the following, you can pretty much guarantee that's a problem area.

```
Processing HomeController#home (for 127.0.0.1 at 2007-08-31 10:15:30) [GET]
Session ID: 39ba59dc6b7a6eb672cf3a0e89bdc75d
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```
Parameters: {"action"=>"home", "controller"=>"home"}
Rendering home/home
Completed in 26.39205 (1 reqs/sec) | Rendering: 0.09310 (1%) | 200 OK
[http://localhost/home/home]
```

If you don't see anything that apparent or there are a lot of code branches in that method, you'll have to dig a little deeper into the workings of your code. Rails provides a benchmarking helper just for this purpose, which is a really handy tool for doing simple analysis of pain points in your code.

4.3.1 Benchmarking a Rails application

Benchmarking is a really easy way to get a lot of data about the performance of your Rails application. Rails offers two ways of getting benchmark data. The first is to use `script/performance/benchmark`, which takes a line of code on the command line, benchmarks it, and report back. For example, let's say that you thought your `Order` model's `cancel` method was getting out of hand. You could do something like the following to find out.

```
$ script/performance/benchmark 'Order.find(:first).cancel!'
```

As a result, the script will give you a simple report of the measurements it took.

```
user    system    total    real
#1      0.000000    0.000000  0.000000 ( 0.017904)
```

Of course, if you had actually problematic code, it would return much larger numbers. This is great for testing things outside the context of your application, but Rails also offers a simple helper for benchmarking inside your application.

Problem

You need to benchmark your Rails inside your application.

Solution

Rails offers a single method named `benchmark` that allows you to benchmark blocks of code throughout your application. For example, let's say you had a model method that looked like Listing 4.1.

```
def create_council_members(members)
  members.each do |member|
    new_member = Member.create(member[:attributes])

    new_member.association = Club.find_by_associate(member[:friend])
    new_member.unverified! if new_member.association.unverified?
  end
end
```

You were having trouble telling which part of that code was causing the problem, but you know it's in this method because of the log files. If you were to wrap that segment of code into a benchmark block, you could get a better idea of what's going on.

```
def create_council_members(members)
  members.each do |member|
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```

Member.benchmark("Creating a member") do
  new_member = Member.create(member[:attributes])
end

Club.benchmark("Associating a member") do
  new_member.association = Club.find_by_associate(member[:friend])
  new_member.unverified! if new_member.association.unverified?
end
end
end

```

If you were to make a few requests and look inside the log file, you should see something like the following snippet.

```

Creating a member (0.34019)
Associating a member (0.90927)
[...]
Creating a member (0.29010)
Associating a member (0.90902)
[...]
Creating a member (0.35069)
Associating a member (1.0937)

```

The time measurements are in parentheses and are usually quite telling (in this case, associating a member takes a lot longer!).

Discussion

Benchmarking models is great, but if you find yourself seeing high render times in your logs, you can benchmark blocks of code in views also in a similar manner. Views also have a benchmark method, as you can see in the following code.

```

<% benchmark "Content tree menu" do %>
  <%= build_content_tree %>
<% end %>

```

This helper is great if you're rendering a menus or complex trees of content and can't seem to nail down the performance problem. If you find you need more data than either of these provides, then perhaps you need to look into profiling.

4.3.2 Profiling a Rails application

Profiling offers runtime analysis of your Rails application, breaking down each method call and showing its share of the execution time. Profiling is a really useful way to get some really in-depth information on your Ruby code in general, but it's usually a little less friendly than the way Rails provides.

Problem

You are having performance issues with a Rails application and would like to find out your application's trouble spots.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

Solution

Rails offers a script that lives in `performance` named `profiler` which will use the Ruby profiler to profile your Rails application. You invoke it in basically the same manner than you invoke the benchmarker. For example, let's say you wanted to profile the confirmation of reservations in your hotel booking application. The following would invoke the profiler with your `confirm!` method.

```
$ script/performance/profiler 'Reservation.confirm!'
```

Rails then produces a report to the console that includes *a lot* of information, some of which is actually useful to you.

Loading Rails...

Using the standard Ruby profiler.

%	cumulative	self	self	total		
time	seconds	seconds	calls	ms/call	ms/call	name
12.20	0.10	0.10	188	0.53	0.64	Array#select
.						
.						
.						
5.12	0.82	0.04	1	1.00	0.98	Reservation#done!
.						
.						
.						
0.00	0.82	0.00	17	0.00	0.00	Proc#new
0.00	0.82	0.00	1	0.00	820.00	#toplevel

We'll say it again: the profiler produces *a lot* of data. A lot of it is interesting (who knew `Array#select` was the worst offender?), a lot of it is useless (of course `#toplevel` doesn't have a performance hit!), and a lot of it is useful (the `done!` method appears to be what's killing the performance here). The important thing to analyze here is not only the percentages and times, but the calls (fourth column). I would worry about `Array#select` if the 12% it takes up wasn't spread across 188 calls; on the other hand, the `done!` method on our model eats up 5% in *one* call. That's where a performance problem lies.

Discussion

If you'd like to get a little more information and fancier reporting on your Rails application, a library called `ruby-prof` can help you out. It's written in C, is much faster, and integrates directly with the Rails profiler script (it will automatically use it if it's available). For example, if you want to generate an HTML report for a graph profile, you can do this:

```
$ script/performance/profiler 'Reservation.confirm!' graph_html
```

It's available as a RubyGem and you can check out its RDoc at <http://ruby-prof.rubyforge.org/>.

4.4 Summary

In this chapter, we looked at a number of techniques that Rails developers should be employing. First, we looked at adding your own extensions to Rails and building reusable components from your own code. We then turned our attention to REST, and refactoring your applications to use it to expose resources to the web.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

Finally, we looked at gauging the performance of your Rails application, with a few practical tools and tips to get you started.

In the next chapter, we'll take a deeper look at Ruby and web services, including another look at REST and REST clients.

5 Web Services

Throughout this book we show you various ways for integrating Ruby with different applications and services. Some scenarios depend on a particular protocol or architecture, whether we're using LDAP, sharing data through a relational database, or moving messages around with WebSphere MQ. In this chapter we'll explore the Web architecture and how we can use Web services to cross language, platform and application boundaries.

Another working title for this chapter could be "Service-Oriented Architecture with Ruby". SOA is not any particular product, technology or protocol, yet to build services that we can reuse in different context and compose them to build larger applications, it helps that all these services are based on common protocols and data formats. So we're going to focus this chapter on HTTP, the underlying protocol of the Web architecture, and the SOAP messaging protocol.

In the first part we'll cover the foundation and show you how to use HTTP and URLs. In the second part we'll venture into the world of RESTful Web services and show you how to handle resources, representations and the uniform interface. In the third part 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. On the Web you will find mostly HTML and various media types like images, music and video. We dealt with HTML when we showed you how to build user-facing applications in Chapter 4. In this chapter we're going to focus on the programmatic Web. The programmatic Web is all about machines talking to machines, application to application integration, and so 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 how to parse the resulting document into structured data that you can process with Ruby. For this particular example we picked CSV. Although most people equate Web services with XML, there's a surprising abundance of information out there that uses structured data but 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

We're building a market intelligence application for which we need to retrieve historical stock prices about various public companies. That information is publicly available on the Web, but we need to retrieve it and parse it into data we can use.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

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`.

USING ALIASING TO EXTEND A METHOD

File processing is common enough in Ruby that it provides an `open` method to each object through the `Kernel` module. You can use this method to open files for reading and writing, and also for spawning processes and exchanging data using pipes. The `open-uri` library extends the `open` method to support the HTTP and FTP protocols. This form of extension is done by aliasing the old method `open` and calling it `open_uri_original_open`, and then defining a new `open` method called that implements HTTP/FTP access, and delegates to the old method for handling files and processes. This practice is common enough that Rails and Facets provide a convenience method called `alias_method_chain`.

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
```

This example is simple enough that we can 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, e-mail (`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 generally it's 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, 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, `open-uri` uses the `extend` method to add meta-data methods to the `String` object it returns. We can use that to access various HTTP headers, for example:

```
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}"
```

EXTENDING OBJECTS VS CLASSES

You're probably familiar with mixins, which allow you to write common behavior using a module and then mix it into your classes using `include`. For example, `Enumerable` is a module you can mix into a class to get the convenience of methods like `map`, `select`, `inject` and `any?/all?`, all you need to do is

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

implement an *each* method in your class. You can also use *extend* to mix the behavior of a module into an object instance, adding all the module's methods to that instance alone. *open-uri* uses that to extend the strings it returns without overloading the String class itself. When you want to add behavior to specific objects, use *extend* instead of *include*.

Not all data on the Web is HTML or XML, and in this particular example we retrieve a CSV document. We're going to use FasterCSV to parse the document. You can learn more about FasterCSV in Chapter 13. For now, let's first install FasterCSV:

```
gem install fastercsv
```

And next, 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 }
```

In a moment 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.

Here's what we get when we run it 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}
```

Listing 5.1 shows the entire program:

Listing 5.1: Get historical stock prices

```
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
```

Discussion

The *open-uri* library is the easiest way to GET content from an HTTP server, it's designed to be simple to use and handle the most common cases.

Besides the example above, there are a few other options worth mentioning.

If you access the Web 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 this example we're going to 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, you can also use a progress bar. Check the *open-uri* documentation for more details.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

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 our network we have an existing service that can receive and process orders. We're not going to build order processing into our Ruby application, but instead using the existing service. We just 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 it easy to use, we're going to support two types of arguments. You can pass the XML document directly, in the form of a string. You can also pass the data as a Hash, and the method will convert it into an XML document.

MORE THAN ONE WAY TO USE XML

Ruby has several libraries for dealing with XML, which one you use depends on the use case. When you need maximum flexibility and need 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 instead. If you have data you need to convert into an XML document, and it looks like a template, you can use Builder instead. Some XML documents map nicely into a hash, in which case XmlSimple is true to its name. There are several other options like Rails's to_xml and atom_feed. Throughout this book we'll show you these libraries by example, always choosing the best library for the task at hand.

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

```
gem install xml-simple
```

And use it 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 setup 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'
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

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) }
```

We're almost ready to make a POST request, but first, let's review a couple more choices, starting with authentication.

Some developers prefer to pass authentication information as a separate method argument, we're going to pass the authentication information in the URL argument. And since we're using HTTPS to encrypt the communication, we'll use HTTP Basic Authentication. The URL will look something like this:

```
send_order 'https://john:secret@example.com/create', order
```

HTTP BASIC AUTHENTICATION

Unfortunately, HTTP Basic Authentication is not user friendly, and so most Web sites prefer to use dedicated login forms and carry authentication information using cookies. That works better for end-users, but when using programmable Web services, HTTP Basic Authentication is the easiest way to authenticate and doesn't complicate clients with custom login forms, or having to carry cookies around in each request. It's also dead simple to use from open-uri, Net::HTTP and Rails 2.0. HTTP Basic Authentication does carry the user name and password in clear text, which are easy to snoop, so we recommend always using it in combination with HTTPS to encrypt all communications over the wire. We also recommend you only store passwords on the server using a salted hash, and Ruby's SHA/MD5 digests are simple enough to use there's no reason not to.

HTTP is a connection-less protocol: it handles each request individually. The server doesn't care if 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
```

In this example we make a single request, so we'll let Net::HTTP open the connection, make the request, and close it without using Keep-Alive:

```
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). The recommended practice is to return 201 or 303.

Most redirect responses, like 301 (Moved Permanently) and 307 (Temporary Redirect) tell us that the service moved to a different URL and our request did not go through, we'll treat those as errors. The 303 (See Other) status tells us the request did go through, there's a new document, and where we can find it, so we'll handle that specific case:

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

Now let's merge all these pieces into a single file, you can see the result in listing 5.2.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

Listing 5.2: Send an order 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::HTTPSeeOther; response['Location']
    when Net::HTTPSuccess; nil
    else response.error!
  end
end

```

And let's see how we can send a new order with two lines of code:

```

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

The Net::HTTP library covers all the features of the HTTP protocol, from one-line requests all the way to Keep-Alive connections, from GET requests to documents and forms, with support for HTTPS and 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 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 two write more convenient libraries. Later on we'll talk about one such library called ActiveResource that provides RESTful access to remote resources, built from Net::HTTP.

5.1.3 Serving HTTP

Now that we know how to access a Web service, let's try and build a simple service. In the previous chapter we talked about Rails, and as you'll see later on in this chapter, Rails is a great framework for building Web

Please post comments or corrections to the Author Online forum at

<http://www.manning-sandbox.com/forum.jspa?forumID=356>

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

Operations set us with an analytics console we can 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 our server we need to setup 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 which also happens to be the default server you use when running Rails application in development mode. For production, though, we recommend Mongrel. Mongrel is faster and more stable than WEBrick, and you can use it as a standalone server, or to deploy a cluster of load-balanced application servers behind Apache or Lighttpd.

REVERSE PROXY APACHE TO MONGREL

One of the more common deployment options for LAMP is running a single front-end Web server using reverse proxy to connect to a cluster of lightweight application server. A common configuration uses Apache as the front-end and a Mongrel cluster as the back-end. The benefit of this architecture is using one central server to handle virtual hosts, URL mapping, access control, and static files, while load-balancing intensive dynamic applications on a cluster of servers, mixing different languages and allowing you to roll updates without brining the entire site down.

In this example we'll use Mongrel as a standalone server, and 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::Handler
  def process(request, response)
    . . .
  end
end
```

Our service only supports GET requests, so we're going to return the status code 405 (Method Not Allowed) for all other HTTP methods:

```
return response.status = 405 unless request.params['REQUEST_METHOD'] == 'GET'
```

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
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

If we get a request we cannot process, we return the 404 (Not Found) status code, but this time we 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, using default compression, and return that file to the client. So we're going to decide on the ZIP file name first, and we'll make sure to use a distinct file name 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 file name or partial file name, 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're going to find all file names that contain the date and end with the extension *log*, for example, "errors-2007-10-05.log".

Once we created the ZIP, we're going to 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 also set three response headers. The first one tells the client the file type, a Web browser can use this to open the file with the right application. The second one tells the client how long the response is, particularly useful for large responses, and showing a progress bar of the download. The third header suggests a file name. Without this header, a request to */logs/last* would attempt to download and save a file called *last*. With this head, the browser will offer you to save the file under a name like *logs-2007-10-05.zip*.

HTTP IS MORE THAN JUST GET AND POST

You probably used Content-Type before, but Content-Disposition? Maybe you used 404 in a sentence, but what is 405? HTTP is a rich protocol and provides a lot of features, most of which are expressed through headers and status codes. It pays to get familiar with HTTP. You will learn how to improve performance through caching, reliability through idempotent and conditional methods, error reporting through status codes, and security through authentication, encryption (HTTPS) and paying attention to data coming over the wire. Have a read, it's worth your time.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

We're just about done. We have a Mongrel *Handler* 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 along 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, 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'
```

And 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, as you can see in Listing 5.3.

Listing 5.3: A service for packaging log files and serving them as a ZIP

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

class LogService < Mongrel::Handler

  def initialize(path)
    @path = path
  end

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

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```

tmp_file = Tempfile.open(zip_filename)
begin
  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
  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!
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

```

Discussion

We used this example to show you how to set up a simple Web service without going the route of a Web framework, and how you can improve the user experience by paying attention to the features of the HTTP protocol. We showed you how to use the 405 status code to restrict access to a given HTTP method, and how to use the Content-Disposition header to specify the file name. HTTP is a rich protocol that provides a large number of status codes and headers to handle a variety of use cases. We recommend you learn more about HTTP, so you can benefit from and put these features to good use.

Now that we covered the basics of HTTP, we're going to take it one step further and explore the REST style, showing you how to create resources, handle multiple representations and use the uniform interface to build RESTful Web services.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

5.2 REST with Rails

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

REST is an architectural style 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 – we're also thinking of the way large scale systems emerge from smaller services, built independently by different groups of people—distributed in design and in implementation.

When we think of REST, we think of following those very same 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 that by 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 these resources. Applying these same principles we can build Web services that are simple to use by both people and applications, and wire them together to create larger applications.

Problem

We're building a task manager which our employees can use to manage their day to day tasks. We're also planning several applications and workflows that will create and act upon these tasks. How would we design our 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.

Whenever we build two components that we want to connect together via an API, our best shot is to design the component using the API first, and extract the API from that. For example, we typically start out by writing controllers and views, and then decide which methods and attributes we need to add to our controllers in support of that.

Web services pose a different challenge. We don't often know our clients in advance. Maybe we do have one or two clients so start with, but even in that case were planning for more. After all, the more applications we can build later on to reuse our service, the more return we get on our investment. So we're always on the lookout for those principles and conventions that would make our service easy to build and reuse.

In this section we're going to do just that by applying REST principles to our task manager. We'll start by identifying the most important resources we need to provide. We have one 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 request to the right controller. We do that in the *config/routes.rb* file:

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```

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

  And for individual tasks, we're going to use the show action:
# GET on /tasks/{id}
# View: tasks/show.html.erb
def show
  @task = Task.find(params[:id])
end

```

SESSIONS, AUTHENTICATION AND INSTANCE VARIABLES

One common mistakes Web developers do is storing the copy of an object in the session, for example:

```
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 than 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.

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=>'index' }
    format.xml { render :xml=>task }
  end
end

# DELETE on /tasks/{id}

```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```

def destroy
  if task = Task.find_by_id(params[:id])
    task.destroy
  end
  head :ok
end

```

We got a bit ahead of ourselves. Before we could do all these things on 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=>'index' }
    format.xml { render :xml=>@task, :status=>:created,
      :location=>url_for(:action=>'show', :id=>task.id) }
  end
end

```

SEMANTIC HTTP AND USER EXPERIENCE

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 new 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 new resource, and where to find that resource, for example, to update it later on. In both cases, we return a document that may be different from the one we received, for example adding fields like id, version and updated_at. Either way, we're using the full semantics of the HTTP protocol to distinguish between creating a new resource and updating an existing one.

People work different from applications, so when responding to a Web browser, we worry about the user experience. The way browsers work, if we simply responded to a POST request with a render, and the user then decides to refresh the page, the browser will 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 decide to take the user back to the tasks lists.

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 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 new 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 a several fields taking up most of the page, we would want to offer those as 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 we'll keep designing our resources hierarchically, assigning it the URL path /tasks/new. Likewise, we'll associate each individual task with a URL for viewing it and a URL for editing it:

```

# GET on /tasks/new
# View: tasks/new.html.erb

```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```

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 more clear why we choose to lay out the resources hierarchically. If you like tinkering with the browser's address bar, try this: open up 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 1000 words of documentation.

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

GET request to `/tasks` return 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 got here not by accident. We intentionally chose these resources so all we need to do is keep track of 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.

THE UNIFORM INTERFACE AND _METHOD

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 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 a second time. The uniform interface saves us from having to reinvent and document these semantics for each and every application, 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 caught up yet, 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.

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, and so our one-line definition of the resource captures all that logic, 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.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

The first resource we're going to add is for viewing the collection of completed tasks, and 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 clicking on one of these numbers sets the task priority. 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 priority by sending it a new priority number. We consider it part of the task resource, so we'll associate it with the URL path */tasks/{id}/priority*.

So 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
```

And now let's go and add the controller actions:

```
# 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 intend. So let's investigate our route definitions using the *routes* task:

```
$ rake routes
```

The output would 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 narrow pages of this book, by ignoring the controller name (no surprise, it's always "tasks") and removed the formatting routes that we'll cover in the next section. We kept the rest. You can see how each HTTP method (second column) and URL template (third column) map to the right controller action (right-most column). A quick peek tells us all we need to know.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

The left-most column deserves a bit more explanation. Rails creates several friendly looking routing methods that we can use in stead 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, shorten it to:

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

We can place next to each task appearing in the tasks list, pointing to that task's view:

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

Or 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 what the resulting controller looks like.

Listing 5.6: Our RESTful tasks list controller

```
class TasksController < ApplicationController
  before_filter :set_task, :only=>[:show, :edit, :update, :complete]

  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 tasks_url }
      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 tasks_url }
      format.xml { render :xml=>@task }
    end
  end
end
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```

end

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

def destroy
  if task = Task.find_by_id(params[:id])
    task.destroy
  end
  head :ok
end

private

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

```

404, 422 AND RESCUE_FROM

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. We let Rails figure it out.

When we call `Task.find`, it throws an `ActiveRecord::RecordNotFound` exception if it can't find a task with that identifier. 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, for example 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 into a 422 (Unprocessable Entity) status code.

You can add your own logic for catching and dealing with these exceptions, and also introduce your own exception and handling logic. Have a look at `ActionController::Rescue`, in particular the `rescue_from` method.

Discussion

We covered the basics for setting up RESTful resources and friendly URLs using Rails, but we couldn't cover it all, and in the next section we'll extend this example to support multiple output formats, adding XML, JSON and Atom.

Once you get the hang of this, we encourage you to explore more. You can take hierarchical resources one step further and create nested resources, for example, `/books/598/chapters/5`. You can use the `to_param` method to create more friendly URLs, for example, `/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.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

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; 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 pull it into a calendar application in the form of an iCal list of todo 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 we're building our task manager we realize the 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. Before we get to show this by example, let's discuss the various ways we can couple resources and representations.

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 in a format the client understands. If the server doesn't support any of the listed content types, it simply responds with 406 (Not Acceptable).

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, 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, you can send someone a URL to an XML document. 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 true way to construct these URLs, but there are two common conventions. One convention 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, for example, `/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 having a file called `tasks.xml` will always open in the right application.

How does Rails handle it? If we use the built-in mechanism to decide on the content type – and we'll show those in a second – then Rails will pick up the expected format from the query parameter *format*, or if not there from the URL path suffix, or if not there from the *Accept* header. So which one you use is up to you, and the same application will work with all three.

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 only shows an HTML output, but in fact it supports as many formats as the views we've got. When

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

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 will use it to render HTML responses. If we add a view called *index.xml.builder*, Rails will 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 of these: ERB (eRuby), Builder and RJS. This is a new feature introduced in Rails 2.0. Earlier versions were less flexible, and always match a combination of format and templating engine, so for HTML it will default to ERB by looking up the view *index.rhtml*, and for XML it will 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
```

That happens to be the long way to do things, now let's look at the short way to do exactly the same:

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

We're using the *respond_to* method to match each format we support and the logic to render it. It's similar to the *case* statement above, 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 set the *Content-Type* header appropriately. Shorter to write, easier to maintain.

ACCEPTING DIFFERENT REPRESENTATIONS

You'll notice in the previous example, when we wrote an action to create a new task, we did this:

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

Multiple representations work both way. If we can create a response and send back an XML document, we better be able to process a request by accepting the same XML document. When Rails processes an XML

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

request, it converts the XML document into a Hash, using the document element's name for the parameter name. The above 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 convention 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 turn 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, for example:

```
<% 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`); give it an existing record and it will point the form to the URL for updating an existing resource (`task_url(@task)`). That probably explains why we used `Task.new` to render the form in the `new` action.

Rails comes with built-in support for HTML forms, XML, JSON and YAML, and if that's not enough you can always add customer parameter parsers, have a look at `ActionController::Base.param_parsers` for more information.

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

Listing 5.7: 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
```

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.

`AtomFeedBuilder` implements a couple of the methods we use here: `updated` and `entry`. The other methods are provided by its base class, `Builder::XmlMarkup`, which we're going to describe next.

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 also included as part of Rails. Builder is very simple to understand and intuitive to use, and a good example for what can be done with a little bit of meta-programming.

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`, there is no need to specify any of these

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

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 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 like *tag!* to create an element with a given name (for example, to handle special characters or namespaces), *text!*, *cdata!* and *comment!* (they do exactly what you think they would), and *instruct!* to create the XML declaration at the top of the document.

METHOD_MISSING AND BLANKSLATE

Builder is an interesting use of *method_missing*. 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, pass it on to the object's *method_missing*. The default implementation throws *NoMethodError*. Builder 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.

We mentioned before that you can use different URLs for the various representations. When we defined the tasks resource, Rails created several named route methods like *tasks_url* and *task_url*. In addition, an we didn't show that before, it 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 will show up when you run the *rake routes* task.

So let's add a link users can use to subscribe to the Atom feed, using a named route:

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

Discussion

We just showed you how easy it is to return different presentations for the same resource. The combination of RESTful routes and multiple representations make it so easy to build a RESTful API into your existing Web-based UI, there's no reason not to do that.

5.2.3 Using ActiveResource

We started this chapter by showing you how easy it is to use open-uri and Net::HTTP. Well, easy is a relative terms. 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, and 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

With the task manager service up and running it's time to move forward and develop our workflow application, and as part of that application we'll need to create and manage tasks. We want to reuse our task manager service, and we want to get it done before the day is over.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

Solution

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

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

If you remember from section 5.1.2, we're using the URL to specify the username/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 yet to write a single line of code, but let's first see what we can do with our new *Task* class. Let's start by creating a new task:

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

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

Let's go and 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 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 update. 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.

MORE THAN CRUD

What we're showing here is a simple CRUD application, and the critic in us recognizes that we've just created a layer of overhead to expose a database table over HTTP instead of accessing it directly. Unfortunately that's the limit of what we can show in this short chapter, but REST applications extend far beyond that.

A real service would involve more intricate logic and processing behind each action. A fully functional task manager would validate tasks before creating them, assign owners, send notifications, list tasks by priority and completion data, handle delegation and escalation, track and report various performance metrics, alter when the workload is more than we can handle. These are all problems solved by application logic, far beyond the simple mapping of HTTP requests to database access. Yet, even as we build such a complex

Please post comments or corrections to the Author Online forum at

<http://www.manning-sandbox.com/forum.jspa?forumID=356>

service, we can expose most of its functionality through the pattern of create/read/update/delete, and do so elegantly using REST.

How do we know our Task class sends requests to the right URL? We assumed it uses XML by default, 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 pattern, but 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, so let's list those:

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

As you can guess, this just is 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. If you remember from 5.2.1, we're passing a priority number in the body of the message.

As you can 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 ActiveRecord class that represents our task manager service.

Listing 5.8: Using our task manager with ActiveRecord

```
class Task < ActiveRecord::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
```

Now let's try it out:

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

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

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

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

Discussion

The basic point of this section was to show you how easy it is to build RESTful Web services in Ruby. Hopefully we've done more than that. We also wanted to share to other lessons you can take with to other frameworks and programming languages.

First, is the simplicity of REST and why it pays to design Web services in terms of resources. Resources capture a lot of the functionality behind services, as does the ability to reference them (URLs), and use a uniform interface for accessing the resource, changing it and creating new resources. Rails is not the only framework that does REST, but it does have a certain level of simplicity and is straightforward to use, and a great learning tool if you are new to this world.

We also wanted to show by example the benefit of picking up on conventions, and how they can help you design better, develop faster and end up with code that's easier to understand and maintain. Once you pick on the conventions, there's a lot less to document and getting into the code is easier. What we showed is one particular example for a task manager, but you can apply these same principles to any Web service or Web application you build.

The SOAP messaging protocol is another way to harness the Web architecture 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 options we're thinking of is SOAP. And yes, Ruby does come with a SOAP stack called, not surprisingly, SOAP4R.

SOAP4R supports SOAP 1.1, SOAP with attachments, and service definitions specified with WSDL 1.1. Security options includes 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 compliance you can expect it to work with the various Java SOAP stacks (Axis, Glue, XFire), .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 client application to invoke it.

■ 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 on the same task manager service we developed in the previous sections, but this time using SOAP instead of REST.

Problem

Develop a task manager service that we 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,

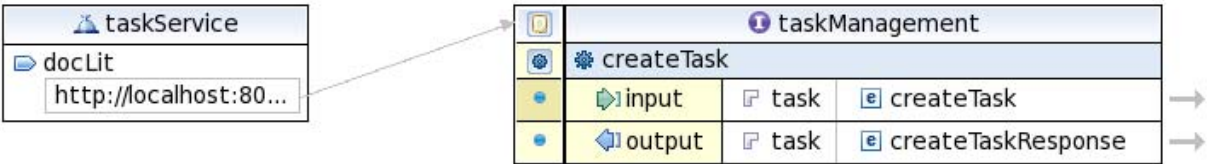
Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

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, now that 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 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 is a simple outline of the service definition, visualized using Eclipse WST. The full WSDL is given in listing 5.9.

Figure 5.1: Simple ta



Listing 5.9: 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>
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```

    <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.

SOAP4R: STANDARD LIBRARY OR GEM?

SOAP4R comes in two flavors. It's part of the Ruby standard library already available in your Ruby installation. It is also available as a packaged Gem that you can install from the public Gem repository at RubyForge.

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 releases updates more frequently, currently at version 1.5.8 and pushing towards 1.6. We recommend you stay up with the latest enhancements and bug fixes by installing and using the SOAP4R Gem in your environment.

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 classes 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 our case), the declared faults (of which we have none) and the document/encoding in use. All that information comes from the WSDL document and 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 parsing the `createTask` element 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.

You may notice 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.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

The third file, `taskServiceServant.rb` contains the actual service skeleton. SOAP4R refers to it as "servant". A servant is simply 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 method `createTask` to create a new task record in the database and return the task identifier. You can see the full service implementation in listing 5.10.

Listing 5.10: 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)
      task = Task.create :title=>request.title, :priority=>request.priority
      return CreateTaskResponse.new(task.id)
    end
  end
end

end
```

Typically, we're going to build a number of services (or servants) and configure 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're going to use the quick prototype standalone 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 5.11).

Listing 5.11: A simple task manager SOAP service

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

# Configure ActiveRecord database connection
confid = YAML.load(File.read('database.yaml'))
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```

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 }
# Start the server
server.start

```

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.yml`, which for our database setup looks like this:

```

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

```

Now let's run it:

```
$ ruby server.rb
```

USING ACTIVERECORD OUTSIDE OF RAILS

ActiveRecord is a key piece of the Rails puzzle and provides the model part of MVC, 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.

In the next section we'll write a client application to use this service, 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 life services.

Our example was simple enough that we wrote the XML Schema type definitions in-line, 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 mappings files, that 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. 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 applications, and 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

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

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.

■ **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 surprising, it will be easier and quicker to do than writing the service itself.

Problem

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 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 5.12.

Listing 5.12: Task manager client using WSDLDriver

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

```
wsdl = File.expand_path('../taskService.wsdl')
driver = SOAP::WSDLDriverFactory.new(wsdl).create_rpc_driver
response = driver.createTask(:title=>'Learn SOAP4R', :priority=>1)
puts "Created task #{response['id']}"
```

\$KCODE AND UTF8

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

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

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 looking at the source code we'll find ready definitions for CreateTask and CreateTaskResponse. The second file, taskServiceMappingRegistry.rb specifies the mapping between XML elements and these Ruby classes. We cover that 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 5.13 shows how we would use the TaskManagement driver with the typed message parts in place of the WSDL driver and hashes.

Listing 5.13: 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}"
```

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 mixing services written in different languages. If you prefer using Ruby MRI, have a look at C-based WS stacks that offer Ruby bindings. One we're aware of are open source Ruby bindings for Axis/C, provided by WS02, the lead developer of Axis.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

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.

Before we complete, there's 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 exists for handling common tasks.

Want to talk to eBay's Web services? eBay4R is the easiest way to get started. How about Amazon on-demand services? Ruby has libraries for using Amazon S3, EC2, SQS and SimpleDB. SAP NetWeaver? Have a look at sapnwrfc, 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.

In fact, head over to the next chapter where we get to talk about asynchronous communication, and show you one example for integrating your internal business applications with an SaaS provider over the Web, using WebSphere MQ and ActiveSalesForce.

6 Automating Communication

With the proliferation of communication technologies like the telephone, the Internet, and cell phones, people are more connected with one another than ever before. Modern technology has greatly increased mankind's ability to not only connect, but to do it more conveniently. No longer do you have to trudge out to a phone booth to make a phone call or to a public terminal in a lab to send an e-mail (if the person you want to contact even has an e-mail address!). These days we can communicate easily with one another from the convenience of our own homes or offices.

Even further, recent software advances have allowed us to automate these communications: electronic call center menus, widespread e-mail, instant messaging bots, and so on. This sort of communications automation software can make your life much easier when it comes to handling interactions with co-workers, employees, and customers. Do you want to send out an e-mail to all 500 customers individually? Or is your hobby to monitor your sales system to inform people who buy your product that their product has shipped? As invisible and minor as this software seems, it really is a very big piece of the infrastructure of modern businesses.

In this chapter, we'll look at a few techniques for creating this sort of software, looking at code examples extracted from real system doing this sort of work every day. We'll start out by looking at e-mail, discussing how to send, receive, and process it. Then, we'll take a look at some instant communication mediums, such as AOL Instant Messenger and Jabber.

6.1 Automating E-mail

E-mail is one of the most popular technologies on the Web today, but thanks to spammers, many users have a bad connotation of the concept of "automated e-mail dispersion" even though this sort of mechanism is probably one of the most common in Internet applications. It's commonly used in consumer applications: Purchased a product from an online shop? Chances are you've received an automatically generated e-mail message. Signed up for a new service lately? You've probably received an activation notice via e-mail.

But these are all business-to-consumer use cases. What about automating e-mail like this for things like your continuous integration server or process monitoring software? You can use e-mail as a powerful way to alert or update people on the state of a system. You could also use automated e-mail reception for tasks like creating new tickets in your development ticketing system or setting up autoresponders for e-mail addresses (e.g., a user sends an e-mail to `autoresponse@yourdomain.com`, and then e-mails are automatically responded to with the message in the body for that address). E-mails ubiquity makes it a powerful tool for communication in your Ruby applications, and in this section we'll show you how to harness e-mail to do your (Ruby-powered) bidding. We'll cover the basic like sending and receiving, then processing e-mail with Ruby. Let's take a look at your options for sending e-mail messages with Ruby.

6.1.1 Automating sending e-mail

Ruby has a few options for sending e-mail messages. First, there's a built-in library, `Net::SMTP`, which is very flexible but also very difficult to use compared with others. There are also other high-level solutions such as TMail or Action Mailer (built on top of TMail), which work just fine, but we've found that a gem named MailFactory works the best. MailFactory gives you nice facilities for creating e-mail messages while requiring very little in the way of dependencies.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

Testing SMTP

If you don't have an SMTP server like Sendmail handy, then Mailtrap is for you. Written by Matt Mowers, Mailtrap is a "fake" SMTP server that will write (or "trap") the messages to a text file rather than send them. You can grab it via RubyGems (`gem install mailtrap`) or download it and find out more at <http://rubyforge.org/projects/rubymatt/>.

Problem

You need to automate sending e-mail from your Ruby application to alert your system administrators that Apache crashed.

Solution

Ruby's built in SMTP library is fairly low-level, at least in the sense that it makes you feed it a properly formatted SMTP message rather than building it for you. MailFactory is a library (available via RubyGems) that helps you generate a properly formatted message, which you can then use to send via the built-in SMTP library.

Creating a message with MailFactory is as simple as creating a `MailFactory` object, setting the proper attributes, and then using the getting the generated message from the object. See a short example in Listing 6.1.

Listing 6.1: Constructing a basic MailFactory object attribute by attribute

```
mail = MailFactory.new
mail.text = "This is the message!"
mail.subject = "Re: Ruby in Practice"
mail.from = "Jeremy <jeremy@rubyinpractice.com>"
mail.replyto = "Assaf <assaf@rubyinpractice.com>"
mail.to = "You <you@yourdomain.com>"
```

```
mail.to_s
# => [Your properly formatted SMTP message]
```

As you can see, setting up a `MailFactory` object is fairly straightforward: instantiate, populate, and output the message to a string.

SHORTCUT

You can also use the `MailFactory#construct` method and a `Hash` of options rather than this verbose method to build a mail message if you'd like to populate the message from values from elsewhere or if you simply want to save a few keystrokes.

Now, all you need to do is feed the message to `Net::SMTP` to send it. For example, let's say you wanted to send an e-mail to your system administration team every time your Apache web server process crashes. Fundamentally the process to do that is to build a `MailFactory` object, giving it a string of recipients, and then send it via `Net::SMTP`. See Listing 6.2 for our implementation.

Listing 6.2: Sending e-mail to our administrators

```
require 'rubygems'
require 'mailfactory'
require 'net/smtp'

ADMINS = ['jeremy@yourco.com', 'assaf@yourco.com',      #1
          'jacki@yourco.com', 'joe@yourco.com']         |
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```

# Construct our message once since our list of admins
# and our message is static.
mail = MailFactory.new #2
mail.text = "The Apache process is down on #{Socket.gethostname}. [CA] Please
restart it!"
mail.subject = "Apache is down on #{Socket.gethostname}"
mail.from = "Apache Notifier <notifier@yourco.com>"
mail.to = ADMINS.join(',')

# Is the process running?
while (true) #3
  unless `ps -A`.include?("apache") #4
    puts "*** Process is down"

    Net::SMTP.start("my.smtp.com", 25, "my.smtp.com", "user", [CA] #5
"password", :plain) do |smtp|
      smtp.send_message(mail.to_s, mail.from.first, mail.to)
    end
  end

  sleep 5
end

```

First, we build an array of administrator e-mail addresses (#1). Then we use this and other information to build the **MailFactory** object (#2). Next, we constantly loop like a daemon #3 (we could also take this out and run the script in a **cron** job or something like that), grabbing the output of **ps** #4 and checking it for the term “apache.” If it’s not found, we send a mail to the administrators using **Net::SMTP** (#5). The **start** method takes parameters for the SMTP server address and port, the “from” domain, your username and password, and the authentication scheme (could be **:plain**, **:login**, or **:cram**, which is CRAM MD5). An SMTP object is then yielded to the block, which we can call methods on to send e-mail messages (e.g., **send_message**).

Discussion

We like to use MailFactory to build the SMTP message like this, but it’s not required. If you’re comfortable enough building properly formatted messages or are grabbing the messages from another source, MailFactory isn’t required. There are also alternatives to MailFactory, like TMail, which both generates and parses e-mail messages. You don’t even have to use the built-in library for sending messages; if you’re really masochistic, you could just use a **TCPSocket** and talk SMTP directly!

SMS MESSAGES

A lot of cell phone carriers let you send SMS messages via e-mail; using e-mail to send SMS messages is a cheap and efficient way to reach people instantly when one of the messages discussed later isn’t available.

One thing to note about our example is that it likely won’t work on Windows. The **ps** utility is a *nix specific utility, which means that if you’re on Linux, Solaris, or Mac OS X, you should be fine, but if you’re on Windows, you’re out of luck. If you really want to implement something like this on Windows, then you can take a few routes. One is to use one of the many WMI facilities available, either through the **win32** library or one of the other WMI specific packages. You could also seek out a **ps** alternative on Windows, many of which are available if you just do a web search for them.

If these approaches strike you as too “low level,” then perhaps Action Mailer might be for you; Action Mailer is Ruby on Rails’ e-mail library and offers a lot of niceties that these approaches don’t have. This isn’t a book all about Rails, so we won’t go into Action Mailer here, but if you’re interested, then you can check out a

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

book dedicated to Rails or the Action Mailer documentation at <http://am.rubyonrails.org/>. Now that you're familiar with sending e-mail with Ruby, let's take a look at receiving it.

6.1.2 Receiving e-mail

Ruby has built-in libraries for both POP3 and IMAP reception of messages, but unfortunately, they're not API compatible with one another. In this section, we're going to concentrate on the POP3 library (`Net::POP3`), but if you're interested, the example will be available for the IMAP library in the downloadable source code for this book.

Problem

You need to receive e-mail in your Ruby application.

Solution

Ruby's POP3 library, `Net::POP3`, is fairly simple to operate. To grab the messages from your inbox, you simply need to use the `start` method on the `Net::POP3` class and manipulate the object given to its block. Check out an example in Listing 6.3.

Listing 6.3: Fetching e-mail using POP3

```
require 'net/pop' #1

Net::POP3.start('pop3.myhost.com', 110, #2
  'ruby', 'inpractice') do |pop| |
  if pop.mail.empty? #3
    puts "You don't have any e-mail!"
  else
    puts "#{pop.mail.size} mails available." #4
  end
end
```

Since `Net::POP3` isn't considered a "core" library in Ruby, you have to require it to use it (#1); once you've gotten the library properly in place, you can take a few approaches to getting your mail. You can instantiate an object and work with it, but we think our approach here (using the class method and a block) is cleaner and more concise (#2). The parameters for the `start` method are the connection's credentials: host, port, login, password. Next, we interact with the object yielded to the block to see how many messages are present in the current fetch. If there are no e-mail messages, then we output a message indicating as much #3, but otherwise we output how many messages were found (#4).

Let's say you wanted to setup a system that will restart your MySQL server when you send an e-mail to a specific e-mail address with the subject "Restart MySQL." To build a system like that, you'd need to grab the e-mails from the address's inbox, iterate them, and check each message for "Restart MySQL" to be included in the subject. You can see our implementation in Listing 6.4.

Listing 6.4: Restarting MySQL via e-mail

```
require 'net/pop'

AUTHORIZED = ['jeremy@yourco.com', 'assaf@yourco.com', [CA] #1
  'jeremymcanally@gmail.com']
RESTART = '/etc/init.d/mysql stop; /etc/init.d/mysql start;' #2
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```

while true
  Net::POP3.start('pop.yourhost.com', 110, [CA]
'mysql@yourco.com', 'dbMASTER') do |pop|

    puts "received #{pop.mails.length} messages"           #3
    unless pop.mails.empty?
      pop.mails.each do |mail|                             #4
        msg = mail.pop

        # If the subject has the phrase and they're authorized...
        if (msg =~ /Subject:.*Restart MySQL.*/i) &&         #5
          (AUTHORIZED.include?(msg.match(/From:.*<(.*)>\r\n/)[1]))
            # ...then, restart MySQL
            `#{RESTART}`
          end

          # Delete the mail message
          mail.delete
        end
      end
    end

    # Don't hammer your POP3 server :)
    sleep 30
  end
end

```

First, we setup a couple of constants: one for the e-mail addresses that are authorized to restart MySQL (#1) and one for the command we'll use to restart MySQL (#2). Since we're using a white list of e-mail addresses, we can lose a little less sleep over security. Next we output a message telling how many messages we've received #3; we then iterate the messages (#4), checking for the proper subject and "From" address #5. If the sender is authorized and the subject contains "Restart MySQL," then we run **RESTART** and MySQL is restarted. Regardless of what happens, the mail message is deleted and the thread sleeps for 30 seconds.

Discussion

The production system that this solution is based on had a few more things that administrators could do via e-mail, such as managing indexes and creating databases. It also allowed administrators to send multiple commands per e-mail. But we decided to strip this solution down to give you a base to work from that you can expand or completely change to your whim. You could change this to manage other long running processes, execute one-off jobs, or even to send other e-mails out.

POP3 is usually good enough for most instances, but on some networks APOP (i.e., Authenticated POP) is required. If your host uses APOP authentication, then you can give the **start** a fifth boolean parameter to indicate that **Net::POP3** should use APOP. So, if we wanted to enable APOP on our previous example, the call to the **start** method would look something like the following:

```

Net::POP3.start('pop.yourhost.com', 110, 'mysql@yourco.com', [CA]
'dbMASTER', true)

```

In our solution, we used a regular expression to extract information from the raw POP message given to us by the server. This works for simple cases like what we've done here, but as your needs get more complicated,

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

the viability of this approach breaks down. In the next section, we'll take a look at a much more robust solution to e-mail processing: the TMail library.

6.1.3 Processing e-mail

Now that you know how to send and receive e-mail, you can start thinking about how to leverage these techniques to solve bigger problems. In this section, we'll combine these two techniques and take a look at one subsystem in a ticketing system that is in production.

Problem

You need to process and respond to support ticket creation e-mail messages in your Ruby application.

Solution

Let's say you had a ticketing system you had built with Rails. It's running great, but it's a bit laborious for users to have to login, click "New Ticket," type all their stuff in, preview it, and then send it, especially since your users file *a lot* of tickets. So, you get the bright idea to allow them to open tickets via e-mail. What a great idea!

But how should you implement it? The smartest flow seems to be to receive an e-mail, process its contents, stick the relevant data in an instance of your model, then delete the mail. Then, pull the model on the front end with a web interface. So, we'll assume you have a **Ticket** model like the following.

```
class Ticket < ActiveRecord::Base
  has_many :responses
end
```

Now, let's take a look at our implementation of the mail handling script (in Listing 6.5) and analyze it piece by piece.

Listing 6.5: Creating tickets via e-mail

```
require 'net/pop'
require 'tmail'
require 'net/smtp'

while true
  Net::POP3.start('mail.yourhost.com', 110,      #1
    'tickets@yourco.com', 't1xeTz') do |pop| |

    unless pop.mails.empty?
      # Iterate each mail
      pop.mails.each do |mail|                  #2
        # Parse the e-mail we received
        ticket_mail = TMail::Mail.parse(mail.pop) #3

        # Create a new Ticket instance
        new_ticket = Ticket.new                 #4
        new_ticket.owner = ticket_mail.from |
        new_ticket.subject = ticket_mail.subject |
        new_ticket.text = ticket_mail.body |
        new_ticket.save |
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```

# Create and send the new e-mail
new_mail = TMail::Mail.new                                     #5
new_mail.to = ticket_mail.from
new_mail.from = 'Jeremy <jeremy@jeremymcanally.com>'
new_mail.subject = "Ticket Created! :: #{ticket_mail.subject}"
new_mail.date = Time.now
new_mail.mime_version = '1.0'
new_mail.set_content_type 'text', 'plain'
new_mail.body = "A new ticket has been created for [CA]
you.\n=====\\n\\n#{ticket_mail.body}\\n\\nThanks!"

Net::SMTP.start("my.smtp.com", 25, "my.smtp.com", "user", "password",
:plain) do |smtp|
  smtp.send_message(new_mail.encoded, new_mail.from, new_mail.to)
end

# Delete the received mail message
mail.delete
end
end
end

sleep 30
end

```

In this implementation, we first receive our e-mail messages using **Net::POP3** (#1). Then we iterate the messages (#2), and use TMail's message parsing abilities to get a usable object with attributes (#3). We then create a new instance of our ActiveRecord model, **Ticket**, and populate it with the data from the e-mail. Then we use TMail to build a new e-mail object (notice the API similarities to MailFactory) #5, and send that e-mail using **Net::SMTP**.

Discussion

Until very recently, TMail was rarely updated since its inclusion in Ruby on Rails. Hopefully releases will continue to come to improve support for multipart messages and to improve security, but the future of the library is still very much in question. While we recommend you use it to parse messages, we've found that the well-maintained MailFactory gem works better for mail generation.

Now that you have a solid grasp of automating e-mail, let's take a look at another problem domain in communication automation: instant messaging.

6.2 Automating instant communication

Sometimes, e-mail just isn't quick enough. Thanks to technologies like online chat and instant messaging, we can now be connected directly with one another, chatting instantly. And the ubiquity of technologies like AOL Instant Messenger, Jabber, and others finally make them a viable solution for business communication. Automating these sorts of communications open up some interesting possibilities: customer service bots, instant notification from your continuous integration system, and so on. This section will concentrate on using two of the most popular options for instant communication: AOL Instant Messenger (AIM) and Jabber.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

6.2.1 Sending messages with AIM

Once released independently of the America Online dial-up client, the Instant Messaging component of the AOL system quickly became one of the most popular systems for private messaging. Still today, it's the de facto standard for instant messaging with over 50% market share. Contacting users or employees through means they already use is a good way to make sure your communication is heard as quickly as possible.

Problem

You need to send server information via instant messages using AIM.

Solution

The `Net::TOC` library provides a very flexible API for interacting with the AIM service. The first approach you can take to using it is a simple, procedural connect/send/disconnect approach. See an example in Listing 6.6.

Listing 6.6: Sending an IM with Net::TOC

```
require 'rubygems'
require 'net/toc'

client = Net::TOC.new("yourbot", "p@$w0rd")      #1
client.connect                                   |

friend = client.buddy_list.buddy_named("youraimuser") #2
friend.send_im "Hello, from Ruby."               |

client.disconnect
```

First, we create an object and connect to the AIM service (#1). Then we find a user (in this case, “youraimuser”), send them a simple message, and disconnect from AIM. This approach works well for most cases where you’re simply sending messages, but it gets awkward when you try to deal with incoming messages.

Fortunately, `Net::TOC` has a nice callback mechanism that allows you to respond to events pretty easily. These events range from when an IM is received to when a user becomes available. See Table 6.1 for a full listing.

Table 6.1: A full listing of the Net::TOC callbacks

<code>on_error { err }</code>	Replaces the error handler; calls the block every time an error is encountered.
<code>on_im { message, buddy, auto_response }</code>	Called when an IM is received. Parameters include IM message and sender.
<code>friend.on_status(status) { }</code>	Called when the given friend's status changes. The status parameter should be one of the following: <code>:available</code>, <code>:online</code>, <code>:idle</code>, <code>:away</code>.

These callbacks make interactions with users much cleaner than trying to clumsily shoehorn them into the sequential method. For example, let’s say you wanted to get some information from a server simply by sending an IM to an AIM bot. Listing 6.7 shows an implementation using `Net::TOC`’s callbacks.

Listing 6.7: Sending the results of `uptime` over AIM

```
require 'rubygems'
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```

require 'net/toc'

def get_server_information                                     #1
  """
  uptime
  -----
  #{`uptime`}

  disk information
  -----
  #{`df -k`}\\n
  """
end

client = Net::TOC.new("youruser", "pa$zWu2d")

client.on_im do | message, buddy |                          #2
  friend = client.buddy_list.buddy_named(buddy.screen_name)
  friend.send_im(get_server_information)
end

client.connect
client.wait                                                  #3

```

To get the server information, we create a `get_server_information` method #1. Next, we use the `on_im` callback to respond to any IM's we receive (#2). The callbacks basically function as a declarative way to define behavior when something happens; as you can see here, the block we provide will be called when an IM is received. To send an IM, the buddy is found (to get a `Net::TOC::Buddy` object) and then an IM is sent via the `send_im` method. Once the callback is setup, we connect and wait for IM's to come in to fire the callback.

Discussion

Little bots and automations like this are becoming more and more popular. Developers are beginning to realize the potential uses for them: information lookups, customer management, workflows, and so on. Many IRC channels for open source packages (including Ruby on Rails) now have IRC bots that will give you access to a project's API by simply sending a message like "api `ActionController#render`." Developers looking for a solution to handle peer and management approval in code reviews or to alert their co-workers of Subversion activity could use an AIM bot like this one.

If you're interested in embedding AIM chat features in a Rails application, your options are slim and aren't very "slick," but it is possible. We're not aware of any Rails plugins that currently handle AIM communications dependably. The best way we've found to handle this has been to build an external daemon that you integrate with your Rails application. For example, you could use the `daemons` gem to generate a daemon script, give it full access to the Rails environment by including `environment.rb`, and then run it along with your Rails application. This will allow your daemon to have access to your application's models, making integration a snap.

If you find that the people you need to communicate with don't like AIM, then you can use `libpurpl`, the library that powers Pidgin, a multi-protocol chat client. There is a Ruby gem named `ruburple` that hooks into `libpurple`, but many people seem to experience sporadic success with building it. If you're able to build it, it's a great way to access a number of chat protocols easily. If that doesn't work for you, you can also use XMPP and Jabber to access other chat protocols.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

6.2.2 Automating Jabber

Jabber is an open-source instant messaging platform. The great thing about Jabber is that you can have your own private Jabber server, which you can keep private or link with other Jabber servers. So you can have your own private IM network or be part of the public “Jabbersphere.” In addition, XMPP (the protocol that Jabber runs on) offers a nice set of security features (via SASL and TLS). XMPP servers can also bridge to other transports like AIM and Yahoo! IM. You can learn more about setting up and maintaining your own Jabber server from the Jabber website at <http://www.jabber.org/>.

Problem

You want your administrators to be able to manage MySQL via Jabber messages sent from your Ruby application.

Solution

Ruby has a number of Jabber libraries, but the most advanced and best maintained is `xmpp4r`. In this section, we’ll look at using a library built on top of `xmpp4r` named `Jabber::Simple`, which simplifies the development of Jabber clients in Ruby. You’ll need to install both gems (`xmpp4r` and `xmpp4r-simple`) to use these examples.

Jabber and Rails

If you’re interested in using Jabber with Rails, take a look at Action Messenger, which is a framework like Action Mailer but for instant messaging rather than e-mail. You can visit the Action Messenger home page at <http://trypticon.org/software/actionmessenger/>.

The process for interacting with Jabber is very similar to the process for interacting with AIM, but the API exposed in `Jabber::Simple` is slightly, well, simpler. Take a look at Listing 6.8 for an example.

Listing 6.8: Building a simple `Jabber::Simple` object

```
require 'rubygems'
require 'xmpp4r'
require 'xmpp4r-simple'

im = Jabber::Simple.new("you@jabberserver.com", "p@s$")
im.roster.add("them@jabberserver.com")
im.deliver("them@jabberserver.com", "Hello from Ruby!")
```

First, we create a new `Jabber::Simple` object. When creating this object, you must provide your login credentials, and the account will be logged in. The new object is essentially an XMPP session with a nice API on top of it. The roster attribute provides an API to the logged in account’s contact list, which allows you to add and remove people. Finally, we use the deliver method to send a message to the person we added to this account’s contact list.

CONTACT LIST AUTHORIZATION

When you add someone to an account’s contact list, the person being added will have to authorize the addition of their account to your contact list. You can use the `subscribe_requests` method to authorize requests for adding your bot. See the `Jabber::Simple` documentation for more information.

`Jabber::Simple` doesn’t implement anything akin to the callbacks in `Net::TOC`, but the mechanism for receiving messages is fairly straight forward. As an example, let’s say you wanted to expand on our earlier

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

MySQL control service to allow your administrators to give the server instructions to stop, start, or restart the server over IM. Listing 6.9 has one implementation of this script.

Listing 6.9: Managing MySQL via Jabber rather than e-mail

```
require 'rubygems'
require 'xmpp4r'
require 'xmpp4r-simple'

AUTHORIZED=['admin@yourco.com', 'admin2@yourco.com']      #1
|
COMMANDS = {
  "start" => "/etc/init.d/mysql start",
  "stop" => "/etc/init.d/mysql stop",
  "restart" => "/etc/init.d/mysql restart"
}
|

im = Jabber::Simple.new("user@yourxmpp.com", "%aZ$w0rDd")  #2

puts "Waiting on messages."
while true
  im.received_messages do |message|                        #3
    # Build a usable user ID from the message
    from = "#{message.from.node}@#{message.from.domain}"  #4

    # Are they authorized?
    if AUTHORIZED.include?(from)
      puts "Received message from #{message.from}"

      if COMMANDS[message.body]                            #5
        `#{COMMANDS[message.body]}`
        im.deliver from, "MySQL told to #{message.body}."
      else
        im.deliver from, "Invalid command!"
      end
    end
  end
end

# Don't want to eat up more CPU than necessary...
sleep 3
end
```

We start off by defining a few constants (#1): **AUTHORIZED** is an **Array** of Jabber users that we permit to issue commands and **COMMANDS** is a **Hash** of command sequences we'll use to control the MySQL server. Next we create our **Jabber::Simple** object (#2), and call the **received_messages** method (#3). This method gives us an iterator that will yield each message received since the last **received_messages** call.

Now we need to figure out who sent us the message; the **message.from** attribute is actually a **Jabber::JID** object that gives us access to some of the internal Jabber data. Since we just need to know their user name and domain, we extract that and build a usable string #4. Next, we check to see if the user who sent us the message (now in **from**) is authorized to be doing so, and if so, we try to issue the command #5. If they sent a bad command (i.e., not "start," "stop," or "restart"), then we tell them so. Otherwise, we go to the next message or begin listening for new messages again.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

Discussion

The `Jabber::Simple` library is nice, but if you like to get down to the bare metal, you could use `xmpp4r` directly. It offers a higher-level API (not quite as high as `Jabber::Simple`, but tolerable), but it also gives you access to much of the underlying mechanics. This access could be useful if you're building custom extensions to XMPP or want to do some sort of filtering on the traffics.

6.3 Summary

As computing power has increased, the ability to automate more and more of our lives has become a reality. We've taken a look at a few approaches to communication automation in this chapter. E-mail automation has been in use for years in certain arenas, but we are seeing it expand out into more business applications (some of which were discussed here) and into the consumer world (with things like Highrise from 37signals and Twitter). AIM bots have been around for years (Jeremy can remember writing one in 1997!), but they've moved out of the "13 year olds tinkering with Visual Basic" and "let's spam people with our ads" arenas and more into "let's use this as a tool in our business" arena. Voice over IP seems to be moving in that same direction with technologies like Asterisk and tools like Adhearsion.

All this is to say that we are beginning to see people use existing communication in new and different ways. Fortunately, the Ruby community is constantly building new tools to work with these technologies, and many of them, like Adhearsion, are pioneers in their field. Another area where Ruby has pioneered is databases, where ActiveRecord and some of the new ORM libraries are pushing the boundaries of DSL usage in database programming.

7 Asynchronous Messaging

Asynchronous messaging is one of the most effective ways to build a loosely coupled system while maintaining a sense of coherence in the system. Coupling the publish/subscribe architecture with a reliable delivery broker gives you a solid architecture for integration with heterogeneous (or homogeneous) systems. In this chapter we'll look at a few options for asynchronous messaging in Ruby, including how to integrate these solutions with Rails. First, we'll look at WebSphere MQ, the grand daddy of Message Oriented Middleware, then proceed to look at lightweight options for those environments that don't need a heavy duty, enterprise grade messaging solution.

7.1 WebSphere MQ

What happens when you combine a powerful message-oriented middleware with a powerful programming language? In the next two sections we're going to look at RubyWMQ, a library for using WebSphere MQ from Ruby.

The best thing about RubyWMQ is how it's implemented. It's essentially a thin wrapper around the WebSphere MQ client API, which means you get access to all the WMQ options and flags, get full control over message descriptors and headers. The get and put methods work exactly like you expect them to, so does browsing and retrieving related messages, managing queues and running WQM commands. If you know WebSphere MQ, there's little new to learn here and you'll be up and running in a few minutes.

Yet, it also brings with it all the useful Ruby idioms, like using hashed arguments, blocks and the each method. As you'll see in the following example, it's easy to get productive with RubyWMQ, and write code that's simple, readable and reliable. So let's get started.

7.1.1 Queuing messages

We're going to use WebSphere MQ for integration, and start by using Ruby as the message producer. In our environment, we have several applications that invite people to register and create new accounts. We're interested in capturing all that information and handling it, so we're going to use WebSphere MQ as a message bus across all these applications. We decided on a common format (using XML) that all our applications will use and a queue to collect all these messages. Now let's put it to use.

Problem

Our Web application allows users to register and create new accounts. Whenever a user creates a new account, we need to capture some of that account information, create an XML message, and push that message into the `ACCOUNTS.CREATED` queue.

Solution

Since we covered Rails in a previous chapter, we're going to use it again in this chapter. We're doing that so we can focus on RubyWMQ, not the details of building a Web application, but obviously anything you learn here is not specific to Rails or Web applications for that matter.

Installing Gems with C extensions

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

RubyWMQ is Ruby code mixed with a native C library. There are other Gems that mix Ruby code with C extensions, some of which we cover in this book, MySQL, Mongrel, Hpricot and RMagick.

If you're using an older version of RubyGems (0.9.4 or 0.9.5) to install any of these Gems, you will be asked to pick a platform. When running on Windows, pick "mswin32" to install pre-compiled DLLs, otherwise pick "ruby" to compile and install a shared library for your specific architecture. We recommend upgrading to RubyGems 1.0.1 or later, which will handle the right platform dependencies for you.

If you're curious what C extensions look like, check out the Gem's `ext/` directory. Typically Gems that use C extensions include an `extconf.rb` file which uses the `mkmf` library to generate the Makefile. The rest is handled by `make` and `gcc`.

You will need to have a development environment that includes these build tools. Some installations include them by default, but if yours doesn't you will have to install them separately. For example, if you're using Ubuntu you can install the development tools and header files using:

```
sudo apt-get install build-essential
sudo apt-get install ruby1.8-dev
```

We talk more about installation and setting up your environment in Appendix A.

Let's first have a look at our existing `AccountsController`. It has several actions, we're only interested in the `create` action that responds to `POST` requests and creates new accounts. Listing 7.1 shows what this action looks like:

Listing 7.1: AccountsController action for creating a new account

```
def create
  @account = Account.new(params['account'])
  if @account.save
    # Created, send user back to main page.
    redirect_to root_url
  else
    # Error, show the registration form with error message
    render :action=>'new'
  end
end
```

Right after we create the new account record and before we respond to the browser, we're going to send a new message using RubyWMQ. Let's keep that code separate and add a new method called `wmq_account_created`.

To start with, this method will need to create an XML document with the new account information. The actual account record contains more information than we want to share, like the user's password and the record's primary key, so we'll be selective with the few attributes we want to share:

```
attributes = account.attributes.slice('first_name', 'last_name', 'company',
'email' }
```

There's another attribute we want to share: the name of the application that created this account. Since we're very Web savvy, we always refer to applications by their domain name, so we'll use the domain name to set the value of this attribute:

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```
attributes.update(:application=>request.host)
```

Now, let's turn these attributes into an XML document with the root element 'account':

```
xml = attributes.to_xml(:root=>'account')
```

A typical message will look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<account>
  <first-name>Assaf</first-name>
  <last-name>Arkin</last-name>
  <company>Ruby in Practice, Inc</company>
  <email>assaf@labnotes.org</email>
  <application>wmq-rails.example.com</application>
</account>
```

Next, we're going to connect to the queue manager. We want to keep the connection configuration in a separate file, so we don't have to dig through source code whenever we need to change it.

In fact, we want to keep several connection configurations that we can use in different environments. Typically we'll use one queue manager that we run locally for development, another one for testing, and the real deal in production. We'll even configure those differently, for example, use non-persistent queues for development and testing. If you worked with Rails before you know that it uses a YAML file to specify the database connection configuration, one for each environment. We'll use a similar structure to configure RubyWMQ. Listing 7.2 shows a sample **wmq.yaml** configuration file.

Listing 7.2: config/wmq.yaml configuration file

```
development:
  q_mgr_name: DEV.QM

test:
  q_mgr_name: TEST.QM

production:
  q_mgr_name: VENUS.QM
  connection_name: central.example.com(1414)
  channel_name: SYSTEM.DEF.SVRCONN
  transport_type: WMQ::MQXPT_TCP
```

Loading YAML files is a trivial matter (head over to Chapter 11 for a longer discussion about YAML and configuration), we just need to pick the right configuration based on our current environment:

```
config_file = File.expand_path('config/wmq.yaml', RAILS_ROOT)
config = YAML.load(File.read(config_file))[RAILS_ENV].symbolize_keys
WMQ::QueueManager.connect(config) do |qmgr|
  . . .
end
```

Now that we have an XML document and an open connection it's a simple matter of putting the message in the right queue:

```
qmgr.put :q_name=>'ACCOUNTS.CREATED', :data=>xml
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

One thing we found helpful during development is keeping a trace of all the messages created by the application. It helps when we need to troubleshoot and look for lost messages. So let's expand this single line to create a `WMQ::Message` object. The message identifier is set when we put the message in the queue, so we can log it afterwards:

```
message = WMQ::Message.new
message.data = xml
qmgr.put :q_name=>'ACCOUNTS.CREATED', :message=>message
logger.info "WMQ.put: message #{message.descriptor[:msg_id]} in
ACCOUNTS.CREATED"
```

Now let's assemble all these pieces into working code. Since we might use RubyWMQ elsewhere in our application, we'll handle the configuration in a single place that is shared by all controllers. Listing 7.3 shows the `ApplicationController` and the method we added to load the right configuration.

Listing 7.3: app/controllers/application.rb modified to read WMQ configuration

```
require 'wmq'

class ApplicationController < ActionController::Base

  def self.wmq_config
    unless @wmq_config
      wmq_config_file = File.expand_path('config/wmq.yaml', RAILS_ROOT)
      @wmq_config =
        YAML.load(File.read(wmq_config_file))[RAILS_ENV].symbolize_keys
    end
    @wmq_config
  end

end
```

Listing 7.4 shows `AccountsController` with the an action that creates new messages and the `wmq_account_created` method. Controllers expose all their public methods as actions, and since this method is not an action, we'll make it private.

Listing 7.3: AccountsController that queues new accounts in ACCOUNTS.CREATED

```
class AccountsController < ApplicationController

  QUEUE_NAME = 'ACCOUNTS.CREATED'

  def create
    @account = Account.new(params[:account])
    if @account.save
      wmq_account_created @account
      # Created, send user back to main page.
      redirect_to root_url
    else
      # Error, show the registration form with error message
      render :action=>'new'
    end
  end

end
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```

private

  def wmq_account_created(account)
    attributes = account.attributes.slice('first_name', 'last_name',
    'company', 'email' }
    attributes.update(:application=>request.host)
    xml = attributes.to_xml(:root=>'account')

    WMQ::QueueManager.connect(self.class.wmq_config) do |qmgr|
      message = WMQ::Message.new
      message.data = xml
      qmgr.put :q_name=>QUEUE_NAME, :message=>message
      logger.info "WMQ.put: message #{message.descriptor[:msg_id]} in
#{QUEUE_NAME}"
    end
  end

end

```

And we're done. We just changed an existing controller to create XML messages and deliver them to other applications using WebSphere MQ.

Discussion

Writing the code was easy, but in spite of best intents we sometimes make mistake, we can't tell for sure this will work in production. Not without a test case. So let's write a test case to make sure we're sending the right information on the right queue. We test the controller in Listing 7.4.

Listing 7.4: Test case for putting message in ACCOUNTS.CREATED

```

module Test::Unit::WMQTest

  # Read the WMQ configuration for this environment.
  def wqm_config
    @wmq_config ||=
      YAML.load(File.read("#{RAILS_ROOT}config/wmq.yaml"))[RAILS_ENV].symbolize_keys
  end

  # Retrieve last message from the named queue, assert that it exists
  # and yield it to the block to make more assertions.
  def wmq_check_message(q_name)
    WMQ::QueueManager.connect(wmq_config) do |qmgr|
      qmgr.open_queue(:q_name=>q_name, :mode=>:input) do |queue|
        message = WMQ::Message.new
        assert queue.get(:message=>message)
        yield message if block_given?
      end
    end
  end

  # Empty queues at the end of the test.
  def wqm_empty_queues(*q_names)
    WMQ::QueueManager.connect(wmq_config) do |qmgr|
      q_names.each do |q_name|

```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```

        qmgr.open_queue(:q_name=>q_name, :mode=>:input) do |queue|
          queue.each { |message| }
        end
      end
    end
  end
end

class AccountsControllerTest < Test::Unit::TestCase
  include Test::Unit::WMQTest

  def setup
    @q_name = 'ACCOUNTS.CREATED'
    @attributes = { 'first_name'=>'John', 'last_name'=>'Smith',
                    'company'=>'ACME', 'email'=>'john@acme.com' }
  end

  def test_wqm_account_created
    post :create, :account=>@attributes
    wqm_check_message @q_name do |message|
      from_xml = Hash.from_xml(message.data)
      assert_equal @account.merge('application'=>'test.host'),
from_xml['account']
    end
  end

  def teardown
    wqm_empty_queues @q_name
  end
end
end

```

The second half of Listing 7.4 is our test case, and it's a very simple one. We **POST** to the create action with new account information, which pushes a new message to the queue. We then ask RubyWMQ to retrieve the new message, assert that the message exists and then yield to the block. In the block we specify another assertion, checking the message content against the expected values.

We may want to use RubyWMQ in more places, so we abstracted the common features into a separate module, **Test::Unit::WMQTest**, that we can mix into other test cases. We have one method to return the queue manager configuration (**wmq_config**), another one to check a message on any given queue (**wmq_check_message**), and a third one we use during teardown (**wqm_empty_queues**).

When test cases fail, we could end up with messages filling up the queue, which would lead to the wrong result the next time we run the test, so during the test tear-down we discard all the messages from the test queue. That explains why we need a separate queue manager for running tests.

RUNNING WMQ COMMANDS

You can also use RubyWMQ to run commands and perform administrative tasks.

For example, if you're sharing this code with other developers, they will need the **ACCOUNTS.CREATED** queue in their development or test queue manager. You can set it up with a simple script or a Rake task:

```
namespace :wmq do
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```

desc 'Creates WebSphere MQ queues on development and test
environments'
task :setup=>:environment do
  wmq_configs =
  YAML.load(File.read(File.expand_path('config/wmq.yaml', RAILS_ROOT)))
  ['development', 'test'].each do |environment|
    config = wmq_configs[environment].symbolize_keys
    WMQ::QueueManager.connect(wmq_config) do |qmgr|
      puts "Creating ACCOUNTS.CREATED queue in
#{wmq_config[:q_mgr_name]}"
      qmgr.mqsc('define qlocal(ACCOUNTS.CREATED) defpsist(no)')
    end
  end
end
end
end

```

So now that we know the basics of producing messages, let's look at another form of integration: consuming and processing messages.

7.1.2 Processing Messages

In this section we're going to use WebSphere MQ for integration by receiving and processing messages. If you recall from the previous section, we have several applications that push account creation messages to a single queue. In this section we're going to turn these messages into sale leads in our CRM.

Problem

We have a number of applications where users can create new accounts. Each of these applications will push a message to the **ACCOUNTS.CREATED** queue. We want to batch process these messages and use them to create sale leads in our CRM database.

Solution

This time around we're looking at batch processing of messages, so we're going to develop a standalone program that we can schedule to run at night. We're going to start by reading the connection configuration, using the same YAML configuration file we specified in the previous section. This time around, we're not running inside Rails, instead we'll pass the environment name using the **ENV** environment variable:

```

ENV = ENV['ENV']
wmq_config = YAML.load(File.read('config/wmq.yaml'))[ENV].symbolize_keys
Next, open a connection to the queue manager and open the queue for reading:
WMQ::QueueManager.connect(wmq_config) do |qmgr|
  qmgr.open_queue(:q_name=>'ACCOUNTS.CREATED', :mode=>:input) do |queue|
    . . .
  end
end

```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

We need to pick up and process each message from the queue. There are a couple of ways for doing that. We can call `get` to retrieve each message and repeat until there are no more messages to receive (`get` returns false). We showed you how to use the `get` method when we covered test cases in the previous section. We're much more fond of using Ruby idioms, so let's do that instead:

```
queue.each(:sync=>true) do |message|  
  . . .  
end
```

The `each` method iterates over every message in the queue and passes it to the block. It also makes our code more reliable. The `:sync=>true` option tells WebSphere MQ to use synchpoints. If the block throws an exception and doesn't catch it, the message goes back to the queue so we can process it again. WebSphere MQ marks the failed delivery attempt and will decide how long to wait before re-delivery and how many failures to tolerate before filing the message in the dead-letter queue. Meanwhile, even if it failed on one message, the loop will continue processing the remaining messages. As simple as it is, this code is fault tolerant and can deal with failures like losing the database connection or tripping over the power cord.

This loop terminates once it's done processing all the messages in the queue, which is exactly what we wanted it to do. Remember, we're running this program on a night schedule to batch process all messages queued during the day. Of course, we could also run this program continuously and have it pick up messages as they become available, blocking to wait for new messages. All we need to do is add the `option :wait=>-1`.

The messages we're processing are XML documents, so this time around we're going to use REXML to parse them. REXML is a standard Ruby library that provides a document API into the XML (we cover REXML in depth in Chapter 11). So let's turn the message data into a Document object:

```
xml = REXML::Document.new(message.data)
```

There are certain elements we're interested in, more specifically, in their contents which holds information like the person's name and e-mail address. We want to map those into a Hash and pass that to the `Lead.create` method. It so happens that the names don't exactly match: `Lead` uses one convention for naming database columns, the XML message a different naming convention. Then there's the application name (XML) that we want to store as the lead source (CRM). What we need here is a transformation. Let's define that:

```
transform = { :first_name=>'first-name',  
              :last_name=>'last-name',  
              :company=>'company',  
              :email=>'email',  
              :lead_source=>'application' }
```

The convention we're using here is to name the target on the left and its source on the right, so `lead_source` comes from (is) application. Doing that, it's easy to iterate over the transformation and map each value from its XML source to its Hash target. We'll use XPath expressions to extract each value:

```
attributes = transform.inject({}) { |hash, (target, source)|  
  nodes = xml.get_text("/account/#{source}")  
  hash.update(target=>nodes.to_s)  
}
```

```
# And create the sale lead record in the database...
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```
Lead.create!(attributes)
```

Where did **Lead** come from? We're using ActiveRecord in this example, so it maps to a record in the database, and since ActiveRecord knows how to extract the class definition from the database schema, all we need to do is specify the class:

```
class Lead < ActiveRecord::Base
end
```

So now let's wrap this example into a single file we can run from the command line. Listing 7.5 is essentially the same code we developed so far, with the addition of error logging.

When you look at Listing 7.5, you may notice this chapter's key lesson. There are a handful of lines dedicated to setting up the environment and to turning an XML document into an ActiveRecord. Yet, what we set up to show you in this chapter – how to easily build programs that can reliably retrieve and process messages – all that semantics are captured in two lines of code: **each** and the following **end**. It really is that simple.

Listing 7.5: Processing messages from WebSphere MQ to create new leads

```
require 'rubygems'
require 'wmq'
require 'active_record'

ENV = ENV['ENV']

# Set up logging and configure the database connection.
LOGGER = Logger.new(STDOUT)
ActiveRecord::Base.logger = LOGGER
database = YAML.load(File.read('config/database.yaml'))[ENV].symbolize_keys
ActiveRecord::Base.establish_connection database

# Define the Lead class.
class Lead < ActiveRecord::Base
end

# Read the WMQ configuration and open a connection.
wmq_config = YAML.load(File.read('config/wmq.yaml'))[ENV].symbolize_keys
WMQ::QueueManager.connect(wmq_config) do |qmgr|
  qmgr.open_queue(:q_name=>'ACCOUNTS.CREATED', :mode=>:input) do |queue|
    queue.each(:sync=>true) do |message|
      begin
        # Parse the document, transform from XML to attributes.
        xml = REXML::Document.new(message.data)
        transform = { :first_name=>'first-name', :last_name=>'last-name',
                      :company=>'company', :email=>'email',
                      :lead_source=>'application' }
        attributes = transform.inject({}) { |hash, (target, source)|
          nodes = xml.get_text("/account/#{source}")
          hash.update(target=>nodes.to_s)
        }
        # Create a new lead.
        lead = Lead.create!(attributes)
        LOGGER.debug "Created new lead #{lead.id}"
      end
    end
  end
end
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```

    rescue Exception=>ex
      LOGGER.error "Failed to create lead: #{ex.message}"
      LOGGER.error ex.backtrace
      # Raise exception, so WMQ keeps the message in the queue.
      raise
    end
  end
end
end
end

```

Discussion

That's really all there is to our solution. We just wrote a simple program that can batch process messages and create new records in the database. It's simple, yet reliable and doesn't get phased when the database goes down, or it encounters an invalid message.

Integrating with WebSphere MQ and Salesforce

Which database are we going to use? Tough call. How about Salesforce? Wouldn't it be interesting if we can turn this example into one that uses Ruby to integrate all our internal applications with Salesforce's SaaS CRM solution?

Let's change out solution to create a new sale lead in Salesforce with these three simple steps.

Step one, install the ActiveSalesForce Gem:

```
$ gem install activesalesforce
```

Step two, to load the ActiveSalesForce connection adapter, add this line as the last require at the top of the file:

```
require 'activesalesforce'
```

Step three, change the database.yml configuration file to use ActiveSalesForce as the connection adapter. Let's do that for the development environment:

```
development:
  adapter: activesalesforce
  url: https://test.salesforce.com
  username: <your username>
  password: <your password>
```

And, we're done! Now when we run this program it will process a day's worth of messages and turn them into sales leads that will show up in your Salesforce account.

7.2 Lighter solutions

If you don't have access to WebSphere MQ or if it's simply too much infrastructure for you, then there are lighter solutions available. In this section, we'll take a look at two of these solutions: ActiveMQ (through the Stomp protocol) and Ruby Reliable Messaging.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

7.2.1 Using ActiveMQ with Stomp

ActiveMQ (<http://activemq.apache.org/>) is a Java-based project from the Apache Software Foundation and is a popular “mid-size” solution for messaging middleware. It’s attractive not only for its price tag (free!), but also because it can offer the right amount of power and features for most situations while requiring minimal setup and administration.

In this section we’ll look at using ActiveMQ with the **stomp** gem, a library that gives you the ability to interface with services over the Stomp protocol. Stomp (or, Streaming Text Orientated Messaging Protocol) offers a lightweight format for compatible clients to interact with message brokers. This means that while we’re talking to ActiveMQ in this section, you could just as easily use the **stomp** gem with any Stomp compatible message broker (a list of which is on the Stomp website at <http://stomp.codehaus.org/>).

Problem

Your Ruby application needs to integrate with a monitoring service that uses a message broker to get information from the services it’s monitoring.

Solution

After you get your ActiveMQ instance installed and configured (see the ActiveMQ and Stomp websites for information on how to do that), you’ll want to use RubyGems to install the stomp library.

```
$ gem install stomp
```

The service you’re integrating with uses a message queue to get information from the services it is monitoring through a queue named `application_errors`. It receives XML messages in a simple format:

```
<?xml version="1.0" encoding="UTF-8"?>
<error>
  <type>StandardError</type>
  <message>Something is broken.</message>
  <backtrace>
    NameError: uninitialized constant X
      from (irb):1
  </backtrace>
</error>
```

In our Websphere example, we used the `to_xml` method to generate XML from an ActiveRecord instance. Since we’re not using ActiveSupport (which provides `to_xml`) and our XML structure doesn’t follow the attributes of the error objects we’ll receive, we’ll need to generate these messages manually using Builder (you can learn more about using Builder to generate XML in Chapter 10, *Structured Documents*). A method to generate this XML structure from a Ruby error object looks something like Listing 7.6.

Listing 7.6: A method to take a Ruby error object and generate XML from it

```
def generate_xml(error_object)
  payload = ""

  builder = Builder::XmlMarkup.new(:target => payload, :indent => 2) #1
  builder.instruct!
  |

  builder.error do |error|
    |
  end #2
end
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```

        error.type error_object.class.to_s
        error.message error_object.message
        error.backtrace "\n#{error_object.backtrace.join("\n")}\n"
    end

    payload
end

```

Our `generate_xml` method takes an `error_object` as a parameter, which we'll grab information from to generate the XML. First, we start a new Builder document by creating an instance of `Builder::XmlMarkup` and telling it to generate the XML declaration #1. Next, we build our document using methods on the `builder` object #2.

Now, what's the best architecture for our error reporter? Since we'll be catching exceptions and reporting them to the error reporting service, it seems like a good idea to avoid creating an instance every time. In that case, we'll probably want to build a class and use class methods to handle the functionality. Listing 7.7 shows our implementation of this class.

Listing 7.7: Implementing our error reporter

```

require 'rubygems'

require 'stomp'
require 'builder'

class ErrorReporter
  def self.report!(error_object, queue="/queue/errors") #1
    reporter = Stomp::Client.new #2
    reporter.send queue, generate_xml(error_object) |
  end

  private
  def self.generate_xml(error_object)
    payload = ""

    builder = Builder::XmlMarkup.new(:target => payload, :indent => 2)
    builder.instruct!

    builder.error do |error|
      error.type error_object.class.to_s
      error.message error_object.message
      error.backtrace "\n#{error_object.backtrace.join("\n")}\n"
    end

    payload
  end
end

```

Here you can see that we define a method called `report!` (#1), which takes an error object as a parameter along with the name of the queue you want to push the messages to (defaulting to `/queue/errors`). Next, we create our `Stomp::Client` object (#2) and tell it to send a message to the queue from the parameters with a payload containing XML from the `generate_xml` method we created earlier. Now, we just have to implement this in some code.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```

def error_method
  FakeConstant.non_existent_method!
rescue StandardError => error_obj
  ErrorReport.report! error_obj
end

```

When running that code, an error will be raised and said error reported via our **ErrorReporter**. Upon checking the queues in your message broker's web interface, you should see a message there that has XML for the error object. That works great, but let's build a message consumer so we can test it easier.

Consuming messages with a **Stomp::Client** object centers around the **subscribe** method, which allows you to define behavior to respond to messages being pushed to a queue. Defining the behavior is handled by providing a block to the **subscribe** method; you can see a primitive example of this in Listing 1.1.

```

client = Stomp::Client.new
client.subscribe("/queue/testing") do |message|
  puts message.body
end

# Join the listener thread
client.join

```

Running the above code and then pushing messages into the **/queue/testing** queue will cause their payload to be printed to the console. The yielded **message** object is actually a **Stomp::Message** object, which also has a little bit more information about the frame if you need it (specifically, the frame headers and command). We join the thread otherwise the script will exit and the listener thread will be killed off.

So, to make a consumer for our error reporter, we'll need to pop the messages off the queue and process the XML inside of our **subscribe** call. To do so, we'll use REXML again (like we did in our RubyWMQ example); see our implementation in Listing 7.8.

Listing 7.8: Our testing consumer for the error reporter

```

require 'rubygems'

require 'rexml/document'
require 'stomp'

client = Stomp::Client.new

client.subscribe("/queue/errors") do |message|
  xml = REXML::Document.new(message.body)                                #1

  puts "*****"
  puts "- Error type: #{xml.elements['error/type'].text}"                 #2
  puts "- Error message: #{xml.elements['error/message'].text}"          |
  puts "- Error backtrace: #{xml.elements['error/backtrace'].text}"      |
  puts                                                                    |
end

client.join

```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

We create a `Stomp::Client` object and invoke `subscribe`; when a message is received we push the message body over to REXML (#1) and pull attributes out to print them to the console (#2). If you run this script and then throw a few errors to the error reporter, your console should look something like the following:

```
*****
- Error type: NameError
- Error message: uninitialized constant NonExistentConstant
- Error backtrace:
my_script.rb:3:in `test_call'
my_script.rb:12

*****
- Error type: NameError
- Error message: uninitialized constant IDontExist
- Error backtrace:
application.rb:9:in `invoke!'
application.rb:20
```

Discussion

In these examples, we've only shown you a few features of the `stomp` gem. To keep the examples short, we left these features out, but the library is actually quite full featured. For example, it supports authentication in the message broker. So, if we wanted to login as a user on the broker listening on the `localhost` at port `13333`, we'd simply do something like the following when creating the `Stomp::Client` object.

```
client = Stomp::Client.new("username", "pass", "localhost", 13333)
```

The library also supports reliable messaging using client side acknowledgement. For example, let's say you wanted to make sure that your authorization messages were being received; your code, at its base level, might look something like the following.

```
message = nil
client = Stomp::Client.new
client.send "/queue/rb", "Hi, Ruby!"

client.subscribe("/queue/rb", :ack => 'client') { |msg| message = msg }

client.acknowledge message
```

To learn more about these and other features (like transactional sending), you can check out the Stomp home page or generate the RDocs for stomp on your local machine.

STOMP DOCUMENTATION

The `stomp` gem doesn't actually generate the documentation for it when it installs itself, so you'll need to navigate to its directory and run `rdoc` to generate it yourself. RDocs aside, we found that the test suite was one of the best locations for information on how to use the library. The same goes for RubyWMQ and other libraries we cover in this book. Chalk it up to a developer community devoted to test-driven development.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

If Java and ActiveMQ aren't available to you or you don't want to go through the effort of setting it up, there is a pure Ruby stomp server available also. It's available as a gem named `stompserver`, which you can install by executing the following:

```
$ gem install stompserver
```

While it's not production quality, it's great for testing your application locally without creating a lot of overhead on your system. Even though it is useful, `stompserver` isn't the best pure Ruby asynchronous messaging solution available; in the next section we'll take a look at the best pure Ruby option that we've found: Ruby Reliable Messaging.

Testing a system like this

You might be asking...

7.2.2 Pure Ruby messaging with Ruby Reliable Messaging

If your preference is to stay in Ruby rather than using a tool from another language (for example, if you plan on hacking in some custom functionality) or if your requirements don't allow for extraneous software installation then perhaps a pure Ruby solution is in order. Assaf wrote a pure Ruby, reliable messaging library obviously named Ruby Reliable Messaging. In this section we'll take a look at using this library to fulfill your asynchronous messaging needs in a small setting where no heavy lifting is required.

Problem

You want to utilize asynchronous messaging, but you need to stay in the Ruby language.

Solution

To get started, you'll want to install the Ruby Reliable Messaging gem, which is named `reliable-msg`.

```
$ gem install reliable-msg
```

Next, you'll want to start the Reliable Messaging library's message broker, which is a daemon that exposes brokering over DRb.

```
$ queues manager start
```

After you have the broker started, you're ready to write some code! The Reliable Messaging library functions primarily through the `ReliableMsg::Queue` class and its `put` and `get` methods. These obviously named methods will put methods into the queue and get them out. As an example, take a look at Listing 7.9.

Listing 7.9: The Reliable Messaging library's core functionality centers on one class and two methods

```
require 'rubygems'
require 'reliable-msg'

queue = ReliableMsg::Queue.new 'ruby'           #1
queue.put "Simple message goes here."          |

msg = queue.get                                 #2
puts msg.object                                 |
# => "Simple message goes here."               |
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

You can easily see the simple API here. We create a `ReliableMsg::Queue` object and use the `put` method to place an object in the queue #1. Next, we use the `get` method to pop the first object (i.e., the string we just created) off the queue #2. Then we use the `object` method to get the object from the message and output it to the console (in this case, we just had a simple string). This simple API makes it very easy to worry more about the business logic around your messaging rather than the messaging itself.

While `reliable-msg` isn't meant for heavy-traffic messaging, it does offer one unique advantage over the other solutions that aren't Ruby from top to bottom: native object handling. When you place a message in the queue, the `reliable-msg` library does this:

```
message = Marshal::dump message
```

What does that mean for you? This means that you can put any type of object into the queue and have it serialized natively in Ruby. No more XML serialization, no tip toe'ing around serialization limitations, and no more worrying about serialization problems. Of course, this means that you can only integrate with Ruby clients, but then again, this section is called "Pure Ruby messaging."

For example, let's say you want to use a queue to pass work order information from your Rails application to your Ruby application on a different host that processes and dispatches work orders. Your `WorkOrderMessage` class might look something like Listing 7.10.

Listing 7.10: A class for work order information passing from a Rails application to a Ruby application

```
class WorkOrderMessage
  attr_accessor :requester, :requested_work, :date_due

  def initialize(params)
    raise "Invalid arguments provided." unless params.is_a?(Hash)

    @requester = params[:requester]
    @requested_work = params[:requested_work]
    @date_due = params[:date_due]
  end

  def unique_id
    "#{@requester.slice(0,5).strip.upcase}-" +
    "#{@date_due.strftime('%m%d%y')}-#{@requested_work.hash}"
  end

  def report!
    puts "Order #{@unique_id} received."
    puts "from #{@requester}, due #{@date_due.strftime('%D')}"
    puts "Work requested:"
    puts @requested_work
    puts
  end
end
```

In the class we have `attr_accessors` for various attributes, a couple of methods to format the information in a variety of ways, and a constructor that takes a `Hash` as its single parameter so we can pass from a request to a create action to the constructor directly.

In our Rails application, we then would have a `create` action that looks something like Listing 7.11.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

Listing 7.11: Creating a work order in the database and passing a message

```

def create
  @work_order = WorkOrder.new(params[:work_order])    #1
  @work_order.save!

  mq = ReliableMsg::Queue.new 'orders_queue'          #2
  message = WorkOrderMessage.new(params[:work_order]) |
  mq.put message                                     |

  flash[:notice] = "Work order submitted."
  redirect_to(work_orders_path)
rescue ActiveRecord::RecordInvalid
  render :action => 'new'
end

```

First, we create an ActiveRecord object and save it to the database #1. If that's successful, then we proceed to create our **WorkOrderMessage** object and place it in the queue #2. The rest of the method will place a confirmation message in the **flash** and redirect the user to the index of the **work_orders** controller.

BLOCKING YOUR RAILS APPLICATION

The only downside to this approach is that it blocks your Rails application when placing the message in the queue, which means that if you're moving around big objects or have a slow or busy queue, then your users will have to wait until it's done (or worse, the request may time out). See the *Discussion* section below for a few tips on how to avoid this.

Now we need to create a consumer for these messages for our Ruby application. Again, we'll use the **get** method to grab the message from the queue and process it. See Listing 7.12 for our implementation of a simple consumer.

Listing 7.12: Consuming messages from the queue

```

require 'rubygems'
require 'reliable-msg'

queue = ReliableMsg::Queue.new 'orders_queue'    #1

while true                                       #2
  while true
    break unless queue.get do |msg|             #3
      msg.report!                                |
                                              |
      # If you had a Processor class...          |
      Process.process! msg                       |
      True                                       |
    end
  end
  sleep 10 # No messages? Hold on!
end

```

First we create our Queue object #1, and then we start looping so that the consumer is always available #2. When we get a message (#3), we call its **report!** method to give us a report of what its work order contains then tell our **Processor** class (assuming we have one) to process it.

Discussion

We said that the method we showed for Rails integration wasn't the best. While it works for 80% of cases, it may cause problems for you if you have high traffic or big message objects. There are two popular options for Rails integrated messaging: AP4R and ActiveMessaging. AP4R builds on reliable-msg and offers a number of Rails specific features that will make your life much easier when trying to handle messaging. If you're interested in the minimal setup and configuration of reliable-msg and want to use it in combination with Rails application, we recommend checking out AP4R.

ActiveMessaging is another alternative for integrating messaging providers into Rails application, its strength is in supporting WebSphere MQ, Stomp, JMS, Amazon Simple Queue Service and reliable-msg, letting you switch between providers as your project need changes. It does require more setup than AP4R, with the clear advantage of supporting more messaging services.

If these libraries strike you as too much to setup or too much code to add to your application, you could also use something like spawn or BackgroundDRb to handle the sending (and optionally consuming) of messages. This option is simpler and possibly cleaner, but isn't as integrated or powerful.

In our examples, we didn't cover some of the more advanced features of reliable-msg. For example, you can specify the delivery method. By default, it's "best effort" (`:best_effort`), which means it will try to deliver the message once and if it fails the message will be discarded. Let's say you wanted a little more resilience. You decide to change the delivery behavior to repeat delivery 6 times and if it failed, place the message in the `dead-letter` queue.

```
queue.put message, :delivery => :repeated
```

You can also set the `:delivery` argument to be `:once`, which will try to deliver the message once and if it fails, the message will be placed in the `dead-letter` queue. These parameters can also be set on Queue objects so that it becomes the default for any message object passed to it.

```
queue = Queue.new("my_queue", :delivery => :repeated)
```

This will cause any message put in that queue to be delivered using the repeated method rather than the default. The `reliable-msg` library also has a very flexible consumption API and a number of options for persistence. You can read more about these in RDocs, which are generated when you install the gem.

VIEWING GEM DOCUMENTATION

If you want to view the documentation for gems on your system, then run `gem server` from your favorite command line. This script will start a web server which you can navigate to and browse a list of the gems you have installed and their accompanying documentation.

7.3 Summary

Integrating with systems that are totally foreign to your development environment can be trying, but solutions like asynchronous messaging make it much easier. Fortunately Ruby has a number of mature libraries to work with a wide variety of messaging vendors. In this chapter, we looked at using RubyWMQ to work with WebSphere MQ. This solution works well for large scale deployments, where WebSphere MQ is a requirement.

We also looked at ActiveMQ, an open source high throughput MoM, using the Stomp protocol. What you learned here will also apply to other messaging services that support the Stomp protocol, and that includes many of the JMS providers out there. Last, we looked at a lightweight alternative that requires minimum setup and configuration, the Ruby Reliable Messaging library.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

Now that we talked about developing Web applications, using Web services and asynchronous messaging, let's look at some options for deploying Ruby applications.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

Licensed to Keith Steward <paypal1@keithsteward.com>

8 Deployment

Much has been written on the topic of deploying applications written in Ruby on Rails. However, not all Ruby applications are web applications, and not all web applications use Rails. This chapter will focus on the deployment concerns that come up when deciding how to create Ruby applications and how to release those applications into a live environment once done.

For veteran Rails developers, a lot of this will be similar: we will discuss using Capistrano to deploy applications, packaging up reusable components in Ruby gems, and using God to monitor a deployed application. But we will go beyond the common uses of these tools, and explain how to customize them for applications that require remote servers but are not necessarily Rails-based.

Keep in mind that the subject of deploying live applications is complex. It could easily, and does frequently, take up an entire book of this size. This chapter will focus on tools written in Ruby that will help deploy applications and libraries. If you're interested specifically in Web application, you can find specific setup and configuration instructions in Appendix C.

8.1 Creating deployable packages with RubyGems

When deciding how to structure new Ruby programs, it's worth taking some time to decide how you will deploy them. While some of the programs you create will be yours and yours alone, you may also find yourself creating code for hundreds or even thousands of developers. This may be because you are writing open-source software, or because others will use your code in a large enterprise situation, or even because you are writing commercial code to be used by those who purchase it.

In this section, you will learn to create skeletons for your code that will make it easy to release and deploy to other developers. Because Ruby's gem system is designed to be robust and decentralized, this section will focus on building RubyGems and creating your own, secure gem repositories.

USING THE LATEST VERSION OF RUBYGEMS

RubyGems is an extension of Ruby and not yet part of the official Ruby distribution, although some distributions do contain it. The One-Click Ruby Installer (Windows and OS/X) includes RubyGems, so does OS/X 10.5 (Leopard), JRuby and Ruby 1.9.

If you do not have RubyGems installed, we recommend downloading RubyGems 1.1.0 or later from the RubyForge Web site and installing it by unpacking the archive file and running the `setup.rb` file found there. For example:

```
$ wget http://rubyforge.org/frs/download.php/29548/rubygems-1.1.0.tgz
$ tar xzvf rubygems-1.0.1.tgz
$ cd rubygems-1.0.1
$ sudo ruby setup.rb
```

If you already have RubyGems installed, we recommend upgrading to 1.1.0 (this may require upgrading to Ruby 1.8.6). You can check which version of RubyGems you have by running the command `gem --version`. To upgrade to the latest version:

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```
$ gem update --system.
```

You can find more information about installing Ruby and setting up RubyGems in Appendix A.

8.1.1 Using RubyGems In your organization

You run the IT department of a medium-size programming firm. Pretty much everyone's writing Ruby code, but projects are having difficulty sharing code amongst each other. For instance, one team spent all month writing some code to send alerts out via AOL Instant Messenger, and another team was several days into implementing the same code before they realized it had already been done. To solve this, you want to provide a centralized repository of Ruby libraries that have been produced by different teams in your organization.

Problem

Specifically, you would like to use the RubyGems system to allow your teams to integrate their packages into their existing libraries of Ruby code. You want them to be able to use their existing tools and have downloaded gems integrate seamlessly into their existing repositories. Finally, you want to be able to support dependencies and requirements, as well as support for automatically installing C extensions.

Solution

Use Ruby tools to generate gem skeletons and fill in information about dependencies and C extensions. Host these gems on a server available to your company and give the developers a URL that they can use with the `gem` command-line utility to pull gems from your company repository instead of the default RubyForge repository.

Let's tackle these problems one at a time. First, let's take a look at generating empty gems that you can fill in with the appropriate information. We're going to use the `newgem` utility to build the skeleton. That utility allows you to build a full skeleton, including a website to upload to RubyForge and other vagaries. However, we're going to use the `--simple` switch to tell `newgem` to generate only enough code to get us up and running.

Inside the directory you just created (by running `newgem ruby_in_practice --simple`), you should see a pretty simple structure containing a series of files at the root level (`LICENSE`, `README`, `Rakefile`, and `TODO`). There's a directory for your library (`lib`), and directory for your tests (`spec`). There will also be a folder called `script` that includes generator scripts for Rubigen (which we will not be discussing here).

The main work we'll be doing here is inside of the `Rakefile`, which holds the configuration for the gem we'll be building. You'll want to replace the sample information at the top, which will automatically be used below in the gem specification. By default, the created gem will include all of the files in your `lib` and `spec` directories; you can change that by modifying the `s.files` declaration to include other files. (Here "s" refers to the variable that holds `Gem::Specification`) You can add dependencies by using `s.add_dependency` inside the gem specification. For instance, if you wanted to require `hpricot 0.5` or higher, you would do `s.add_dependency "hpricot", ">= 0.5"`. You can include dependencies that are private to your organization or common, gems that are on the main RubyForge server. Newer versions of RubyGems will automatically install extensions that it finds in all available locations.

Including C extensions is pretty simple too, assuming you have a working `extconf.rb` file already. Create a new directory inside the gem skeleton called `ext` and place your `extconf.rb` and any required C files into it. Then, add the following line to the gem specification:

```
s.extensions = "ext/extconf.rb"
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

This tells the gem installer to run `extconf.rb` and then `make` and `make install` the generated `Makefile`. Effectively, this will work the same as requiring your users to manually install the extension. Of course, if you don't have C extensions to install, you can safely ignore these instructions. Your gem specification should look like Listing 8.1.

Listing 8.1: Gem Specification

```
GEM = "ruby_in_practice" 1
VERSION = "0.1.0"
AUTHOR = "Sample McSample"
EMAIL = "sample@example.com"
HOMEPAGE = "http://sample.example.com"
SUMMARY = "A sample gem for Ruby in Practice"

spec = Gem::Specification.new do |s|
  s.name = GEM 2
  s.version = VERSION
  s.platform = Gem::Platform::RUBY
  s.has_rdoc = true
  s.extra_rdoc_files = ["README", "LICENSE", 'TODO']
  s.summary = SUMMARY
  s.description = s.summary
  s.author = AUTHOR
  s.email = EMAIL
  s.homepage = HOMEPAGE

  s.add_dependency "hpricot", ">= 0.5"

  s.require_path = 'lib'
  s.autorequire = GEM 3
  s.files = %w(LICENSE README Rakefile TODO) + Dir.glob("{lib,specs}/**/*")
end
```

1 This name is used below

2 The name of the gem

3 The name of the ruby file to require automatically

Next up, you're going to want to add your code to the generated `lib/ruby_in_practice.rb`. If you have more than one file, you should create a new directory called `ruby_in_practice`, and require them from `ruby_in_practice.rb`. You should name the directory the same name as your gem so that people requiring individual files under your directory could use the conventional `require "ruby_in_practice/component"`. Your `ruby_in_practice.rb` would look something like Listing 8.2.

Listing 8.2 Requiring Ruby files from a Gem

```
require "ruby_in_practice/parser"
require "ruby_in_practice/lexer"
require "ruby_in_practice/interactive"
```

Requiring `ruby_in_practice` from an external file will automatically push the items under `s.require_path` (in this case `lib`) into the load path, so requires like those in listing 8.1 will work great.

Now that we've created our gem, what about serving it to others in our company? We're going to leave out details like monitoring the server and keeping the server up for later in this chapter. The next step is to package up your gems to deploy onto the remote server. Thankfully, you're in luck; `rake` includes a gem packager and

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

`newgem` includes this rake task in its generated code. All you need to do is go into the `ruby_in_practice` directory and run `rake package`. That will compile the gem and put it into the `pkg sub`-directory.

Once you're done, copy the gem onto the server you'll be serving the gems from. On the remote server, run `gem install ruby_in_practice-0.1.0` in the directory you copied the gem into. It will now be in the server's local gem repository. You can turn it into a gem server by running, you guessed it, `gem server --port 80`. Once you've done that, other machines will be able to run `gem install ruby_in_practice --source your_url.com`, and the local instance of RubyGems will work exactly the same as the remote version on RubyForge. Hitting the URL from a browser will provide a list of all of the available gems, with links to RDoc. In essence, it becomes your company's private little RubyForge.

Of course, you'll want to restrict access to the URL, probably by keeping it inside your company's intranet, and monitor it to keep it up (more on that later in this chapter), but you're pretty much done at this point. Ruby and RubyGems provided all the tools necessary to host your own company gem repository out of the box.

8.1.2 Setting up a RubyGems repository

If you have an organization that needs to share ruby code written by different departments, or if there's another reason you're distributing code to be used by others, you're going to want to start by setting up your own repository.

Problem

You need to deploy various software packages to a remote server to run, but managing dependencies has become quite complex. Additionally, it's becoming quite painful to manage the process of getting your files to the server and keeping track of which versions of things are running. Having already set up a server in the previous section, you want to incorporate your newly organized dependency system into your deployments.

Solution

You'll package up and version your code into their own gems, and deploy the gems onto the server that will be running the code. Using the binary features of RubyGems, you will be able to create a binary with your package that will run your code, and that you will be able to execute and monitor on the remote server. Because you will be using gems, you'll be able to specify both RubyForge dependencies and dependencies on your own gem repository.

You already know the basics of packaging up your code into gems. If you use the technique from this section, you'll want to develop inside of a gem structure, so that you'll be able to use `rake package` and `rake install` to test your gem locally. We're going to need to make a few changes to the gem specification from Listing 8.1 to add support for a binary that will run our code. Add a `bin` directory under `ruby_in_practice`, and add a file called just `ruby_in_practice` under it. Your binary will typically look something like Listing 8.3.

Listing 8.3: Deployable binary

```
#!/bin/env ruby
require "ruby_in_practice"
RubyInPractice.start
```

That assumes that you have a module in your codebase somewhere called `RubyInPractice` with a class method called `start`. This is a convenient way to structure your code, so you can keep your binary very simple and unlikely to change between versions. Keep in mind that in Ruby, `Dir.pwd` is the directory that the

binary was run from, while `File.dirname(__FILE__)` is the directory that the code is in. This will allow you to flexibly decide where to store support files like logs and pid files.

In order to make sure that your binary will get deployed along with your code, you'll need to make some changes to the gem specification we put together for Listing 8.1. Let's take a look at just the gem specification part of the Rakefile in Listing 8.4.

Listing 8.4: Gem specification including binary

```
spec = Gem::Specification.new do |s|
  s.name = GEM
  s.version = VERSION
  s.platform = Gem::Platform::RUBY
  s.has_rdoc = true
  s.extra_rdoc_files = ["README", "LICENSE", 'TODO']
  s.summary = SUMMARY
  s.description = s.summary
  s.author = AUTHOR
  s.email = EMAIL
  s.homepage = HOMEPAGE

  s.bindir = "bin"
  s.executables = "ruby_in_practice"

  s.add_dependency "hpricot", ">= 0.5"

  s.require_path = 'lib'
  s.autorequire = GEM
  s.files = %w(LICENSE README Rakefile TODO) +
  [CA]Dir.glob("{lib,specs,bin}/**/*")
end
```

1 The binaries directory

2 Executables to install, relative to the binaries directory

3 Add bin to the list of files to include in the gem package

Now, all you need to do is package up the gem, drop it on your server, and run `gem install` on the server. If you have a company-wide gem server, you could drop it on that server, and deploy from there. You can specify dependencies and C extensions just as before, and your app will gracefully refuse to run if a dependency is not met. Say goodbye to dependency hell!

The `gem server` command will open up a server on port 8808, making the local repository available to users via the `-source` parameter to the normal `gem` command. For some examples, see Listing 8.5.

Listing 8.5: Using a custom repository with the gem command

```
gem install rails --source=http://example.com:8808
gem list --source=http://example.com:8808
gem query -nrails --source=http://example.com:8808
```

Your users will be able to install a gem (#1), list all available gems (#2), query the gem list for those matching a particular pattern (#3), and many other commands that are available via the `gem` command. In essence, your machine has become an alternative to the canonical RubyGems server. For more information on the available commands, you can run the `gem help` command from your command line.

When you create your repository using `gem server`, you also have a number of options that will allow you to customize how you expose your repository to the world. Those options are detailed in Table 8.1.

Table 8.1: The available options to the `gem_server` command

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

Option	Effect
<code>--daemon</code>	Run the server in daemon (background) mode
<code>--p[ort]=PORT</code>	Run the server on the designated port. If you use port 80, your users will be able to access the gem server without supplying a port to <code>gem --source</code>
<code>--d[ir]=LOCATION</code>	The location of the gem repository. See listing 8.3 for more details

As indicated in Table 8.1, you can specify a location for the gem server that you expose. Say, for instance, that the server you were using for gems also had its own set of gems that you did not wish to expose (to make for more readable `gem list` output, for instance). You could install the gems you wished to expose in a custom location via `gem install -i`, and then use the `location` parameter to `gem_server` to expose just those gems.

Listing 8.6: Using a custom location for `gem_server`

```
gem install rails -i ./gems 1
gem_server -d ./gems --daemon 2
```

Assuming you are in the directory you wish to use to store just the gems to expose, Listing 8.6 demonstrates how you can install the gems you want into a new directory (#1) and then call `gem server` with the necessary settings (#2). You don't need to call the directory `gems`, but it's conventional, and good practice to do so.

Ruby Gems are the standard mechanism for deploying libraries, command line tools, plugins, even full libraries like Rails. Deploying Web applications is a different process. Besides pushing Ruby code, you'll want to manage the Web server, update the database schema, and various other management tasks. In the next section we'll talk about Capistrano and Vlad the Deployer, two tools designed specifically for deployment and management of Web applications.

8.2 Deploying Web Applications

So now you know how to deploy libraries and simple command-line applications. Web applications tend to be more complex. Besides the application and libraries you have to take care of the Web server itself, manage the database, process log files, mount network drives, and so forth. You might be deploying to a cluster of machines, configuring and managing multiple servers. These tasks call for a different kind of deployment and management tool.

In this section we're going to discuss two such tools, Capistrano and Vlad the Deployer. Capistrano was developed originally for deploying Rails applications, but can be used for many other remote deployment and management tasks. We'll show a couple of examples for that. Or you can use Vlad which provides all the same features, but based entirely on Rake.

8.2.1 Simplifying Deployment

You maintain a daemon written in Ruby that needs to be deployed to four servers. You use subversion, and release the daemons to the server off trunk when the release is ready. You want to make sure that all four releases make it to production, and that if any of them fail, they silently roll-back.

Problem

Your daemon is started and stopped via a shell command (`daemon_ctl start` and `daemon_ctl stop`). The start and stop commands handle cleaning up any zombie PID files. Finally, you have a production

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

environment, with four servers, and a staging environment with a single server. Both environments have identical requirements.

Solution

For this task we're going to use Capistrano. Before setting up, you'll need to gather some information. Capistrano requires the username for both your remote servers and subversion repository, the repository URL and what directory your code will be checked out to.

Once you've gathered all the information you need, you need to add a file called **Capfile** to the root of your source tree. Deployment with Capistrano uses recipes, which are similar in nature to Rake tasks. You'll find recipes for managing Apache Web servers, deploying Rails applications, using source control, starting and stopping background processes, migrating database schemas and much more. These are placed in the Capfile, loaded by the command line tool **cap**. The Capfile we'll need for this problem is pretty straight forward, and Listing 8.7 is a sample of what you might need.

Listing 8.7: Capfile needed to deploy a simple daemon

```
load 'deploy' if respond_to?(:namespace)

set :keep_releases, 5
set :application,   "ruby_in_practice"
set :repository,    "http://svn.rubyinpractice.com"
set :scm_username,  "ruby"
set :scm_password,  "1npr4ct1c3"
set :user,          "rubies"
set :password,      "pract1cin6"
set :deploy_to,     "/deploy/#{application}"
set :deploy_via,    :export

ssh_options[:paranoid] = false
1

task :production do
  role :app, "http://prod1.rubyinpractice.com"
  role :app, "http://prod2.rubyinpractice.com"
  role :app, "http://prod3.rubyinpractice.com"
  role :app, "http://prod4.rubyinpractice.com"
end
2

task :staging do
  role :app, "http://staging.rubyinpractice.com"
end
3

before "deploy", "daemon:stop"
after "deploy",  "daemon:start"

namespace :daemon do
  desc "Start daemon"
  task :start do
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```

    on_rollback { run "cd #{deploy_to} && daemon_ctl stop" }
    run "cd #{deploy_to} && daemon_ctl start"
end

desc "Stop daemon"
task :stop do
  on_rollback { run "cd #{deploy_to} && daemon_ctl start" }
  run "cd #{deploy_to} && daemon_ctl stop"
end
end
1 Required by Net::SSH
2 Production environment
3 Staging environment
4 Set rollback proc

```

In this relatively simple Capistrano recipe, we set up our environment on top, set up different environments for production and staging, and created two new tasks to start and stop the daemon. In order to be sure that the daemon will be turned off while we deploy, and turned back on after we deploy, we used before and after filters to trigger the daemon at the appropriate times.

The most interesting part of the Capistrano recipe is the **on_rollback** callbacks. Capistrano will automatically execute the rollback functions if any of the deployments fail, so this allows you to specify that if the deployments fail, your daemons will turn back on.

REUSING RECIPES

Don't have time to re-invent the wheel? A quick search on Google will reveal Capistrano recipes that you can copy and paste into your Capfile and get started building on other people's experience.

One you want to look at first is Deprec, available from deprec.rubyforge.org. Deprec is a collection of pre-canned Capistrano recipes for setting up a production Rails server from scratch. And we do mean from scratch. It will copy SSH public keys over, install the entire Ruby on Rails stack, setup a working Web server (Apache or nginx, Mongrel) and get you up and running with minimum fuss.

Discussion

As you can see, it's pretty simple to use the default Capistrano recipe for deployment to deploy non-Rails applications. That's because Capistrano's default code mainly handles checking code out of your SCM and deploying it to a remote location. You can use the **run** command to call out to shell commands on your remote servers, and they will be fired on all remote servers.

If necessary, you can use the **sudo** command to run the commands with superuser privileges. This allows you a fair bit of flexibility, effectively allowing you to do anything through Capistrano that you'd be able to do by manually logging into your remote servers. Capistrano uses SSH to log into the remote server as the current user; if you maintain separate accounts you can tell Capistrano to log in as a different user by passing **USER=name** as a command line argument.

An important caveat of using **run** and **sudo** is that each command is run separately, and history is not preserved. That's why we do **run "cd #{deploy_to} && daemon_ctl stop"** instead of two separate run calls. That is typically not a problem, but it sometimes requires the creation of fairly convoluted code. In this respect, at least a passing knowledge with the intricacies of bash can become quite helpful.

Next, let's take a look at using Capistrano to intercept the incoming SSH stream and allow you to tail all of your remote logs at once.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

8.2.2 Tailing remote logs

Now that we have our daemon purring away on four remote servers, we might have occasion to want to look at the remote logs. Typically, you want to see the logs from all four servers at once, with some sort of line-header indicating what server it's from. Since Capistrano uses Net::SSH, we can intercept the incoming stream, examine it, modify it, and then output it.

Problem

You want to connect to all of your remote servers, tail the logs, and have the results outputted to your local monitor.

Solution

We're going to use Net::SSH's incoming stream to get what we need. Capistrano's `run` and `sudo` commands, both take an optional block with three arguments. The first argument is the channel, which is a `Net::SSH::Connection::Channel` object. You can get the full information about the object in the Net::SSH RDoc, but we will mainly use this object to extract information about which connection is being used. The next argument is the stream, which will be `:err` if the SSH connection returned an error. Finally, the last argument is the data being returned from the remote server.

This all comes together in Listing 8.8, where we add support for tailing remote logs to our Capistrano recipe.

Listing 8.8: Tailing a remote log

```
desc "Tail log file to console"
task :tail do
  run "tail -f #{deploy_to}/log/daemon.log" do |channel, stream, data|
    server = channel[:server]
    puts if @last_host != server
      1
    puts data.map {|line| "#{server}: #{line}" }.join("\n")
      2
    break if stream == :err
      3

    @last_host = server
  end
end
1 Remote server name
2 Print the data to the screen
3 Break if the server returned an error
```

As you can see, the recipe is pretty simple: we run a command on the remote server, and then process it. We use an instance variable called `@last_host` to track the previous returned message, so we can put a blank line between messages from different servers.

Discussion

There's not much to say about this recipe, because it's so straight-forward. You probably want to take a look at the Net::SSH documentation, just to satisfy yourself about the internals (and especially if you want to go further with remote tailing), but the basics are pretty... well, basic. It's possible to get more adventurous, and make each host use a different color for output. That's left as an exercise to the reader, but you might want to check out the `ansi-colors` gem, which allows you to do things like `"string".red.on_white`.

INTERACTIVE TASKS USING HIGHLINE

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

When Capistrano needs to prompt you for input, it uses a terminal I/O library called HighLine. HighLine has a lot of interesting features for writing interactive tasks that require user intervention. You can use it for colored output, to ask questions, present a menu of choices and hide passwords entered on the console.

We won't discuss HighLine at length here, but instead show you a couple of examples:

```
task "time" do
  Capistrano::CLI.ui.say("The time is <%= color(Time.now, BOLD) %>")
end

task "nuke_everything" do
  if Capistrano::CLI.ui.ask("Nuke everything!?" ) == 'yes'
    nuke!
  else
    puts "Better safe than sorry."
  end
end
```

Another tool for remote deployment and management is Vlad the Deployer. The concepts are all the same, both fulfill the same role, where Capistrano and Vlad differ is in the way you define and configure tasks. Vlad is based on Rake, which we covered in Chapter 3, and if you're already using Rake extensively you may find it more familiar.

8.2.3 Vlad the Deployer

As promised, let's dive into using Vlad the Deployer to handle deployment tasks. As we write this, it's still not quite up to snuff (it wouldn't be trivial to tail logs with Vlad as we did with Capistrano), but for simple deployment it's certainly a sight to behold.

Problem

As in Section 8.3 above, we have a daemon that needs to be deployed to four servers via subversion.

Solution

Vlad the Deployer is a ruby package written by the Ruby Hit Squad. It aims to resolve several perceived core deficiencies with Capistrano. Specifically, it is dramatically smaller than Capistrano, weighing in at only 500 lines of code. It uses **rake** as its core, so you can leverage your existing rake knowledge and existing rake tools in your deployment process. If you like using Rake for various development and management tasks you'll appreciate the being able to intermix deployment into you existing arsenal (think: run your tests before deploying).

The Rakefile in Listing 8.9, shows how we would use Vlad to perform the same tasks as we use Capistrano for earlier.

Listing 8.9: Vlad the Deployer

```
require 'vlad'
Vlad.load

set :domain,      "rubyinpratice.com"
set :deploy_to,   "/deploy/ruby_in_practice"
set :repository,  "http://svn.rubyinpractice.com"

role :app,        "prod1.#{domain}"
role :app,        "prod2.#{domain}"
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```

role :app,          "prod3.#{domain}"
role :app,          "prod4.#{domain}"

namespace :vlad do
  task :update => "daemon:stop"

  task :start do
    Rake::Task["daemon:start"].invoke
  end
end
end

```

Discussion

Since Vlad uses Rake, you can use Rake's before and after mechanisms instead of needing custom methods. In particular, you can get a before hook by simply re-declaring a task and giving it a dependency (Rake calls them prerequisites). You can get an after hook by re-declaring a task and providing additional contents.

To trigger a Vlad deploy, run `rake vlad:update vlad:migrate vlad:start`. You can produce a simple rake `vlad:deploy` easily as shown in Listing 8.10.

Listing 8.10: Rake vlad:deploy

```

namespace :vlad do
  task :deploy => [:update, :migrate, :start]
end

```

Now that we've covered two ways to get your code onto the server, let's tackle keeping your code running once it's on your remote server. Next up, we'll be taking a look at a little Ruby utility called God that should handle most, if not all, of your monitoring needs.

8.3 God.rb

Once you have your daemon running on the remote server, you'll want to keep an eye on it. If it starts consuming too many system resources, you'll want to restart it; if it goes down, you'll want to be notified.

The tool we're going to discuss in this section is called God. God's influence is monit, a well known UNIX system management tool. Whereas monit uses its own syntax and mini configuration language, God allows you to write notification rules directly in Ruby, for more flexibility and control. It also solves some of the most annoying issues that were present in monit, especially when handling daemons and PIDs. In short, God is everything that monit is as well as everything people wanted monit to be. You can find God at <http://god.rubyforge.org>.

Let's start by showing you how to setup God to monitor a number of daemons deployed on a remote server.

8.3.1 A typical God setup

Problem

You want to watch your daemon, and make sure it stays up. Additionally, you want a way to gracefully start and stop the daemon, as well as a way to make sure that it's not gobbling up all of your system resources.

Solution

In Listing 8.11, we'll use God.rb to set up our monitoring environment, which will include monitoring, graceful startup and shutdown, and resource-usage monitoring.

Listing 8.11 Watching daemon processes with God.rb

```
GRACE = 10.seconds
```

```
[1, 2].each do |number|                                     1
  God.watch do |w|
    w.name = "my-daemon_#{number}"
    %w{ start stop restart }.each do |task|                 2
      w.send("#{task}=", "/releases/daemons/daemon_#{number}_ctl #{task}")
    end
    w.start_grace = GRACE
    w.restart_grace = GRACE
    w.pid_file = "/releases/damons/logs/daemon_#{number}.pid" 3
    w.group = "my-daemons"                                     4

    w.behavior(:clean_pid_file)                               5

    w.start_if do |start|                                     6
      start.condition(:process_running) do |c|
        c.interval = 5.seconds
        c.running = false
      end
    end

    w.restart_if do |start|                                   7
      restart.condition(:memory_usage) do |c|
        c.above = 50.megabytes
        c.times = [2, 3]
      end

      restart.condition(:cpu_usage) do |c|
        c.above = 30.percent
        c.times = 5
      end
    end

    w.lifecycle do |on|                                       8
      on.condition(:flapping) do |c|
        c.to_state = [:start, :restart]
        c.times = 5
        c.within = 5.minutes
        c.transition = :unmonitored
        c.retry_in 10.minutes
        c.retry_times = 5
        c.retry_within = 2.hours
      end
    end
  end
end
end
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

- 1 Loop through both daemons
- 2 Loop through daemon tasks
- 3 Location of the PID file
- 4 Group daemons together
- 5 Clean zombie PIDs
- 6 Start condition
- 7 Restart conditions
- 8 Lifecycle conditions

Discussion

This configuration file, while designed to handle our special case that we developed above in the Capistrano section, is still very similar to the configuration file that the creator of God.rb uses to monitor his mongrels at gravatar.com, and which he makes available on the God.rb site. That's because the basic notion of monitoring tasks is extremely repetitive, and why you'd want a tool like God.rb in the first place.

In order to avoid having to type the entire configuration file twice, we loop over the daemon numbers (#1), similarly to how we handled the problem in Capistrano. If we add additional daemons, we'll probably want to add more daemons to this list. Also note that this God configuration file is appropriate for our staging environment, where we have both daemons on the same server. However, we can use the same exact script on the production environments, just removing the daemon numbers that will not be run on the server in question.

Also similarly to our solution in capistrano, we loop over the start, stop, and restart tasks, so we can easily change the exact command that is used to manage the daemons later (#3) without having to change it in three places. We specify the location of the PID file that will be created by the daemon (#4), which in this case will be inside the log directory in our release directory.

Because we rely on PID files, we must consider the possibility that our code will be killed without the opportunity for the PID to be correctly cleaned. This is called a "zombie PID" and it can wreak havoc with our daemon start script. That's why God provides the `:clean_pid_file` behavior, which will wipe out existing PIDs before it attempts to startup your code (#5).

God's condition syntax is pretty simple as well. For every event (start, stop, restart), you can specify conditions that should trigger the event. In this case, we trigger the start event if the daemon is not running, and we poll the daemon every five seconds. We trigger the restart event if memory usage is greater than 50MB (two out of the last three times we've tested) or CPU usage is greater than 30 percent five polls in a row.

Note that the start condition uses a special five-second interval, while the restart tasks use the default 30 second interval. If you want the interval to be different, simply specify `w.interval = 60.seconds`, or whatever you want your interval to be, and it will be used anywhere a condition does not set its own interval.

Finally, we can set some events to occur across state changes (called "lifecycle conditions"). In this case (#8), we want to handle the situation where God keeps trying to start the process, but it keeps failing (the "flapping" condition). In this case, we want to try more frequently early on, but then taper off as time goes on, and eventually give up. Specifically, in this case, if the daemon gets started or restarted five times in five minutes, stop monitoring for ten minutes (in case the problem was intermittent and will go away if you leave it alone for a bit). If the "flapping" occurs more than five times over two hours, give up.

You'll probably want to set some sort of notification when the monitoring fails so miserably. And lucky for you, that's next!

8.3.2 Notifications

Now that we have an infrastructure set up for keeping our daemons up and running, we'll probably want to be notified if they go down. In particular, when the "flapping" condition gives up after two hours of failures, we probably want the entire team to be notified. On the other hand, more minor issues can be sent to the team lead only.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

Problem

We want to set up God.rb to notify the entire team after a flapping failure, but only the team lead if the process is restarted due to excess CPU or RAM usage.

Solution

In Listing 8.12, we'll take a look at how to set up God to send emails.

Listing 8.12 Telling God.rb how to notify our team members

```
God::Contacts::Email.message_settings = {
  :from => 'daemon_master@example.com'
}

God::Contacts::Email.server_settings = {
  :address => "daemons.example.com",
  :port => 25,
  :domain => "example.com",
  :authentication => :plain,
  :user_name => "daemon_master",
  :password => "t3hm4n"
}

{:lead => "lead@example.com",
 :joe => "joesmith@example.com",
 :john => "theman@example.com",
 :mark => "marky@example.com"}.each do |name, email|
  God.contact(:email) do |c|
    c.name = name
    c.email = email
    c.group = "dev"
  end
end
```

To simplify setting up large numbers of emails, we've looped through a Hash and set the emails all at once. If you had a very large number of emails to set up, you might do something similar but with an external YAML file. Now that we have our emails and groups set up, we'll attach our notifications to the earlier conditions in Listing 8.13.

Listing 8.13 Monitoring with notifications

```
GRACE = 10.seconds

[1, 2].each do |number|
  God.watch do |w|
    w.name = "my-daemon_#{number}"
    %w{ start stop restart }.each do |task|
      w.send("#{task}=", "/releases/daemons/daemon_#{number}_ctl #{task}")
    end
    w.start_grace = GRACE
    w.restart_grace = GRACE
    w.pid_file = "/releases/damons/logs/daemon_#{number}.pid"
    w.group = "my-daemons"

    w.behavior(:clean_pid_file)
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```

w.start_if do |start|
  start.condition(:process_running) do |c|
    c.interval = 5.seconds
    c.running = false
    c.notify = "lead"
  end
end

w.restart_if do |start|
  restart.condition(:memory_usage) do |c|
    c.above = 50.megabytes
    c.times = [2, 3]
    c.notify = "lead"
  end

  restart.condition(:cpu_usage) do |c|
    c.above = 30.percent
    c.times = 5
    c.notify = "lead"
  end
end

w.lifecycle do |on|
  on.condition(:flapping) do |c|
    c.to_state = [:start, :restart]
    c.times = 5
    c.within = 5.minutes
    c.transition = :unmonitored
    c.retry_in 10.minutes
    c.retry_times = 5
    c.retry_within = 2.hours
    c.notify = "developers"
  end
end

end
end

```

We covered the basics of God.rb and showed you how to monitor a set of daemons, respond to different lifecycle events and send notifications when things go wrong. That's enough information for you to get started using God.rb in your environment.

8.4 Summary

Deploying Ruby applications is a complex topic. Most people use a mix of Ruby tools, Unix tools, and hand-rolled tools in their full deployment solution. What most people don't know, however, is how mature existing Ruby tools are for the task. In fact, Ruby being a flexible and powerful language for running command line scripts, you should also consider using these tools for tasks not specific to Ruby, such as deploying PHP applications or monitoring various servers and processes in your environment. For automated deployment to a remote server, you can use either Capistrano or Vlad the Deployer to automate even fairly complex, transactional remote tasks. To share code within your organization, the RubyGems infrastructure has

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

become extremely stable with 1.0, allowing you to create, within your organization, a repository akin to the one people use for their day-to-day packaging systems.

For monitoring, you can say goodbye to the arcane monit tool, and say hello to the pure-Ruby God.rb. God.rb even handles emailing and fairly complex monitoring rules, so you'll never need to manually SSH into your server to find out why your app is down again.

And the landscape is only getting better with time. Capistrano, Vlad, and God.rb get better with every release, and as Ruby improves, so will these tools. Compared with old-school Unix utilities that are stuck in the 90s and require some fairly advanced system programming knowledge to hack on, the choice is clear. Ruby deployment tools have arrived.

We just covered Web applications and deployment, and through some of the examples showed you how to build Ruby applications that use relational databases. In the next chapter we're going to delve deeper into databases and show you more options for handling data storage from Ruby applications.

9 Database Facilities and Techniques

So far we've taken a look at a number of technologies and how to use them in Ruby. Now we'll turn our attention to databases, and this chapter will introduce you to some key database tools available with, and for, Ruby. We'll take a broad view of what it means for something to be a "database." Our working definition will be that a database is a storage unit external to your program where data can be stored. This definition includes highly-evolved structures like relational databases, but also includes, potentially, plain-text files. We'll also stipulate that the stored data has to have some intelligence about itself. A file containing a stream of words—though it may be loaded into a program as a string or an array—will not count as a database, because the file itself does not preserve any information about the structure of the data.

A flat file in YAML format, however, will count as a database, under our definition. As you'll see, it's possible and not even terribly difficult to wrap a YAML file in a workable API, and reap the benefits of addressing your data structurally while also having access to it in text form. Text files are not the whole story, of course. Ruby ships with several flavors of the DBM (Berkeley database) system. We'll also look at some of the tools available for creating and manipulating relational databases in Ruby.

Throughout the chapter, we'll work with a specific example: the implementation of an API for storing personal contacts — in plainer speech, an address book. We're not so much concerned here with implementing the whole address book, as with implementing the programming interface to the database facilities. Examples will include a certain amount of address-book implementation, but primarily we'll be looking at how to set up a database in which contact information can be stored, and how to talk to such a database in Ruby. We'll implement the API twice, once for YAML storage and once for GDBM. We'll then use the MySQL and DBI libraries to move data from one database format to another.

We're going to start with YAML, a data serialization tool that can, with a little assistance from Ruby, form the kernel of a simple data persistence library.

9.1 Using plain-text files for data persistence

Using YAML as a data persistence tool is an example of the more general case of using plain-text files for this purpose. You'll find other ways to do this in Ruby, such as the CSV (comma-separated values) facility in the standard library. XML files fall into this category, too.

None of these are full-fledged database systems. What they have in common with such systems, though, is the fact that they include information about the data, together with the data. CSV files don't contain *much* information about the data, but they do preserve ordering and, often, something like column or header information. XML preserves relationships in and among nested data structures; YAML does something similar, though with the avowed goal of being somewhat easier to read than XML, and more suitable for storing arbitrary Ruby data structures.

In looking at plain-text data storage techniques, then, we'll focus on YAML, as the one that offers the richest combination of complexity on the data end and editability on the text end.

9.1.1 Data persistence with YAML

YAML ("Yet Another Markup Language", or "YAML Ain't Markup Language", depending who you ask) is a data serialization format: Ruby objects go in, and a string representation comes out.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```

hash = { :one => 1, :two => 2, :colors => ["red", "green", "blue"] }
# => {:colors=>["red", "green", "blue"], :one=>1, :two=>2}
# require 'yaml'                                #1
puts hash.to_yaml                                #2
---
:colors:
- red
- green
- blue
:one: 1
:two: 2

```

YAML is part of the Ruby standard library: you just have to load it (#1) and then your objects can be serialized to YAML using the `to_yaml` method #2.

The strings generated by YAML conform to the YAML specification. YAML itself is not specific to Ruby; Ruby has an API for it, but so do numerous other languages.

Objects serialized to YAML can be read back into memory. Picking up from the last example:

```

y_hash = hash.to_yaml
# => "--- \n:colors: \n- red\n- green\n- blue\n:one: 1\n:two: 2\n"
new_hash = YAML.load(y_hash)
# => {:colors=>["red", "green", "blue"], :one=>1, :two=>2}
new_hash == hash
# => true

```

The `YAML.load` method can take either a string or an IO read handle as its argument, and deserializes the string or stream from YAML format into actual Ruby objects.

In the meantime, while objects are in serialized string form, you can edit them directly. In other words, YAML gives you a way to save data and also edit it in plain-text, human-readable form. (Part of the incentive behind the creation of YAML was to provide a plain-text format for representing nested data structures that wasn't quite as visually busy as XML.)

Problem

You need a way to automate the storage and retrieval of professional and personal contacts—an address book—but you want it to be in plain text so that you can edit the entries in a text editor as well as altering them programmatically.

Solution

We'll write some code that uses YAML, together with simple file IO operations, to provide a programmatic interface to a plain text file containing contact entries.

First things first: let's start with a test suite. Aside from the merits of writing tests in general, this will allow us to write examples of how the code should be used, before we've even written it. There's no better way to describe how you want an API to work than to write some tests that put it to use.

Listing 9.1, which you can place in a file called `contacts_y_test.rb` (the 'y' means this is for the YAML implementation), shows the class declaration and setup method for an appropriate test suite. The API for the contact code has already started to take shape. We'll create two classes: `Contact` and `ContactList`.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

The initializer for **Contact** will take the contact's name as the sole argument, and will yield the new **Contact** instance back to the block, where it can be used to set more values.

Listing 9.1: Class declaration and setup method for testing the contact code

```
require "test/unit"
require "contacts_y" #1
class TestContacts < Test::Unit::TestCase
  def setup
    @filename = "contacts"
    @list = ContactList.new(@filename) #2

    @contact = Contact.new("Joe Smith") #3
    joe.email = "joe@somewhere.abc" #4
    joe.home[:street1] = "123 Main Street" #5
    joe.home[:city] = "Somewhere"
    joe.work[:phone] = "(000) 123-4567"
    joe.extras[:instrument] = "Cello"

    @list << @contact #6
  end
end
```

In addition to test/unit, we'll load what will eventually be our implementation file: **contacts_y.rb** #1. After the loading preliminaries, a contact list gets instantiated along with a filename (#2), and a contact with a name. (#3). Beyond the name, the contact has an email address (#4), and several apparently deeper, hash-like data structures: home, work, extras. #5. The contact list object itself appears, not surprisingly, to have an array-like interface, judging by the appearance of the append operator. #6

Now it's time to write some tests for business logic. The setup up method inserts one contact object into the list. What about retrieving an object?

Listing 9.2: Testing the removal of a Contact object from a ContactList object

```
def test_retrieve_contact_from_list #1
  contact = @list["Joe Smith"]
  assert_equal("Joe Smith", contact.name)
end

def test_delete_contact_from_list #2
  assert(!@list.empty?)
  @list.delete(@contact.name)
  assert(@list.empty?)
end
end
```

Listing 9.2 shows a method that does exactly that. #1 It also includes a method that removes a contact from the list. #2 These two methods go in the test file, after the setup method. We'll also close out the class, so that we can write the implementation and get the test to pass.

Let's start with the **ContactList** class. We'll give each **ContactList** instance an array, in which it will store the actual contact objects. The business of the **ContactList** class itself will consist mostly of deciding what, and when, to pass along to this array: inserting contacts, removing contacts, and of course persisting contacts to a YAML file and reading contacts from a file.

The initial implementation of `ContactList` is shown in Listing 9.3. (We're not using YAML yet, but we'll need it so it's being loaded.) Most of the action is in the `@contacts` array, which is expected to contain `Contact` objects. Listing 9.3 can be saved to `contacts_y.rb`.

Listing 9.3: Initial implementation of the ContactList class

```
require "yaml"
class ContactList
  attr_accessor :contacts

  def initialize(file)          #1
    @file = file
    @contacts = []
  end

  def <<(contact)              #2
    @contacts << contact
  end

  def delete(name)             #3
    @contacts.delete_if {|c| c.name == name }
  end

  def empty?                   #4
    @contacts.empty?
  end

  def size                     #5
    @contacts.size
  end

  def [](name)                 #6
    @contacts.find {|c| c.name == name }
  end
end

<#1 Create a new ContactList object, storing file name and creating @contacts array >
<#2 Add a contact to list, via @contacts array >
<#3 Delete a contact from list >
<#4 Test for emptiness, via @contacts array >
<#5 Size of list is size of @contacts array >
<#6 Fetch a contact by doing a name lookup in @contacts >
```

The `@contacts` list fields requests for array-like operations. Some of these operations work the way they do out of the box for any Ruby array (`empty?`, `size`, and `<<`). Others require special implementation to make sure they do the right thing for a list of contact objects; note that `remove` and `[]` use a name lookup to figure out which contact you want to operate on.

Next comes the implementation of the `Contact` class, which will be responsible for storing the contact information itself. We want to be able to store separate contact info for home and work. Each of these data sets will be stored as a hash, and accessed as an attribute of the contact object. The name and email properties will be separate, stored as individual attributes rather than parts of any of the hashes (see Listing 9.4).

Listing 9.4: The Contact class

```
class Contact
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```

attr_reader :name, :email, :home, :work, :extras
attr_writer :name, :email
def initialize(name)
  @name = name
  @home = {}
  @work = {}
  @extras = {}
end
end

```

Now, run the tests:

```

$ ruby contacts_y_test.rb
Loaded suite contacts_y_test
Started
..
Finished in 0.000594 seconds.

2 tests, 3 assertions, 0 failures, 0 errors

```

Now let's get to the YAML side of things. We want a **ContactList** object to know how to save itself, in YAML format, to a file; and we want the **ContactList** class to know how to load a YAML file into a new **ContactList** instance. Listing 9.5 shows a test for these functions; you can paste this test into the existing test class.

Listing 9.5: Saving and loading a ContactList object

```

def test_save_and_load_list
  @list.save
  relist = ContactList.load(@filename)
  assert_equal(1, relist.size)
  contact = relist["Joe Smith"]
  assert_equal("Joe Smith", contact.name)
end

```

To get these new assertions to succeed, we have to add **save** and **load** methods to the **ContactList** class, as shown in Listing 9.6.

Listing 9.6: Second set of methods for the ContactList class

```

def save
  File.open(@file, "w") do |fh|
    fh.puts(@contacts.to_yaml)          #1
  end
end

def self.load(file)                    #2
  list = new(file)
  list.contacts = YAML.load(File.read(file)) #3
  list
end

```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

These two methods can be pasted into the class definition for `ContactList`. (Note that `load` is a class method, hence defined directly on the class object `ContactList` (represented in context by "self". [#2])

It's in the `load` and `save` methods that you see the use of YAML. And it's very simple: When you want to save the list, you convert its `@contacts` array to YAML, and print the resulting string to the file. #2. When you want to read the list in, you use the `load` class method of YAML, passing it a string consisting of the contents of the file. #3 (You can also pass it an open `File` object.)

The new tests pass. And so, with rather little fanfare, we have data persistence. And running the tests will create a file called "contacts", containing the YAML representation of the Joe Smith contact. This brings us to the other side of the YAML coin: the ability to edit the YAML file itself. You do have to be a bit careful, because the YAML specification has rules you have to follow. But as long as you follow the YAML rules, you can make as many changes as you want to the file between reads.

Listing 9.7 shows the contacts file resulting from running the tests for the contact classes.

Listing 9.7: The contacts output file, in YAML format

```
---
- !ruby/object:Contact
  email: joe@somewhere.abc
  extras:
    :instrument: Cello
  home:
    :city: Somewhere
    :street1: 123 Main Street
  name: Joe Smith
  work:
    :phone: (000) 123-4567
```

You don't want to touch the first two lines, which are YAML's business. But you can change the values of the strings, or add more data, and all your changes will be happily absorbed into the in-memory `Contact` objects next time you use the `Contact` class.

Discussion

YAML provides easy serialization of objects to strings, and it's not much harder to save those strings to a file. Whether or not you decide it's technically correct to call this a database, it certainly has database-like properties. You don't *have* to edit your YAML files by hand; you can treat them as a black box. But it's nice to know that they're easy to edit.

Still, plain-text files are far from the only game in town. We'll look next at Ruby's API for GDBM, the GNU version of the Berkeley DB database system.

9.2 Using the (G)DBM API

Ruby ships with wrappers for the DBM, GDBM, and SDBM database libraries. These libraries are a family, of which the original member is DBM, the Berkeley database manager. GDBM is the GNU version; SDBM is a public-domain version based on the earlier NDBM (*New Database Manager*, a successor to DBM). We'll focus on GDBM here, though everything in the examples should work with any of the three *DBM libraries included with Ruby.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

Problem

You want a simple contact manager, and you need to share the files with someone who may not have access to YAML.

Solution

Our hypothetical YAML crisis provides a chance for a look at a GDBM-based solution. We'll aim for something that's as close as possible to the YAML version of the contact manager—and the best way to guarantee that closeness is to use a similar test suite.

The tests are shown in Listing 9.8. Note that we're loading "contacts_g", implying that the implementation of the two classes will be in `contacts_g.rb`

Listing 9.8: The contact application tests

```
require 'test/unit'
require 'contacts_g'

class GDBMTest < Test::Unit::TestCase
  def setup
    @list = ContactList.new("gdbm_contacts")
    @contact = Contact.new("Joe Smith")
    @list << @contact                                     #1

    @contact.home["street1"] = "123 Main Street" #2
    @contact.home["city"] = "Somewhere"
    @contact.work["phone"] = "(000) 123-4567"
    @contact.extras["instrument"] = "Cello"
  end

  def test_retrieving_a_contact_from_list
    contact = @list["Joe Smith"]
    assert_equal("Joe Smith", contact.name)
  end

  def test_delete_a_contact_from_list
    assert(!@list.empty?)
    @list.delete("Joe Smith")
    assert(@list.empty?)
    assert(@list.contact_cache.empty?)
  end

  def test_home
    contact = @list["Joe Smith"]
    assert_equal("123 Main Street", contact.home["street1"])
  end

  def test_non_existent_contact_is_nil
    assert_equal(nil, @list["Some Person"])
  end

  def teardown
    @list.delete("Joe Smith") if @list["Joe Smith"]
  end
end
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

end

The test suite for the GDBM implementation is close to the one for the YAML implementation, but there are a few differences. Some tests have been replaced; these are just to show you some of what you might want to do for GDBM specifically (neither test suite is exhaustive). It's now necessary to add the contact to the list *before* setting any of the contact's properties. #1 The reason for this is that the hash-to-database magic only works if the contact's components—home, work, extras—are GDBM filehandles. And that will only happen when the contact objects become part of a `ContactList`; it's the contact list object that knows where the directory of GDBM files is.

Another tweak is that all keys are now strings, instead of symbols. #2 GDBM doesn't like symbols; it wants everything to be strings. Nonetheless, the goal of preserving the API in the GDBM reimplementaion has been largely met—assuming we can get the tests to pass. On now to the implementation itself.

The way GDBM works is very different from the way YAML works—definitely more of a real database facility, but more simple in terms of the kinds of data structures it can store. GDBM stores key/value pairs. The Ruby `gdbm` extension is programmed almost exactly like a hash. Here's a simple example of using GDBM. First, run this code from a file. (It will create a file called `movies.db`, so make sure you're not clobbering one!)

```
require 'gdbm'

movies = GDBM.new("movies.db")
movies.update(
  { "Vertigo" => "Alfred Hitchcock",
    "In a Lonely Place" => "Nicholas Ray",
    "Johnny Guitar" => "Nicholas Ray",
    "Touch of Evil" => "Orson Welles",
    "Psycho" => "Alfred Hitchcock",
  })

movies.close
```

(Notice the use of the update method, which is familiar as a hash operation.) Now, go into irb and do this:

```
>> require "gdbm"
=> true
>> movies = GDBM.new("movies.db")
=> #<GDBM:0xb7ef12cc>
>> movies.values.uniq
```

You'll get a list of all the directors in your database:

```
=> ["Orson Welles", "Nicholas Ray", "Alfred Hitchcock"]
```

The GDBM API is designed to be as hashlike as possible, with all the hash operations automatically writing to, or reading from, the database file.

The contact data we're storing isn't a simple hash, though. A contact has a name and an email address, which are just strings and could serve as hash keys—but the additional information, such as the home data, comes in the form of nested or embedded hashes.

One way to use GDBM in a situation where you need hashes-within-hashes is to serialize the inner hashes with YAML, and store them as strings. There's nothing terrible about doing that; but since we've already

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

looked at a YAML solution, we'll do it a different way here. For our GDBM implementation of the contact list, the entire list will be represented by a *directory*, and each contact will have its own subdirectory. Inside that subdirectory, there will be individual GDBM files for each of the inner data structures: home, work, extras.

In the YAML implementation, a `ContactList` object was a kind of proxy to the list—the array—itsself. In the GDBM implementation, `ContactList` objects will be proxies to directories. It's desirable to keep the API as transparent as possible; as API users, we don't want to have to know the details of how the files are being handled. So all of that has to be encapsulated in the `ContactList` class.

The `Contact` objects can do their share of the lifting, too. A contact list is in charge of a whole directory, and a contact has a subdirectory. It makes sense, then, for the contact object to know the name of its directory.

Let's start this time with the `Contact` class. It's shown in Listing 9.9.

Listing 9.9: the `Contact` class for the GDBM implementation of the contacts library

```
class Contact
  COMPONENTS = ["home", "extras", "work"]      #1
  attr_accessor :name, :email, *COMPONENTS    #2
  attr_reader :dirname                         #3

  def initialize(name)                         #4
    @name = name
    @dirname = @name.gsub(" ", "_")
  end

  def components                               #5
    COMPONENTS.map {|comp_name| self.send(comp_name) }
  end

  def open
    COMPONENTS.each do |component|
      self.send(component + "=", GDBM.new(component)) #6
    end
  end

  def close
    components.each do |component|
      component.close unless component.closed? #7
    end
  end
end
```

One addition to the class is the `COMPONENTS` constant, (#1) which contains an array of strings corresponding to the nested containers in the contact objects. The point of having this constant is to encapsulate these names in one place. That way, both the contact and contact list objects can find all of them easily, and the names will not have to be hard-coded in specific methods. Moreover, as you'll see when we look at the `ContactList` class, each component is actually going to be the name of a GDBM database: one each for "home," "work," and "extras."

We want read-write attributes for name and email, and for each of the components. #2 Using the unary `*` operator on `COMPONENTS` has the effect of turning the array into a bare list, so that it's as if we'd written:

```
attr_accessor :name, :email, :home, :work, :extras
```

We also want a reader attribute for this contact's directory (#3), which will be a subdirectory of the master directory of the list to which we assign the contact.

The initialize method (#4) preserves the name (which is the actual name of the person whose contact information this is), and also stores the directory name. The creation of the directory name involves replacing spaces in the name with underscores. You can adjust this if you prefer a different "munging" algorithm, as long as the result is a valid directory name (and preferably a reasonably cross-platform one).

The components method (#5) provides a translation from the component names to the actual components. This, in turn, allows the close method to walk efficiently through all the components, performing a close operation on any that are not already closed.

Speaking of the `close` method: the Contact class provides both open and close methods. The open method creates a new GDBM object for each component (#6), assigning that new object to the relevant component attribute of the contact. Due to the way GDBM works, if there's already a file with the appropriate name (e.g., "extras"), it will be opened for reading and writing. Otherwise, it will be created. (There are some further subtleties to the way GDBM handles read and write access, but we'll assume for purposes of the contact list that it's private and only being accessed by one program at a time.)

The `close` method goes through the components — again, corresponding to the filenames in which the data is stored — and performs the GDBM close operation on each one. This terminates the database connection to each file.

Now, let's look at `ContactList`. The specifics of implementing this class derive partly from the way GDBM works, and partly from the way directories work. We want to be able to get a contact from a contact list:

```
contact = @list["Joe Smith"]
```

The list object is going to need to go to the directory, look for a directory with the right name, and load all the GDBM files into a contact object. Fair enough — but not very efficient. It might pay to keep a cache of contact objects on hand, so the search doesn't have to be repeated if the same contact is requested twice.

With that in mind, look at Listing 9.10, which shows the first segment of the `ContactList` class including the `initialize` method, the `[]` method (to retrieve a contact object by name), and a helper method called `populate_contact`.

Listing 9.10: The beginning of the `ContactList` class

```
require 'gdbm'
require 'fileutils'                                     #1

class ContactList
  attr_reader :contact_cache                           #2
  def initialize(dir)                                  #3
    @dir = dir
    @contact_cache = []
  end

  def [](name)
    contact = @contact_cache.find {|c| c.name == name }  #4
    return contact if contact
    contact = Contact.new(name)                           #5
    Dir.chdir(@dir) do
      if File.directory?(contact.dirname)                 #6
        populate_contact(contact)                         #7
        @contact_cache << contact if contact              #8
      end
    end
  end
end
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```

        else
            contact = nil
        end
    end
    contact
end
#9

def populate_contact(contact)
    Dir.chdir(contact.dirname) do
        contact.open
    end
end
#10
end

```

(The `fileutils` extension is loaded (#1) for the sake of one or two utility methods to be used later.)

Upon initialization, (#3) the list stores its directory and creates an array which will serve to cache contact objects. That array is available as a `reader` attribute. (#2) When you try to retrieve a contact from the list, the list object first checks the cache. #4 If the contact is there, it returns it. If the contact isn't there, the real fun starts.

First, a new content object is created. #5 Then, the list object switches to its own directory, where it looks for a subdirectory whose name is the same as the directory name reported by the contact object — "Joe_Smith" for the contact "Joe Smith," for example. #6 If such a directory exists, then the contact's components get initialized to a new GDBM database object, based on the component name #7. The contact is now added to the list's content cache, (#8) so that the whole directory-based instantiation won't have to be repeated during this session. If, however, the directory corresponding to the requested component name does not exist, contact is reset to nil. This indicates a request for a non-existent contact. (Remember, you're just trying to fetch an existing contact, not create a new one.)

Finally, the value of the variable `contact`—which is going to be either nil or a GDBM object—is returned from the method. #9 If it's `nil`, you know that the contact you requested does not exist on this list. The utility method `populate_contact` drops down from the top directory of the list, to the specific subdirectory for this contact. #10 It then calls the "open" method on the contact—which, as you'll recall, calls `GDBM.new` on each of the contacts components, creating database handles on the relevant files.

Now let's look at adding an object to a list — and removing one from the list. We're shooting for the same API as the YAML version:

```

@list << contact      # add a contact
@list.delete(contact) # delete a contact

```

As Listing 9.11 shows, these operations too require attention to the filesystem and directory structure.

Listing 9.11: Adding and removing a contact

```

def <<(contact)
    Dir.chdir(@dir) do
        Dir.mkdir(contact.dirname) unless File.exists?(contact.dirname) #1
        populate_contact(contact)
    end
    @contact_cache << contact
end
#2

def delete(name)

```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```

    contact = self[name]
    return false unless contact          #3
    contact.close                        #4
    Dir.chdir(@dir) do
      FileUtils.rm_rf(contact.dirname)  #5
    end
    contact_cache.delete_if {|c| c.name == name } #6
    true                                #7
  end
end

```

Adding a contact involves creating a new directory, unless one exists (#1) (which is possible; there could be a contact object with this directory, even if it's not part of a list). It also involves going into the directory and populating the contact's components based on the GDBM files in the directory, creating them as necessary or just opening them for reading if they're already there. Finally, we add the contact to the list's contact cache #3.

Removing a contact involves several steps. First, we only want to delete contacts we actually have, so the method returns **false** if the contact is unknown. #3 Second, we ask the contact to close itself, which means walking through the components and closing each GDBM connection in turn. #4 Then we use the **FileUtils.rm_rf** method to delete the contact's directory (#5); remove it from the cache if it's there (#6); and return true to indicate a successful deletion.

Arguably, directory removal is a rather harsh step; it means that the information is really gone, not just hidden from the list. The reasoning, though, is that the list is really in charge of the contact, and that in practice, we'll always access contacts through their lists. You can of course soften the code, if you want to devise a way to keep the contact but remove it from the list—or a way to copy contacts from one list to another.

We can now add, remove, and retrieve contacts from the list. We just need a couple of query methods: **names** (all the names of all the list's contacts), **empty?**, and **size**. Listing 9.12 shows the remaining code necessary to complete the **ContactList** class.

Listing 9.12: The remaining methods for the ContactList class

```

def names
  Dir["#{@dir}/*"]
end

def size
  names.size
end

def empty?
  names.empty?
end

end

```

This class, along with the **Contact** class, now gives us all the functionality we need to make our tests pass.

```

$ ruby contacts_g_test.rb
Loaded suite contacts_g_test
Started
....
Finished in 0.060656 seconds.

```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

4 tests, 6 assertions, 0 failures, 0 errors

The key, again, is the directory structure, which we have to walk through explicitly but which adds an extra level of nesting so that a Contact object can present us with what amounts to a collection of hashes.

Discussion

The GDBM version of the contact manager is a bit more sprawling, in terms of file space and directory structure, than the YAML version—and GDBM files, unlike YAML files, are not human readable. But GDBM is a real database tool, not a different tool (data serialization, in the case of YAML) being recruited for database-like operations. In practice, that means that GDBM is optimized for database operations. Furthermore, having the entries spread out over many file directories means that you don't have to read them all in at the same time.

If you're just dealing with a few contacts—or a small amount of data in whatever domain you're writing this kind of code for—it's not going to make any noticeable difference whether you load in all the data at once or traverse a directory looking for it. If you've got a lot of data, you'll probably hit a point where splitting the data out into many GDBM files will speed things up. Of course, as with all performance questions you have to try it out and measure for yourself. And if you're using these tools for relatively small operations, you'll probably find any of them perfectly acceptable in terms of performance. That's part of their merit: they're easy to write and, in most cases, fast enough in operation.

One interesting programming issue arising in connection with the GDBM implementation of the contact manager is the issue of how much functionality to embody in the `Contact` objects, and how much to delegate to the `ContactList` objects. Notice, for example, that a contact does not know how to populate itself, in the sense of opening connections to all of its GDBM files. The list handles that, through the `populate_contact` method. Would it be better to teach a contact how to populate itself, and then have the list call:

```
contact.populate
```

instead of calling its own `populate_contact` method? You could certainly make a case for that; it puts the knowledge inside the object that's nearest to the operation. On the other hand, the contact list is really what we're modeling here; in a sense, the contact class only exists as a convenient way to split out some of the list's functionality. If that's true, then how *much* functionality we split out is pretty much discretionary. And there's certainly no harm in experimenting with different ways of doing the same thing.

We'll turn next to the Ruby MySQL module, which takes us into the realm of relational databases.

9.3 The MySQL driver

We've now got enough usable code for manipulating contacts and contact lists that we can put it to some use. This will also allow us to delve into the world of relational databases and the available APIs for Ruby.

Overall, the story is that there are individual drivers for MySQL, PostgreSQL, sqlite, and others. Any of these can be programmed individually. On top of these, you can also use the DBI (Database Interface) package. DBI provides an abstract API which can serve as a frontend to any of the DBD (database driver) packages available.

We'll look at a use case for the “pure” MySQL driver. Later we'll do something with DBI. Unlike the YAML and GDBM examples, these examples will not involve writing an API for a contact list but, rather, moving contact list data around: *to* a MySQL database first, and *from* a MySQL database second.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

9.3.1 Migrating data from YAML to MySQL

It's not uncommon to use database tools in situations where you're moving data from one storage system to another. The nice thing about the facilities available in Ruby is that you can do quite a lot of this at a rather high level of abstraction.

Problem

You've got a YAML file of contact information, and you want to store it in a relational database.

Solution

Let's tackle this problem by using the MySQL driver. This driver is available as a gem, or as a non-gem Ruby package. To install it as a gem, simply run the following:

```
gem install mysql
```

To get started, take a look at Listing 9.13, which shows several contact objects in the YAML format.

Listing 9.13: Several contact objects in YAML

```
--- !ruby/object:ContactList
contacts:
- !ruby/object:Contact
  extras:
    :sport: bowling
    :car: Toyota
    :pets: armadillo

  home:
    :postal: "12345"
    :state: NJ
    :country: USA
    :street1: 123 Main
    :city: Somewhere
  name: David Black
  work: {}
  email: dblack@somewhere

- !ruby/object:Contact
  extras: {}

  home:
    :postal: "23456"
    :state: AB
    :country: USA
    :street1: 234 Main
    :city: Somewhere
  name: David Smith
  email: dsmith@somewhere
  work:
    :company: The Somewhere Consultants
    :street1: 234 Main
    :street2: Suite 33943
    :city: Somewhere
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```

      :postal: "23456"
      :state: AB
      :country: USA
- !ruby/object:Contact
  extras:
    :instrument: violin
    :car: Honda
    :pets: cat
  home:
    :postal: "00000"
    :state: US
    :country: USA
    :street1: 9393 WEst Main
    :city: Nowhere
  name: Joe Smith
  email: jsmith@somewhere
  work: {}

- !ruby/object:Contact
  extras: {}

  home:
    :postal: "98765"
    :state: HH
    :country: USA
    :street1: 8 North Main
    :street2: Apt. 3
    :city: Anywhere
  name: John Smith
  email: jsmith2@somewhere
  work: {}

```

The first step in migrating these contacts to a MySQL database is designing and creating the database itself. We'll use the contact's email address as the primary key for the contacts table, and as the foreign key for all the other tables. Listing 9.14 contains SQL instructions suitable for creating this database in MySQL. Running Listing 9.14 as a MySQL script, with appropriate permissions, should create the database.

Listing 9.14: SQL instructions for creating the contacts database

```

drop database contacts;
create database contacts;
use contacts;

drop table if exists contacts;
create table contacts (
  name varchar(100),
  email varchar(50),
  primary key (email)) ENGINE=INNODB;

drop table if exists home;
create table home (
  street1 varchar(100),
  street2 varchar(100),

```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```

    city varchar(50),
    postal varchar(20),
    state varchar(20),
    country varchar(25),
    contact_email varchar(50),
    foreign key(contact_email)
        references contacts(email)
        on delete cascade) ENGINE=INNODB;

drop table if exists work;
create table work (
    company varchar(100),
    street1 varchar(100),
    street2 varchar(100),
    city varchar(50),
    postal varchar(20),
    state varchar(20),
    country varchar(25),
    contact_email varchar(50),
    foreign key(contact_email)
        references contacts(email)
        on delete cascade) ENGINE=INNODB;

drop table if exists extras;
create table extras (
    label varchar(50),
    description varchar(150),
    contact_email varchar(50),
    foreign key(contact_email)
        references contacts(email)
        on delete cascade) ENGINE=INNODB;

grant all on contacts.* to 'contacter'@'localhost' \
    identified by 'contactme'
```

The last command (**grant**) gives all the necessary privileges to the “contacter” user. You can of course make up your own username and password for that command. The main thing is to create a non-root user, since the username and password are going to appear in a plain-text program file. (Not that you'd be handing the file around anyway, but it's always better not to put very important passwords in plain text anywhere.)

The next step is to create the “glue” program that will take us from YAML to MySQL. This program is going to read the YAML file in—which will create a **ContactList** object containing several **Contact** objects—and then write it out again, this time to the MySQL database. The input, however, isn't entirely congruent with the output; we have to do a little work to make it fit.

Contacts have home and work components. Those components map very easily from a hash (the structure by which they're represented inside the contact object) to a database table; in fact, the keys of the home and work tables in the database schema are simply lifted from the key names in the home and work hashes.

The “extras” component is a little different. Here, keys are arbitrary. That means they cannot match up to database column names, since those names have to exist in advance. (It's possible to create columns on the fly using the MySQL API, but that's just asking for confusion and trouble.) Instead of storing each value in a column named for its key, the extras table has two storage columns: **label** and **description**. It also has a **contact_email** column, where the email address of the contact will be stored.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

What this means is that while there will be one record in the home table, and one in the work table, for each contact, there will be one record in the extras table for every entry in the contact's extras hash. The key/value combination in the hash becomes the label/description combination in an extras record.

Most of the rest of the work of writing the program involves creating query strings and shipping them off to the database, via the MySQL API. Each contact object will trigger a three-part cycle of insertions:

The query that creates the entry in the contacts table itself

The loop that creates the entries in the home and work tables

The loop that creates the entries in the extras table

First things first: before any of this happens, we need to load in the necessary libraries, as well as our application code (`contact_y.rb`), and make connections on both the YAML side and the MySQL side:

```
require 'yaml'
require 'mysql'
require 'contact_y'

conn = Mysql.new("localhost", "contacter", "contactme", "contacts") #1
list = YAML.load(File.read("contacts.yml")) #2
```

The assignment to `conn` (#1) gives us a new `Mysql` object — essentially an addressable handle on a database connection. Note the arguments given to `Mysql.new`: `host`, `username`, `password`, and database name. The variable `list` (#2) will contain the `ContactList` represented in the YAML file; this contact list, in turn, contains and manages the specific contacts.

Now let's take care of the first of the three major stages: creating the entry in the contacts table. Just for fun, here's how to do it using Ruby's `%` interpolation operator:

```
list.contacts.each do |contact|
  conn.query "INSERT INTO contacts (`name`, `email`)
    VALUES ('%s', '%s')" % [contact.name, contact.email]
```

The `%s` format specifiers work as they do in the `sprintf` method family: they serve as placeholders for a string-to-be-determined-later. The two necessary strings are provided in an array after the main string. Between the main string and the array comes another `%`-sign; this is what triggers the interpolation operation.

Amidst all this handy string manipulation practice, notice what's actually happening here: a call to `conn.query`. We're using `conn`, the database connection handle, to send a SQL query to the database.

Next up: the home and work components. For each of these, the goal is to send a query that will insert the right values in the right fields. The field names are the keys, plus the extra field name `contact_email` — the primary key column in the home and work tables, which has no corresponding value in the YAML structure.

We want to drill down, then, from the contact to its components; then we want to isolate each component's keys, and create a query string that uses them as field names, together with the component's values as the database values. Here's the code that will do this:

```
%w{ home work }.each do |component|
  data = contact.send(component) #1
  items = data.keys #2
  fields = (["contact_email"] + items).map {|field|
    "`#{field}`"
  }.join(",") #3
  values = [contact.email] + items.map {|field| data[field]} #4
  values.map! {|value| "'#{value}'"} #5
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```

        values = values.join(",")
        conn.query("INSERT INTO #{component} (#{fields}) VALUES (#{values})")
#7
    end

```

The object `data` is the entire component hash; it's acquired by sending the name of the component ("`home`" or "`work`") to the contact object. #1 Then we isolate the keys — which will be things like "`street1`", "`street2`", "`phone`" — as "`items`". #2 If we add the string "`contact_email`" to this list of items, we'll have all the necessary column names for the relevant database table. (There's no "`contact_email`" field in the YAML version, so we have to shoehorn it in to please the MySQL database schema.) These field names get inserted into backticks and strung together with commas. #3 That will give us something like this for the `fields` variable:

```
`contact_email`, `street1`, `street2`, `city`, etc.
```

Those are the fields we'll be inserting values into. The values themselves come from a similar — but not identical — mapping of the actual value for `contact_email`, plus the actual values for the items (as retrieved from the data hash). Here, we put the array together first, because that's a somewhat longer operation in this case. #4 Then the array gets mapped — `map!`ped, actually; the values get changed in place — and then joined with commas (#5) to make a values string out of the array.

The result will be something like this:

```
'jsmith2@somewhere', '123 Main', '', 'Somewhere', etc.
```

Thus the values string lines up nicely with the fields string, ready for insertion into the appropriate table. #7

You may be wondering why it's necessary to march through all the keys and dig out all the values for each hash, when there's a `values` method that would do the same thing in one step. The reason is order — or, rather, lack thereof in a hash. Hashes are unordered, which means that the order in which their entries are returned is not guaranteed. While it's very likely that the keys, when returned separately via the `keys` method, will line up with the values as returned by the `values` method, it's a good idea nonetheless to go ahead and retrieve each value explicitly, so as not to depend on hash ordering in any guise.

ENCAPSULATING COMPONENT REFERENCES

You'll recall that for purposes of the GDBM contracts implementation, we neaten things up by setting a `COMPONENTS` constant inside the `Contact` class, so that it wasn't necessary to hard-code the names of the components wherever they were used. There are a couple of reasons it's probably not worth doing that here. First, the "extras" component has to be split out anyway. Second, the remaining components have to correspond exactly to table names in the database. It is, however, possible to query the database for its table names; and if you subtract "extras" and "contacts" from the list, you'll have (as it's currently engineered) the names of the other components. Still, while it's great to automate things as much as possible, you'll probably find that with more or less one-time conversion scripts, it's inevitable that certain things are going to have to be hard-coded and in some cases require individual treatment (like "extras").

The handling of "extras" comes next. It's actually a bit less involved than the handling of the home and work components, though it does potentially create more records:

```
contact.extras.each do |label, description|
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```

    conn.query("INSERT INTO extras (`contact_email`, `label`,
`description`) VALUES ('#{contact.email}', '#{label}',
'#{description}')"")
  end
end

```

Each entry in the extras hash gets an entry in the database, with its foreign key ("contact_email") set to the current contact's email field. And that last **end** tells you that the big loop—the loop through all the contacts—is finished.

Discussion

The MySQL API may seem a bit raw to you, especially if you're used to a full-blown object-relational mapper like ActiveRecord or Og. When you use the API directly, you write a lot of SQL yourself; and that may or may not be to your taste.

Keep in mind, though, that having the programming interface available can make a big difference. It gives you all the power of Ruby in front of your SQL generation. You can, of course, grep through a YAML file and try to piece together queries yourself...but all you'll learn by doing that is how much easier it is when you have a programmatic layer, even if it's not an ORM.

If you do have, say, an ActiveRecord-friendly database, then you can of course use the higher-level commands available to you to handle the data more abstractly. It all depends on what your starting point and goal happen to be in a given project.

One thing that isn't covered in the solution above is error-handling. Our script will simply die if something goes fatally wrong in a database operation—and that, of course, could leave the database in an inconsistent state. To prevent this, you can wrap each contact's set of queries up as a transaction.

To do this, you first need to turn off auto-commit mode:

```

...
conn = Mysql.new("localhost", "contacter", "contactme", "contacts")
list = YAML.load(File.read("contacts.yml"))

```

```

conn.autocommit(false)

```

Now you need to put each contact's worth of querying inside a begin/rescue/end block, where rescuing from a Mysql::Error will result in a rollback:

```

list.contacts.each do |contact|
  begin
    # rest of code, through the "extras" loop

  rescue Mysql::Error => e
    puts "Problem with contact #{contact.name}: #{e}"
    conn.rollback
  next
end

conn.commit
end

```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

The commit only happens if no `Mysql::Error` gets raised along the way. Doing it this way will help you avoid getting the database into an inconsistent state (e.g., a contact record getting saved but the home record not) and will also avoid stopping the whole program just because of one malformed record.

So far we've looked at sending insert queries to the database—which is important, but not the whole story. Our next problem will take us in the other direction: starting from a relational database, and reading data out in order to store it in a different format. Moreover, we'll take this opportunity to turn the corner to DBI, the high-level Ruby database interface library.

9.4 Using DBI, the high-level database interface

DBI ships as a separate library, and provides a high-level interface that allows you to come about as close as you can to programming different relational databases with the same programming tools. The goal is to abstract away the differences between various databases, and focus on what they have in common. At the same time (though we won't pursue it in detail here), DBI can also take advantage of features that one database but others don't.

DBI rests on top of, and needs, one or more database driver (DBD) packages, such as the MySQL library we used in the last example. At time of writing, DBI supports twelve DBDs (plus one deprecated one). If you learn how to use DBI, it's almost one-stop shopping for database APIs.

Problem

You need to read out some contact records from a relational database and save them to a GDBM database.

Solution

We're going to use the same MySQL database as in the last example, but this time we'll address the database with DBI. The goal is to migrate the data to the GDBM-style contacts database. That means we'll need to load `contacts_g.rb`, as well as DBI. Loading `contacts_g.rb` will in turn cause the GDBM driver to be loaded.

```
require 'dbi'
require 'contacts_g'
```

Now we need a `ContactList` object, which will serve as the receptacle for the data coming in from the MySQL database — massaged, of course, so as to fit the shape of `Contact` objects. (Make sure that the directory you use already exists, or add something to the script to create it.) We also need a database connection. Finally, we need a list of all the tables in the database, *except* contacts and extras; those two tables require special handling.

```
list = ContactList.new("migrated_contacts")
conn = DBI.connect('DBI:Mysql:contacts', 'contacter', 'contactme')
tables = conn.select_all("show tables").flatten - %w{ extras contacts }
```

The first argument to `DBI.connect` is the name of the database, contacts, qualified with `DBI::Mysql` to point DBI to the correct driver. Note that the separators here are single colons, not the double-colon operator that indicates nested constants in Ruby. The whole thing is just a string; none of the characters have significance until they're scanned and interpreted by DBI itself. The host will default to `localhost`.

To get the list of tables, we use `select_all` to return an array of rows. In this case, each row is a table name. Rows are returned as arrays inside arrays; flattening the array takes out the nesting. Finally, we remove

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

"extras" and "contacts," so that we're left with the ones we want to operate on in a loop. And now: the loop itself. Listing 9.16 shows the loop through the rows in the contacts table.

Listing 9.16: The loop through the contacts table

```
conn.select_all('select name,email from contacts') do |row|      #1
  name,email = row
  where_clause = "where contact_email = '#{email}'"             #2
  contact = Contact.new(name)
  list << contact                                               #3

  tables.each do |table|
    result = conn.execute("select * from #{table} #{where_clause}") #4
    record = result.fetch_hash                                  #5
    record.delete_if {|k,v| k == "contact_email" or v.nil? }     #6
    contact.send(table).update(record)
  end

  extras = conn.execute("select label,description \              #8
                        from extras #{where_clause}")
  extras.each do |label,description|                             #9
    contact.extras[label] = description
  end
end

<#1 Get an array of relevant rows >
<#2 Store where clause for convenience >
<#3 Put new contact in the list >
<#4 Request the relevant row for current table (home, work) >
<#5 Fetch the row as a column/value hash >
<#6 Remove any nil values, and the foreign key, from the hash >
<#7 Transfer the result hash to the relevant component of the contact >
<#8 Retrieve the relevant extras record >
<#9 Transfer the extras data to the contact's extras hash >
```

The table names will match (because we designed the database this way) the relevant components of the GDBM contact: specifically, as currently implemented, home and work.

As with the previous script, where we went from YAML to MySQL, we're dealing here with three major phases: creating the **Contact** objects, handling the miscellaneous tables, and handling the extras table. Creating and storing a new contact is easy (#3), thanks to the API we've already developed for the GDBM contacts database. Using DBI is, in general, also easy. After getting hold of all the contact rows (#1), we store a **WHERE** clause that we're going to use repeatedly (#2), just to save having to type it out.

Then we go through the list of tables. For each table (remembering that we've subtracted "extras" and "contacts" from the list), we execute a query returning the relevant row for this contact #4. This request returns an object of class **DBI::StatementHandle**; this object takes requests for actual delivery of the rows. The relevant request, here, is **fetch_hash** (#5), which returns the next row in the form of a hash of column names against values. (A certain amount of error-checking has been left to the reader, including making sure that there is only one row returned per table for each contact!) We then cleanse the hash of the irrelevant **contact_email** key, as well as any entries with **nil** values (#6) (which GDBM doesn't like; it only wants strings, and there's not much point storing empty strings for nils). The hash of columns and values can now be transferred to the relevant component of the contact. (#7) The hash update method is effective because the component is actually a GDBM object, and GDBM objects are programmed like hashes.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

Next comes the "extras" special case. Here we get all the extras records belonging to this contact (#8), and make the appropriate assignments in the extras component. #9

Discussion

DBI is a great tool for addressing a variety of database systems. It shares with the individual drivers the fact that it keeps you fairly close to the SQL; like the MySQL driver itself, DBI is not as abstracted or high-level as ActiveRecord or Og. But even if you're writing the statements mostly by hand, the ability to embed them inside Ruby logic and string-handling is extremely valuable.

In the MySQL-to-GDBM example, we see again the special casing of the extras table. Going from YAML to MySQL, the special casing involved the need to create a separate record for each key/value pair in the extras hash, using the column names "label" and "description," because the keys for extras are arbitrary and it was impossible to have a separate column name for each one. Going the other way—extracting the data *from* the MySQL database and putting it back into a Ruby hash structure (which is what GDBM objects essentially are), it's possible to shed the label/description wrapper and go back to having each label be a hash key and the corresponding description its value.

This kind of massaging of structure is fairly common when you migrate around the different data storage facilities available in Ruby (and, of course, not just in Ruby). The decisions you make about the shape of the containers are as important as any other feature of the storage process.

9.4 Summary

In this chapter we've looked at a representative set of database and data storage facilities in Ruby. We looked at YAML, the data serialization format, as a kind of database tool, having the particular dual merit of allowing for direct storage of Ruby data structures and offering a human-readable, editable text representation of the data it's storing. From there we went to GDBM, one of the drivers in the DBM Berkeley database family. GDBM objects are tied invisibly to files, and are programmed like Ruby hashes, making for another quick and easy data storage tool.

In both the YAML and GDBM, we developed library code for creating and managing a list of contacts. The rest of the chapter involved using this code in the service of trying out the relational database facilities available for Ruby. First, we undertook the task of migrating a YAML database of contact objects to a MySQL database; second, we migrated the same MySQL database back out, into GDBM contact-list format. Both of these projects were written "script-style" (no classes; just procedural code), and both shed some light on the process of formulating and implementing data structures across different storage systems.

We haven't covered everything in the Ruby database realm, by any means; but you've seen enough to give you a toolset for writing your own scripts, and enough to give you your bearings as you explore some of the other available tools. And you'll very likely find that these techniques have some very practical uses—if not every single day, then at critical junctures in data migration and reorganization projects. In the next chapter, we'll take these ideas further and look at handling structured data files more in depth.

10 Structured Documents

Almost any Ruby program you write will involve either loading in data from an external source or exporting information produced in your program to be reloaded later or loaded by another program. You might use a dead-simple representation like YAML or a more complex one like Atom, but the basic principles will remain the same.

While you'll have the ability to choose the data format for your configuration files or external storage, you will often run into situations where you will need to use data produced by someone else, often by programs written in other programming languages or even created manually by human beings. In these cases, you might need to be able to read in, and correctly interpret broken data files. The most common example of this is reading in HTML files from the Internet, which are frequently impossible to parse without first repairing the data.

Thankfully, almost any structured format you might come across has an associated Ruby library that will make reading in data for use by your program, or writing out information you've collected internally, a trivial matter. Some of these libraries, like the Hpricot library that we will discuss later, also specialize in fixing broken input before giving you a simple API to parse and manipulate it.

In this chapter, we'll look at XML, including specific forms like RSS, Atom and XHTML. We will also look at YAML, Ruby's built-in, simple serialization format, and CSV, which is commonly exported by programs like Microsoft Excel and Outlook.

10.1 XML in the Enterprise

The most common data interchange format is XML. It is used to encode all sorts of information from documents produced in Microsoft Word to the votes of members of Congress made available online. This is due, at least in part, to the fact that virtually all programming languages have extremely good support for XML, so it's virtually guaranteed that information exported by one program will be easily to interpret by any other. It also means that vendors will frequently use XML as a format for configuration files, because they have to do little or no custom work to read them in or write them out.

With that said, let's take a look at some typical XML usage, and how the writers of Ruby libraries designed to parse it want you to think about the problem.

10.2 Typical XML Usage

Because XML is such a common interchange format, there are a number of extremely common uses, like using XHTML for web pages, and RSS or Atom for online feeds. We'll get to those cases later, but another extremely common use of XML is for storing configuration for your application. Many of you will need to parse legacy data or configuration in the form of XML, and fortunately for you Ruby has a nice set of XML libraries to help you out.

For our first set of examples, we'll look at parsing the configuration for a fictitious calculator program that performs similarly to the calculator built into Apple's OSX operating system or Microsoft's Windows. Let's assume that the calculator supports two modes: standard and scientific, and that we'll store its starting position

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

in our configuration file. We will also store information about the decimal-point precision, as well as the number currently stored in memory, if any.

10.2.1 Using XML to Read in Configuration

Problem

You need to parse an XML configuration file.

Solution

For your calculator, you need to be able to read in the configuration details, and instantiate a Calculator object with the appropriate settings that were previously saved by the user. A sample configuration file for the calculator is provided as listing 10.1.

Listing 10.1: Our calculator XML

```
<?xml version="1.0" encoding="ISO-8859-15"?>
<calculator>
  <startup mode="standard" precision="2" />
  <memory type="float">16.24</memory>
  <keyboard type="Macintosh">
    <numeric-keypad enter="=" clear="C" />
    <max-fkey>F16</max-fkey>
  </keyboard>
</calculator>
```

This is a very simple configuration file, but with enough variance to explore the features of Ruby's built-in XML parser, REXML. Figure 10.1 shows the XML represented graphically as a tree of nodes.

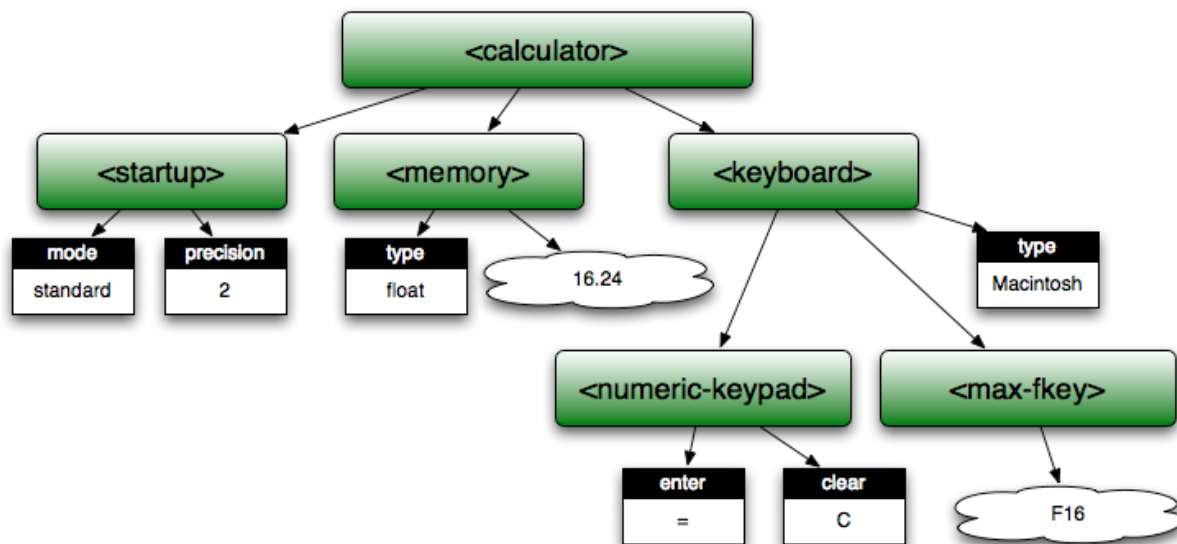


Figure 10.1A graphical representation of the calculator XML document. Element nodes are represented by rounded rectangles, attributes by squares, and text nodes by bubbles.

There are a number of libraries that allow you to parse XML. For this simple case, REXML, the XML parser that comes built-into Ruby, will more than suffice.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

Listing 10.2 Getting our XML into Ruby with REXML

```
require 'rexml/document'
include REXML

module Calculator
  class Config
    def initialize(memory, startup, keyboard)
      @memory = memory, @mode = mode, @keyboard = keyboard
    end
  end

  class Keyboard
    def initialize(type, numeric_keypad, max_fkey)
      @type = type, @numeric_keypad = numeric_keypad, [CA]
      @max_fkey = max_fkey
    end
  end
end

string = File.read("calculator.xml")
calculator = Document.new(string)
root = calculator.root
memory = root.elements["memory"]
memory = memory.text.send("to_#{memory.attributes['type']}")
startup = {:mode => root["startup"].attributes["mode"], [CA]
          :precision => root["startup"].attributes["precision"]}

keyboard = root.elements["keyboard"]
keyboard_type = keyboard.attributes["type"]
numeric_keypad = keyboard.elements["numeric-keypad"].attributes
max_fkey = keyboard.elements["max_fkey"].text

keyboard = Calculator::Keyboard.new(keyboard_type, [CA]
                                   numeric_keypad, max_fkey)

config = Calculator::Config.new(memory, startup, keyboard)
```

Discussion

In the example above, we started by producing a hypothetical API for our calculator configuration object. Because, in this example, you are producing both the application and the XML configuration to read in, you are free to create an API for your application that closely mirrors the XML format. In this case, we used the simple XML reading methods of REXML to extract the information we needed from the configuration file and populate our configuration object. In a real-world scenario, this configuration object would presumably be passed into the application when it was instantiated.

For simplicity, we have left out any error handling, but you would probably want to handle errors caused by incorrectly formatted XML more gracefully than simply throwing an error and exiting the application, as this example would.

Because this example is so simple, there are only a few things that need discussion. First off, the example uses a common Ruby trick to produce the correct value for the starting memory value. The XML format can

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

only return strings, but we wanted to be able to specify that the value in memory was some other type. In our case, we wanted to support integer or floats to be loaded in. Because Ruby allows conversion from string to numeric formats via `to_f` for floats and `to_i` for integers, we were able to provide a single character in the type attribute of memory, and then easily call the appropriate conversion function by using the `send` method.

Also, REXML allows you to use XPath selectors, and returns the appropriate node-type based upon the selector. For instance, when getting the precision, we used `root.elements["startup/@precision"]`. Because we used an attribute selector, the expression returned an attribute, which we could then get the value of. In other cases, we selected elements, and got elements back.

Another neat trick is to realize that the `attributes` method on an element returns a `Hash`. As a result, if you can structure your API to accept a Hash, you can simply pass in the result of the `attributes` method, as we did for `numeric_keypad`. This allows us to support an arbitrary (and even changing) set of options for the numeric keypad without having to change the way we parse out the configuration file.

10.2.2 Writing Your Configuration to Disk

Now that we've put together some code for reading in configuration from an XML file, what happens if you want to save changes to the XML? Unless you expect your user to hand-modify the document, you need a way to output any changes to the configuration back from whence it came.

Problem

You need to write an XML file to disk.

Solution

For the purposes of this example, let's just assume that we need to produce a function that will take a configuration object, as we defined it previously, and serialize it back into outputtable XML. Because we're doing more than just pulling in some data from a file, we'll get a bit more Object Oriented this time, and take the opportunity to write some tests proving that our function actually works.

In addition to creating a mechanism for outputting our configuration object to XML, we'll pull our input conversion into `Calculator::Config`. We'll create `to_xml` and `from_xml` methods on the `Config` class. To make things even more encapsulated, we'll create `to_xml` and `from_xml` methods on the `Keyboard` class, which will allow us to specify how we want keyboard specification objects to get saved and restored within the larger specification XML.

Listing 10.3 Reading and writing the configuration

```
require 'rexml/document'
include REXML

module Calculator
  class Config
    attr_accessor :memory, :startup, :keyboard
    def initialize(memory, startup, keyboard)
      @memory, @startup, @keyboard = memory, startup, keyboard
    end

    def self.from_xml(xml_file)
      string = File.read(xml_file)
      calculator = Document.new(string)
      root = calculator.root
      memory_el = root.elements["memory"]
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```

        memory = [CA]
memory_el.text.send("to_#{memory_el.attributes['type'].slice(0,1)}")

        startup = {"mode" => root.elements["startup"].attributes["mode"], [CA]
        "precision" => root.elements["startup"].attributes["precision"].to_i}

        keyboard = Calculator::Keyboard.from_xml(root.elements["keyboard"])

        return new(memory, startup, keyboard)
    end

    def to_xml
        doc = Document.new
        doc << XMLDecl.new(1.0, "ISO-8859-15")
        root = doc.add_element("calculator")
        root.add_element("startup", @startup)
        root.add_element("memory", "type" => [CA]
@memory.class.to_s[0].chr.downcase).add_text(@memory.to_s)
        root.add_element(keyboard.to_xml)
        doc
    end
end

class Keyboard
    attr_accessor :type, :numeric_keypad, :max_fkey
    def initialize(type, numeric_keypad, max_fkey)
        @type, @numeric_keypad, @max_fkey = type, numeric_keypad, max_fkey
    end

    def self.from_xml(keyboard)
        keyboard_type = keyboard.attributes["type"]
        numeric_keypad = keyboard.elements["numeric-keypad"].attributes.to_hash
        max_fkey = keyboard.elements["max-fkey"].text
        new(keyboard_type, numeric_keypad, max_fkey)
    end

    def to_xml
        el = Element.new("keyboard")
        el.add_attribute("type", @type)
        el.add_element("numeric-keypad", @numeric_keypad)
        el.add_element("max-fkey").add_text(@max_fkey)
        el
    end
end
end

```

Discussion

All of the code inside the new `from_xml` method should look very familiar, as it's basically just cribbed out of our previous example. As described earlier, however, we moved the logic dealing with the `Keyboard` into a `from_xml` method on the `Keyboard` object, which allows us to pass in the root `<keyboard>` XML node, and have it return a `Keyboard` object, which we can use to instantiate our specification object.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

Note that `Keyboard.from_xml` is a class method on the `Keyboard` class, just as `Config.from_xml` is a class method on the `Config` class. There's a slight incongruity between the two, in that `Config` takes a file name, while `Keyboard` takes a DOM node. This is just because of the way we expect to instantiate the objects: Configs will pull in an external file, while Keyboards will receive part of an existing DOM structure.

The `to_xml` function we wrote in the `Config` class is even simpler to understand than the original parsing function. First, we create a new XML Document. We then add an XML declaration (which should appear at the top of XML files, if we're following good form), and then the root element (`<calculator>`). The `add_element` function has a nice property that makes the next step very simple. It takes an element name as its first parameter, and an optional `Hash` object for any attributes you wish to instantiate. Well, holy attributes Batman, we already store the `startup` attribute of `Config` as a `Hash`, so we simply pass it in. Elegant, huh?

We're not quite as lucky with memory, because we have to do the reverse procedure that we used in `from_xml` to store the number and its type. For type, we simply slice off the first character of the memory object's class, downcased of course ("s" for String, "i" for Integer, and "f" for Float). For the value, which we store as text inside the `<memory>` node, we simply use `to_s` to stringify the value. The real trick, however, comes next, with our encapsulated `Keyboard` object.

As you probably remember, from the first problem above, we decided to keep the Keyboard information in its own object, which would encapsulate information about a Keyboard. At the time, there was very little utility in this decision, but when we made our code more Object Oriented, we got to further make use of this encapsulation. Since our `Config` object has a `Keyboard` object in its `keyboard` attribute, we can simply call `keyboard.to_xml` to get the appropriate XML for the keyboard.

Jumping down to `Keyboard#to_xml`, we see that the function is also pretty simple. We start by creating a new wrapper element (`<keyboard>`), and then adding the `type` attribute based upon the object's `@type` instance variable. You might have thought you could use the earlier shortcuts to add the attributes when you created the elements, but unfortunately `Element.new` does not support that shortcut. We then use the shortcut syntaxes we used above to finish up the element, and return it.

The takeaway from all this XML creation is that writing somewhat elegant REXML code requires a good knowledge of the chaining and shortcut possibilities. Because `add_element` returns the element in question, for instance, we can immediately throw an `add_text` on top of it. We can also use existing Hashes that we have created for our application for the attributes of any node, a trick we made extensive use of.

10.3 HTML and XHTML Made Easy with Hpricot

With the explosion of the web more than a decade ago, a tremendous amount of information is available on the Internet in HTML form. Unfortunately, that information is rarely, if ever, presented in fully valid, parsable form. Since you won't be able to control the creation of the data you'll be reading in, it will become important for us to have fast, dead-simple tools for taking broken markup and converting it into machine-readable content.

Fortunately, Hpricot, an XML parser written by Ruby luminary "why the lucky stiff" provides us with just that. Its most performance-related features are written in C, so it's very fast, and it uses CSS3 selectors (plus a number of custom CSS extensions), so finding precisely the nodes you want, which is very important in massive, markup-heavy HTML pages, is much simpler than in REXML. Finally, it was born out of the philosophy of the popular jQuery JavaScript library, so you'll find that certain common tasks, like adding a class to a series of elements in a document, become extremely simple.

In this section, we will explore processing HTML documents that you create, as well as reading it files from the wild.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

10.3.1 Post-Processing Your HTML

You use a Ruby web framework like Ruby on Rails, Merb, or Nitro, and have a number of pages that produce tables. You want all of the tables to have zebra-stripping (even rows should be colored light-blue) when your pages are rendered on the browser. You've already explored a few options, but none of them have met your needs.

You've tried using client-side JavaScript code, but some of your tables are quite large, and those pages can take minutes to load with this solution. You've considered manually ensuring that your tables are properly striped, but you have so many different mechanisms for creating tables that trying to manually ensure that the striping occurs has quickly become tedious. That said, it is not difficult for you to ensure that all tables that need to be striped have a specific class, say, "zebra."

Problem

You need to process and transform an HTML file.

Solution

We'll use Hpricot to post-process the rendered page and set an appropriate style on every other row. Then, we will use a CSS sheet to pull in the style. Ruby on Rails is used for this example, but the solution in general is framework agnostic. As long as you have access to the rendered response, you can use the technique outlined here to post-process your HTML.

Listing 10.4 A simple post-processing filter

```
class ApplicationController < ActionController::Base

  after_filter :zebra

  def zebra
    doc = Hpricot(response)
    doc.search("table.zebra tbody [CA]
tr:nth-child(even)").add_class("even_zebra")
    response.body = doc.to_s
  end

end
```

Listing 10.5: The associated CSS

```
.zebra_even {
  background-color: #ddf;
}
```

Discussion

First off, note that I kept the example very simple, just to demonstrate the technique at hand. Later in this chapter, we will explore some more features of Hpricot, which you will be able to use together with this technique to do more advanced post-processing of HTML data.

With that said, let's take a quick look at the example. You can ignore the Rails-specific semantics. All that we're doing is saying that the post-processing should occur for all pages in the application (by including it in ApplicationController), and specifying that we will do the post-processing via an `after_filter`. The real action happens in the `zebra` method, and you're probably surprised at how short the code is.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

The first line simply grabs the existing response body and parses it via Hpricot. Next, we find the rows we want to change. This might look a bit odd to you, but it's just a simple CSS3 selector. We find all tables with the class `zebra`, which is how we're going to specify that we want a table zebra-striped. Then, we'll jump down to the `<tbody>`, because we probably don't want to be striping our rows in the table head (if we have any).

Once we have the `tbody`, we get all the rows that are even, relative to their parent. We need to use `:nth-child` (rather than the `:even` selector supported by Hpricot) in case we have multiple tables on the page. If we simply used `:even`, if the first table ended on an odd row, the next table would have the striping skewed by one. Using `:nth-child` scopes the even-ness to the table in question. Once we've gotten the rows we need, we can use the `add_class` convenience method provided by Hpricot to add the `zebra` class to every matched row.

As we'll see later, Hpricot returns the `Elements` set that was matched by the original selector from `add_class`, so you can chain further modifications on to the end of it. Finally, we reset `response.body` to our modified document, and we're off to the races. Simple, huh?

10.3.2 Reading In Broken HTML

As anyone who has developed any sort of application that deals with parsing HTML, most people who write Your application reads in data from a series of blogs and aggregates the data for your readers. In most cases, the blogs provide a nice RSS feed, and you happily use the feeds to get the data you need. Unfortunately, a few of them don't provide any kind of feed, and the HTML is extremely sloppy. On the bright side, they haven't updated their layout in at least a decade or two, and you can be pretty sure that the template they're using isn't going to change any time soon.

Listing 10.6 A Sample Blog Entry

```
<h3>Open Thread </h3>

<p class="byline"><i>by</i> <a href="http://foo.com/">Mr. Foo</a>, Fri Aug
17, 2007 at 11:15:45 PM EST</p>

<div class="story_summary">
<p>Some Text Here</p>

<p>Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod
tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam,
quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo
consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse
cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non
proident, sunt in culpa qui officia deserunt mollit anim id est laborum.</p>

<p>
</div>

<p class="byline story_trail">
<br><a href="/story/2007/8/17/55555/5555">Permalink</a> :: 5 <a
href="/story/2007/8/17/55555/5555#commenttop">Comments</a>
<br>Tags: <a href="http://www.myblog.com/tag/foo%20tag">foo tag</a> (<a
href="http://www.myblog.com/tag">all tags</a>)
<br>
</p>
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

The post above uses a typical blog engine, and includes information about the person who wrote the article, when it was created, a permalink to it, and some tags, among other things. We want to get the information from the blog entry into an object we can use elsewhere in the application.

Solution

Although it would be very common to store the information from the post in an ActiveRecord object, which is exposed by the Ruby on Rails framework, and used standalone by other applications, we will use a run-of-the-mill class for this example. The code included with the book includes an example of using ActiveRecord to scrape and store the information that is very similar to the solution presented below.

First, you'll notice that there's a little bit of broken HTML in the example above. While it's mostly acceptable, it's a far cry from perfectly valid XHTML (there's an open `<p>` tag on its own before the close of the `<div>` tag, and the `<br>` tags aren't self-closing). Hpricot will attempt to fix these problems when it parses the HTML, so the output you'll eventually get will be, at the least, parsable.

Listing 10.7 Getting the HTML into a Post object

```
require 'rubygems'
require 'hpricot'
require 'time'

module RubyInPractice
  class Post

    attr_accessor :title, :body, :by, :posted_on, :permalink, :tags
    def initialize(title, body, by, posted_on, permalink, tags)
      @title, @body, @by, @posted_on, @permalink, @tags = title, body, [CA]
by, posted_on, permalink, tags
    end

    def self.from_blog_html(html, base_url)
      doc = Hpricot(html)
      title = (doc/"h3").inner_text
      body = (doc/"div.story_summary").inner_html.gsub(/\n/, "").
        gsub(/<\p><p>/, "<br/>").gsub(/<\/?p>/, "")
      by = (doc/"p.byline:first a").inner_text
      posted_on = Time.parse(doc/"p.byline:first [CA]
:last").inner_text.gsub(/^\s*/, "")
      permalink = base_url + [CA]
(doc/search("a:contains(Permalink)").attr("href")
      tags = (doc/"p.story_trail a:gt(1)")[0..-2].map {|x| [CA]
x.inner_html}.join(", ")
      new(title, body, by, posted_on, permalink, tags)
    end

  end
end
```

Discussion

The screen-scraping example is fairly rudimentary, but it allows us to take a look at a number of the techniques you can use with Hpricot to get very specific pieces of information quickly and easily. Before we take a look at

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

that, let's peruse through the structure we're using here. As in the previous example involving **Configuration**, we have separated out the initialization of the class from the function that will pull the information from an external source.

This is an important technique, because it will allow you to extend this class later with additional parsers that scan in different types of blog entries. You might have a **Post.from_wordpress_entry**, **Post.from_typo_entry**, and **Post.from_mephisto_entry** (while all of those have RSS feeds, let's pretend we needed to do screen-scraping for the sake of argument). Because we've left the initialization method so generic, we can easily plug into it after doing whatever complex parsing we need to do. That said, let's take a look at the actual code we used to get the various pieces of information out of the HTML fragment we pulled from the web page.

The first thing we did, as in the previous example, is create an **Hpricot** object from the HTML. To get the title, we simply grabbed the **inner_text** from the document's **<h3>** element. Take a moment to marvel at the simplicity of the API before we move on. The body of the article was in a **<div>** with the class of **story_summary**. All you need to do to get the contents of that element is to use elementary CSS knowledge: **div.story_summary** represents a **<div>** with the class **story_summary**, so that's all you need here. We do some fancy **gsubbing** to normalize line-spacing, and we're done.

Getting the byline is a bit trickier. Examining the HTML, we see that it's inside of a link that's inside a **<p>** element with the class **byline**. Unfortunately, there are **two** **<p>** elements with that class on the page. But that poses no difficulty for the dexterity of Hpricot. Using the custom **:first** extension to the typical CSS3 syntax, you can grab just the **first** matched element. Putting it together **p.byline:first** matches the link we need, and as before, **inner_text** will grab the text inside it.

Moving on to finding the date and time the post was created, we encounter another difficulty. The time is formatted well enough for our needs, but it's not surrounded by any tags. That, too, poses no problem for Hpricot. Examining the HTML, we see that the time is also in the first **<p>** element with the **byline** class, and that it's the very last thing in the **<p>**. Because all of the text is grouped together, Hpricot considers it a single **text node**, so the selector **p.byline:first :last** matches it. The **:last** selector is another one of Hpricot's custom extensions, and the space in between the two parts of the selector indicate that Hpricot should find the last child of the first **p.byline**. Using **Time.parse**, which is included from the standard library when we did **require 'time'** wraps it all up.

Getting the permalink proves easy after wrapping ourselves in knots to solve the last two problems. Hpricot supports the CSS3 selector **:contains**, which returns all elements whose text contents match the text passed in to **:contains**. In this case, the link we want has the text "Permalink" helpfully embedded inside, so we get the element and use the handy **attr** method to get its **href** attribute. We then append the **href** to the **base_url** passed in, as we can see that the permalink provided in the HTML is a relative URL.

Finally, we get the tags attached to the article. Looking at the HTML, we can see that the tags are all enclosed inside links in a **<p>** element with the class **story_trail**. Looking closer, we see that we don't want to include the first, second, or last link. Using Hpricot's **:gt** selector, we can filter out the first two elements. Once we have those, we can use Ruby's **Array#[]** operator (since an Hpricot element collection is simply an extension of the base **Array** class) to remove the last link. We then create an array containing the **inner_text** of the links, and **join** them with a comma, producing a nice, comma-separated list of tags.

All in all, using Hpricot to scrape information from typical HTML sites is a piece of cake once you learn the supported selectors. But it's key that you do. Most of the power of Hpricot comes from being able to drill down into the dark corners of the document you're parsing without resorting to manually iterating through them. Since Hpricot supports all of the same selectors as jQuery, you can get documentation on supported selectors by visiting <http://docs.jquery.com/Selectors>.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

10.4 Writing Configuration Revisited

Problem

Now that we've learned a bit about Hpricot, let's take a brief detour to look at how you might be able to rewrite our earlier Configuration example using Hpricot. Because we have tests for the class we wrote (which you can find on the downloads available with this book), let's make sure our new version still passes all of those tests as a sanity check.

Solution

I'm just going to show the modified `#to_xml` methods from `Config` and `Keyboard`, which reflect the usage of Hpricot instead of REXML. All tests written for the Section 10.2.2 still pass.

Listing 10.8 Redone Config#to_xml

```
def to_xml
  doc = Hpricot.XML("<?xml version='1.0' encoding='ISO-8859- [CA]
15'?><calculator/>")
  (doc/"calculator").append("<startup/>").[CA]
  search("startup").attr(@startup)
  (doc/"calculator").append("<memory>#{@memory.to_s}</memory>").
    search("memory").attr("type" => @memory.class.to_s[0].chr.downcase)
  (doc/"calculator").append(keyboard.to_xml)
  doc.to_s
end
```

Listing 10.9: Redone Keyboard#to_xml

```
def to_xml
  el = Hpricot.XML("<keyboard/>")
  (el/"keyboard").attr("type" => @type).
    append("<numeric-keypad/>").
    append("<max-fkey>#{@max_fkey}</max-fkey>").
    search("numeric-keypad").attr(@numeric_keypad)
  el.to_s
end
```

Discussion

Using Hpricot for XML creation involves a lot of literal node creation. You create fragments of XML, and stick them where they belong, and Hpricot takes care of creating XML nodes out of them. In the above examples, we also used a fair amount of chaining, taking advantage of the fact that Hpricot tends to return the original `Elements` array from operations performed on it.

Quickly looking through `Config#to_xml`, you can see that we replaced our individual calls to create the XML document, its root node, and the XML declaration with a single call to `Hpricot.XML`. We use `Hpricot.XML` rather than the typical `Hpricot` call when we want the parser to use strict XML semantics instead of HTML semantics. We then find the `<calculator>` node and append the empty `<startup>` node. Because the append method returns the original `Elements` array, we can then search for the new `<startup>` node and set its attributes in the same way that we did in the REXML solution. We use the same techniques until the end of the method, when we append the XML returned by `Keyboard#to_xml`.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

Because Hpricot uses string-based manipulation, we put together the XML for the Keyboard in much the same way that we created the `Config`'s XML, but return a `String`, so that it can be appended to the `<calculator>` node in the `Config` object. Rerunning the tests gives us solid green, so we're good to go.

The main thing to remember about using Hpricot to produce XML is that you'll be mostly producing XML fragments and using methods like `append`, `inner_html=`, and `attr` to set up your document. Once you get comfortable with it, though, it's extremely powerful.

10.5 Reading RSS Feeds

Over the last few years, RSS has become a common way for content authors to keep readers updated about the material they are posting. Blog authors use RSS to keep their readers updated with their freshest content, booksellers use RSS to let people know about new books they are selling, and podcasters use a specialized form of RSS to point listeners to their latest episodes.

Because RSS content is so consistent, it's very easy to write web applications that take advantage of its aggregation capabilities. Feed readers galore have popped up, allowing users to enter their favorite feeds and giving them back different ways to view the contents. We're going to take a look at how Ruby allows us to read through an RSS document, and build a rudimentary feed reader that we can use to get information about our favorite feeds.

Problem

We want to build a simple feed reader that will allow us to get the titles, URLs, brief descriptions, and publication dates for the articles in the feed. We want to be able to display the data on the command line by accepting command-line arguments, or on the web, as formatted HTML. While we will not provide details on how to integrate this solution with an web application framework such as Ruby on Rails in this book, code samples along those lines are available in the downloadable materials you have access to with this book.

Solution

Listing 10.10 A simple RSS Parser

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

module RubyInPractice
  class RssParser

    attr_accessor :rss
    def initialize(file, options = {})
      f = open(file)
      s = f.read
      @rss = RSS::Parser.parse(s)
      @options = {:truncate => 500}.merge(options)
    end

    def titles
      @rss.items.map {|x| x.title}
    end
  end
end
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```

    def short_info
      @rss.items.map {|x| [x.title, Hpricot(x.description).[CA]
inner_text[0..100] + "..."]}
    end

    def details
      @rss.items.map do |item|
        [ item.title,
          item.pubDate,
          item.link,
          Hpricot(item.description).[CA]
inner_text[0..@options[:truncate]] + "..."]
      end
    end

    def to_html
      @rss.items.map do |item|
        %{
          <h3><a href="#{item.link}">#{item.title}</a></h3>
          <h4>#{item.pubDate}</h4>
          <div class="body">#{item.description}</div>
        }
      end
    end

  end
end

```

Listing 10.11 A small command-line script for running the RSS parser

```

#!/usr/bin/env ruby
require 'optparse'
require 'rss_parser.rb'

module RubyInPractice
  class RssRunner

    def self.parse_options
      @config = {}
      OptionParser.new do |opts|
        opts.banner = "Usage: rss_parser [options]"
        opts.separator ""
        opts.on "-f", "--file [FILE]", [CA]
"the file or URL you wish to load in" do |file|
          @config[:file] = file
        end
        opts.on "-t", "--titles", "specify titles only" do |titles|
          @config[:titles] = true
        end
        opts.on "-s", "--short", [CA]
"show a short version of the feed" do |short|
          @config[:short] = true
        end
        opts.on "-r", "--truncate [NUMBER]", [CA]

```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```

"the number of characters to truncate the long [CA]
version to (defaults to 500)" do |trunc|
  @config[:truncate] = trunc.to_i
end
end.parse!
end

def self.run
  parse_options
  file = @config.delete(:file)
  r = RubyInPractice::RssParser.new(file, @config)
  if !@config[:titles] && !@config[:short]
    r.details.each do |item|
      puts "\"#{item[0]}\" published at #{item[1]}"
      puts "Available at #{item[2]}"
      puts
      puts item[3]
      puts
      puts
    end
  elsif !@config[:titles]
    r.short_info.each do |item|
      puts "\"#{item[0]}\""
      puts item[1]
      puts
    end
  else
    puts r.titles.join("\n")
  end
end

end
end

RubyInPractice::RssRunner.run

```

Discussion

There are two parts to the solution provided here. First up, we created a small class to read in an RSS feed and extract the useful information into some useful configurations. Second, we created a command-line script written in Ruby that allows you to provide a file-name and have it spit out a human-readable version of the feed, using the class we created first. But before we take a look at the command-line script, let's take a look at the class that does the hard work.

At the very top, we initialize our class with a file name. Because we have required `open-uri`, we can provide either a local filename or an Internet URL, which makes the code more versatile than it appears at first glance. We use `RSS::Parser.parse` to read the file into an RSS object, which will give us access to its contents. We also provide some default options, which we will allow to be set using command-line switches later on.

We'll now create the methods that will aggregate the information from the feeds that we'll need later. Pretty much all you need to know here is that the RSS object exposes an `items` array, and that each Item has a series of information associated with it, including its title, link to URL, publication date, and description. The

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

description will frequently have HTML data in it, so we'll strip that out using Hpricot for data being dumped to the terminal.

We'll also create a `to_html` method, which can be used by web frameworks to take the RSS feed and produce simple web-ready content. To see how it might be used, take a look at the `rss_controller.rb` file that is included with the downloadable files for this chapter. Note that we do not strip out the HTML content in the `to_html` method, because we are assuming it provides useful formatting instructions. As appropriate, you might choose to use the techniques you learned in the previous sections to convert the description HTML into more usable format.

Next, we use some simple command-line techniques to make the class we just created produce some useful content. It's not important that you fully understand how it works, but the basic principles should be obvious. We're using Ruby's built-in option parser to accept options to pass into our RSS parsing class. We accept a file parameter, which will be passed directly into our class (and as a result, will support either a local file or a URL). We also support a choice of long, short, or titles-only display. Finally, we allow the user to specify how many characters to truncate the long version's descriptions.

Because we are using `optparse`, we automatically support `-h` (or `--help`), which will provide usage instructions based upon the information we specified. You can see for yourself by running `parse_it.rb`, which is provided with the downloadable materials for this chapter. Try entering a URL to the feed for your favorite blog. If you don't have one, check out Manning's feed at <http://www.manning.com/links/index.xml>.

Returning briefly to the command-line script, you can see two main sections: `#parse_options` and `#run`. These names are not required, but they're conventional. On the bottom, you can see that we call `RubyInPractice::RssRunner.run`, which itself calls `parse_options`. The option parser will take apart the arguments you passed into the function and set up the `@config` instance variable, which we use in `#run` to determine the file name and to pass configuration options to the RSS parser.

10.6 Creating Your Own Feed

Now that you've learned how to read feeds from other sites, you might be thinking that you'd like to create your own feed. Unfortunately, there are a number of feed formats, and creating a program that exports only to RSS1, for instance, will leave some of your potential readers in the dark. Thankfully, there exists a library called `feedtools` that will allow you to create a generic feed, and then export it into multiple formats.

Problem

You want to create a feed for your website, but you only want to have to create a single feed, and have it export feeds in the formats you want. Ideally, you would be able to decide at a later point exactly which feed formats you wanted to support, so the Ruby class should be flexible enough to support various formats.

Solution

We'll create a wrapper around `feedtools` that will allow us to easily create a new feed and export it to whichever format we want. We'll only wrap a few commonly used functions for demonstration purposes, but it's trivial to extend this wrapper to provide additional functionality. Note that before you use this code, you will need to install the `feedtools` gem.

Listing 10.12 Wrapping Feed Tools

```
require 'rubygems'
require 'feed_tools'
```

```
module RubyInPractice
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

module Feeds

```
class Feed
  attr_accessor :feed
  def initialize(options)
    @feed = FeedTools::Feed.new
    options.each do |option, value|
      @feed.send("#{option}=", value)
    end
  end

  def to_atom
    feed.build_xml("atom", 1.0)
  end

  def to_rss(version=2.0)
    feed.build_xml("rss", version)
  end

  def add_entry(options)
    feed << Item.new(options).item
  end

  def add_entries(*args)
    args.each do |entry|
      add_entry(entry)
    end
  end

  def author=(name, email, href=nil)
    a = FeedTools::Author.new
    a.name = name
    a.email = email
    a.href = href if href
    feed.author = a
  end

  def add_links(hash)
    hash.each do |href, title|
      l = FeedTools::Link.new
      l.href = href
      l.title = title
      feed.links << l
    end
  end

  def method_missing(meth, *args)
    if meth.to_s =~ /=$/
      feed.send(meth, *args)
    end
  end
end
```

class Item

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```

attr_accessor :item
def initialize(options)
  @item = FeedTools::FeedItem.new
  options.each do |option, value|
    @item.send("#{option}=", value)
  end
end

def method_missing(meth, *args)
  if meth.to_s =~ /=$/
    item.send(meth, *args)
  end
end
end
end

```

Discussion

While Feed Tools is quite cool, in that it provides a single API for multiple feeds, it suffers from extreme API complexity. Adding an author to a feed, for instance, requires creating a new `FeedTools::Author` object (trying to call `author=` on a feed with a `String` will populate only the `name` field of the object). In order to simplify things a bit, we have created a new class called `RubyInPractice::Feeds::Feed`, which accepts a series of fields and instantiates the feed quickly and easily.

Because you set most of the fields on the `FeedTools::Feed` object by calling methods like `link=`, `id=`, and `copyright=`, we've allowed you to instantiate our wrapper object with a Hash with corresponding values. For instance, to create a new feed object with the feed's ID set to `http://www.manning.com`, you would do `RubyInPractice::Feeds::Feed.new(:id => "http://www.manning.com")`. Adding entries is a similarly complex process with `FeedTools`, so we have created a similar API for adding new entries. Once you have your wrapper object (let's call it `feed`), you would call:

```

feed.add_entry(:id      => "http://www.manning.com/foo",
               :title    => "NEW!",
               :abstract  => "Brand new feed entry",
               :description => "This is the first feed entry for our feed",
               :content   => "Some longer content would go here")

```

If you were creating an iTunes feed, there are additional properties, like `enclosure`, `itunes_summary`, `itunes_image_link`, and more than can be set using this method. Keep in mind that atom feeds require unique IDs, and those IDs must be URLs, or you will not be able to generate an Atom feed.

Because setting up an author correctly using `FeedTools` can also be complex, we have added a `set_author` method that takes a `name`, `email`, and optional `href`, creates the required `FeedTools::Author` object, and associates the object with the wrapped feed. We do a very similar thing with the `add_links` method, which allows the creation of `FeedTools::Link` object by simply passing in a hash of `href => title` pairs. You would do something like `feed.add_links("http://www.manning.com" => "Manning Publications")`.

To facilitate the creation of RSS and Atom feeds, we have also created two generation methods: `to_rss` and `to_atom`. Both methods simply delegate to the `build_xml` method of the underlying `FeedTools::Feed` object. Before we move on, let's take a look at the feeds that will be generated using our wrapper script.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

Listing 10.13 Creating a new feed

```
f = RubyInPractice::Feeds::Feed.new(:id => [CA]
"http://rubyinpractice.manning.com", :title => "Ruby in Practice Feed", [CA]
:description => "A sample feed for the Ruby in Practice chapter on Data [CA]
parsing")
f.set_author "Yehuda Katz", "wycats@gmail.com"
f.add_entry(:title => "NEW!", :abstract => "This is a new feed", [CA]
:summary => "This is a new feed for testing", :id => [CA]
"http://rubyinpractice.manning.com/new", :content => "There might [CA]
normally be some long content here")
```

The API using our wrapped set is fairly simple, as you can see. Now, let's see what happens when we go to generate the feeds using our `to_atom` and `to_rss` methods.

Listing 10.14 Generating RSS 2.0

```
<?xml version="1.0" encoding="utf-8"?>
<rss xmlns:taxo="http://purl.org/rss/1.0/modules/taxonomy/"
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:itunes="http://www.itunes.com/dtds/podcast-1.0.dtd" version="2.0"
xmlns:media="http://search.yahoo.com/mrss"
xmlns:dc="http://purl.org/dc/elements/1.1/"
xmlns:content="http://purl.org/rss/1.0/modules/content/"
xmlns:trackback="http://madskills.com/public/xml/rss/module/trackback/">
  <channel>
    <title>Ruby in Practice Feed</title>
    <link>http://rubyinpractice.manning.com/</link>
    <description>A sample feed for the Ruby in Practice chapter on [CA]
Data parsing</description>
    <managingEditor>wycats@gmail.com</managingEditor>
    <ttl>60</ttl>
    <generator>http://www.sporkmonger.com/projects/feedtools/</generator>
    <item>
      <title>NEW!</title>
      <link>http://rubyinpractice.manning.com/new</link>
      <description>This is a new feed for testing</description>
      <content:encoded>
        <![CDATA[There might normally be some long content here]]>
      </content:encoded>
      <pubDate>Tue, 28 Aug 2007 13:37:55 -0000</pubDate>
      <guid isPermaLink="true">http://rubyinpractice.manning.com/new</guid>
    </item>
  </channel>
</rss>
```

Listing 10.15 Generating ATOM 1.0

```
<?xml version="1.0" encoding="utf-8"?>
<feed xml:lang="en-US" xmlns="http://www.w3.org/2005/Atom">
  <title type="html">Ruby in Practice Feed</title>
  <author>
    <name>Yehuda Katz</name>
    <email>wycats@gmail.com</email>
  </author>
  <link href="http://rubyinpractice.manning.com/" rel="alternate"/>
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```

    <subtitle type="html">A sample feed for the Ruby in Practice chapter [CA]
on Data parsing</subtitle>
    <updated>2007-08-28T13:38:09Z</updated>
    <generator>FeedTools/0.2.26 - [CA]
http://www.sporkmonger.com/projects/feedtools/</generator>
    <id>http://rubyinpractice.manning.com</id>
    <entry xmlns="http://www.w3.org/2005/Atom">
        <title type="html">NEW!</title>
        <author>
            <name>n/a</name>
        </author>
        <link href="http://rubyinpractice.manning.com/new" rel="alternate"/>
        <content type="html">There might normally be some long content[CA]
here</content>
        <summary type="html">This is a new feed for testing</summary>
        <updated>2007-08-28T13:37:55Z</updated>
        <id>http://rubyinpractice.manning.com/new</id>
    </entry>
</feed>

```

As you can see, we get back two very different feeds, with all of the required feed-specific data, even though we were able to create them very simply. As I said above, the wrapper API is far from complete, but the code should provide you with enough to get started and adding more functionality. Unfortunately, the rdoc for FeedTools is not easily available online; however, it is installed with the gem. The easiest way to take a look at it would be to run `gem_server` and navigate to http://localhost:8808/doc_root/feedtools-0.2.26/rdoc/index.html.

10.7 Using YAML

YAML, which stands for YAML Ain't Markup Language, is a lightweight alternative to XML, which, even in its more structured and parsable forms, requires quite a bit of code to get simple things done. At first glance, YAML can store and reload simple objects like Hashes, Arrays, Integers, and Strings, which will allow you to develop a serialization and load strategy for your configuration objects. For instance, Ruby on Rails uses YAML to store information about the database it uses in different operating modes.

With that said, YAML is even more powerful, in that it is able to do quick-and-dirty serialization of your custom objects, as long as they're relatively simple. In this section, we'll take on final foray into the land of calculator configuration to see how we can use YAML to make the entire process of serializing the configuration object and reloading it later absolutely trivial.

Problem

As in the above examples, we would like to be able to take an existing `Calculator::Config` object and store it to disk. When our application starts, we would like to be able to load in the file from disk and create a new `Calculator::Config` object for us to use.

Solution

Listing 10.16 Using YAML to solve the calculator config problem
require 'yaml'

```

module Calculator
  class Config

```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```

attr_accessor :memory, :startup, :keyboard
def initialize(memory, startup, keyboard)
  @memory, @startup, @keyboard = memory, startup, keyboard
end

def save_to(file)
  f = File.open(file, "w")
  f.puts(self.to_yaml)
end

def self.get_from(file)
  YAML.load(File.read(file))
end
end

class Keyboard
  attr_accessor :type, :numeric_keypad, :max_fkey
  def initialize(type, numeric_keypad, max_fkey)
    @type, @numeric_keypad, @max_fkey = type, numeric_keypad, max_fkey
  end
end
end

```

Listing 10.17 Using the YAML solution in an equivalent manner to our XML solution

```

k = Calculator::Keyboard.new("Macintosh", {"enter" => "=", [CA]
  "clear" => "C"}, "F16")
c = Calculator::Config.new(16.24, {"mode" => "standard", [CA]
  "precision" => 2}, k)

c.save_to("config.yml")
Calculator::Config.get_from("config.yml")

```

Discussion

Using Ruby's built-in YAML serialization, we were able to take a complex problem that we previously solved using XML serialization, and convert it into a one-liner. It's important to realize that YAML serialization isn't the solution for every problem, because it can only serialize data structures, not objects that contain baked-in code (like procs and metaclass methods). However, for most configuration requirements, simple data structures are more than adequate. In fact, if you're getting much more clever, you're probably doing something wrong. Creating a separate **Keyboard** class to hold keyboard information was pushing it for us, but it was useful for the examples at hand.

For more complex examples, you can configure the YAML serialization. By defining the **to_yaml_properties** method to return an array of the properties you wish to serialize, you can cause **YAML#dump** to ignore properties that are not appropriate for serialization. You can also define **yaml_dump** and **yaml_load** methods on the object you are serializing to use a custom serialization strategy. You can get more information of customizing the default load and dump in the Pickaxe book.

11 Identity and Authentication

Trust is the foundation of successful systems. You need to be able to trust that your users are who they say they are, and your users must trust you with their identity and their personal data. This chapter looks primarily at how you can trust your users – by providing a robust authentication mechanism. How well you execute that authentication goes a long way toward how much trust your users will place in your application. Whether it is ensuring users that their passwords are safe, or offering an authentication option that integrates a trusted third party, your approach to security has a big impact.

In this chapter, we take a look at how to use Ruby and Rails to implement your own secure authentication schemes as well as integrate with established authentication mechanisms like Lightweight Directory Access Protocol (LDAP), Active Directory, and OpenID.

11.1 Securely storing a password

With few exceptions, your users' passwords should never be persisted in a way that would allow anyone access to them. For example, if a hacker were to get a hold of your site's database, there should not be a table with a "password" column filled with passwords as plain text. Hopefully, this is obvious, but there are enough sites on the web violating this rule that it is worth making sure that your applications don't make this mistake.

Problem

You need to store a password for later authentication, and do it securely.

Solution

Securely hash the password using a salt and store the resulting string. When it is time to authenticate a user, run the submitted password through the same one-way process and compare the result to what you stored. Here is an example of an ActiveRecord class implementing this approach:

```
require 'digest/sha2'

# table "users"
#   login, string
#   secured_password, string(40)

SECRET_SALT = "abracadabra"

class User < ActiveRecord::Base

  attr_accessor :password
  before_save :secure_password

  def self.authenticate?(login, pass)
    user = find_by_login_and_secured_password(login, encrypt(pass))
    return true unless user.nil?
    return false
  end
end
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```

protected

def secure_password
  self.secured_password = self.class.encrypt(password)
end

def self.encrypt(password)
  Digest::SHA1.hexdigest("--#{SECRET_SALT}--#{password}--")
end

end

```

Discussion

An implementation of the SHA1 algorithm is included in the standard Ruby library, where it was written in C to make it perform as quickly as possible. The properties of this government standard algorithm make it a great solution to the problem of storing passwords. Specifically, it is repeatable and non-reversible. It is important that it be *repeatable* so that the same password will produce the same result, which can be compared. It is important that it be *non-reversible* so that given access to the encrypted string, it is computationally infeasible to discover the actual password.

The salt makes it more difficult to do a dictionary attack against your encrypted passwords, by precomputing a list of the encrypted versions of common passwords and checking them against yours.

Although the SHA1 algorithm is in wide use, vulnerabilities have been discovered. The vulnerability is not believed to be serious as an attacker would need much greater computational power than is typically available. Still, developers with healthy paranoia would do well to consider the more secure cousins of SHA1: known collectively as SHA2. The members of SHA2 that have implementations included in Ruby's standard library are SHA256, SHA384 & SHA512. Here is some sample usage:

```

require 'digest/sha1'

puts Digest::SHA256.hexdigest("Please hash me.")
puts Digest::SHA384.hexdigest("Please hash me.")
puts Digest::SHA512.hexdigest("Please hash me.")

```

Here is an alternate approach to the solution that adds some complexity but offers a more secure overall system. Instead of a single salt, we will create a salt for each user and store it as an additional field. This dramatically increases the resources necessary for a dictionary attack like the one described above. It also means that if two users choose the same password, the encrypted version will still be different. If one of them had access to the encrypted passwords, they could not discover they were using the same password as the other user.

```

require 'digest/sha1'

# table "users"
#   login, string
#   secured_password, string(40)
#   salt, string(40)

class User < ActiveRecord::Base

  attr_accessor :password
  before_save :secure_password

```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```

def self.authenticate?(login, pass)
  user = find_by_login(login)
  return false if user.nil?
  return user.authenticate?(pass)
end

def authenticate?(pass)
  secured_password == encrypt(pass, salt)
end

protected

def secure_password
  self.salt = generate_salt if new_record?
  self.secured_password = encrypt(password)
end

def generate_salt
  Digest::SHA1.hexdigest("--#{Time.now.to_s}--#{login}--")
end

def encrypt(password)
  Digest::SHA1.hexdigest("--#{salt}--#{password}--")
end

end

```

Note that in this version, a row must be retrieved from the database in order to validate a password. This is because the salt is stored alongside the login and secured password. At first glance, this seems like it might be insecure, and indeed, the separate location of the salt and secured password is an advantage of the original message. However, that advantage is outweighed by how much more difficult we make life for an attacker by having a unique salt for each user.

Note: this approach is used by the `acts_as_authenticated` Rails plugin, discussed later in this chapter.

Bonus: Avoid writing passwords to your Rails log files

If we go through great efforts to avoid persisting our users' passwords in plain text, it would be careless to write passwords to our log files. To avoid this in your Rails application, add the following to your controller that is handling authentication:

```

class UsersController < ApplicationController
  filter_parameter_logging "password"
  ...
end

```

where "password" is the name of your password parameter. This will replace the value of any parameter whose name contains "password" with the string "[FILTERED]".

11.2 Authenticating against LDAP

LDAP, or Lightweight Directory Access Protocol, is an open standard for interacting with hierarchical sets of people, groups, or resources. Popular open source LDAP servers include OpenLDAP and Fedora Directory Server (FDS). Commercial offerings include Sun's Java System Directory, Novell's eDirectory, and others.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

LDAP servers are well-suited to hold information about people and their security credentials. As a result, they are commonly used for authentication purposes. Let's take a look at how you can accomplish this with Ruby.

Problem

You need to authenticate users against an LDAP server.

Solution

We can use a Ruby LDAP library like the `ruby-net-ldap` gem to validate a username/password pair:

```
require 'net/ldap'

def valid_user?(username, password)
  ldap = initialize_ldap(username, password)
  ldap.bind
end

def initialize_ldap(username, password)
  Net::LDAP.new(:base => "dc=example,dc=com",
               :host => "your-ldap-server",
               :auth => {:username => "uid=#{username},cn=users",
                       :password => password,
                       :method => :simple})
end
```

The method `LDAP#bind` attempts to connect to the LDAP instance using the credentials you supply and returns true if successful, and false if not.

Discussion

The primary alternative to the `ruby-net-ldap`, used above, is `ruby-ldap`. While `ruby-net-ldap` is a pure Ruby LDAP client implementation, `ruby-ldap` is a wrapper around the OpenLDAP C library. As you might guess, this means that `ruby-net-ldap` is far more portable and easier to install, though `ruby-ldap` offers significantly better performance.

`ruby-ldap` has an interesting library built on top of it called `Ruby/ActiveLdap`. `ActiveLdap` maps LDAP entries to objects in much the same way that `ActiveRecord` maps database rows to objects. Here is an example of defining a `User` class using `ActiveLdap`:

```
class User < ActiveLdap::Base
  ldap_mapping :dn_attribute => 'uid', :prefix => 'cn=users',
               :classes => ['top','account']
  belongs_to :groups, :class => 'Group', :many => 'memberUid',
             :foreign_key => 'uid'
end
```

`ActiveLdap` is likely overkill if your only use for LDAP is authentication. In that case, `ruby-net-ldap` is probably fast enough and your best bet.

On the other hand, if your application is doing a lot with LDAP entries, like queries and read/write operations, `ActiveLdap` is definitely worth a look.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

11.3 Authenticating against ActiveDirectory

ActiveDirectory, from Microsoft, is used by many businesses for identity management. Integrating your authentication system with ActiveDirectory is a great way to provide a good sign-on experience for internal applications and help the business keep identity and password management centralized.

Problem

You need to authenticate users against ActiveDirectory.

Solution

Fortunately, ActiveDirectory is compatible with the LDAP spec, so we can use a Ruby LDAP library like the `ruby-net-ldap` gem to validate a username/password pair:

```
require 'net/ldap'

def valid_user?(username, password)
  ldap = initialize_ldap(username, password)
  ldap.bind
end

def initialize_ldap(username, password)
  Net::LDAP.new(:base => "dc=example,dc=corp",
               :host => "exampledomaincontroller",
               :auth => {:username => "#{username}@example.corp",
                       :password => password,
                       :method => :simple})
end
```

Ask your ActiveDirectory administrator for the name of the “domain controller” – that’s the LDAP “host” in the case of an ActiveDirectory installation.

The example above takes advantage of a convenience offered by ActiveDirectory. You can send `user@domain` as the username for the purposes of binding and ActiveDirectory will handle it properly. Normally, LDAP instances expect a distinguished name (“DN”) in this spot.

Discussion

The approach above is suitable for an application that should be available to any member of the ActiveDirectory instance, but to get more information about a user, you need to use `LDAP#search`. This would be useful, for example, to validate that a user is not only who she says she is (via password), but also that she is a member of a particular group:

```
def valid_app_user?(username, password)
  ldap = initialize_ldap(username, password)
  entries = ldap.search :filter => "sAMAccountName=#{username}"
  return false unless entries && !entries.empty?
  return member_of_ad_group?("G-MyAppUsers", entries.first)
end

def member_of_ad_group?(group, ldap_entry)
  group_names = ldap_entry[:memberof] # returns a collection of fully-
                                     # qualified ActiveDirectory groups
  group_names.any?{|name| name.include? "CN=#{group}," }
end
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

The search method uses a filter based on the “sAMAccountName” attribute, which is the closest thing to a simple username attribute in ActiveDirectory’s schema.

The array of groups returned by the :memberof attribute of an ActiveDirectory LDAP entry contains the “distinguished name” of the groups. Something like "CN=G-MyAppUsers,OU=Groups Security,OU=Example,DC=example,DC=corp". In the `member_of_ad_group?` method, we’re looking for a match in the first part, the Common Name.

A library built for ActiveDirectory

ruby-activedirectory takes all the features and idiosyncrasies of ActiveDirectory into account. It provides classes for Users, Groups and Computers. `ActiveDirectory::User#authenticate` will validate a particular user’s password. Using it is straightforward:

```
require 'activedirectory'

def valid_user?(username, password)
  user = ActiveDirectory::User.find("jdoe")
  user.authenticate(password)
end
```

This library depends on ruby-ldap being installed, which can be a little tricky as it wraps the OpenLDAP C library. If you can get over that hurdle, `ruby-activedirectory` is quite useful in working with an ActiveDirectory installation.

11.4 Adding authentication to your Rails application... fast

Most non-trivial Rails applications offer a personalized experience to each user of the site. The most common solution to personalization needs is letting users sign up for an account and authenticating them when they visit your site. While implementing a username / password authentication eventually leads to various decisions and custom code, adding a basic implementation shouldn’t get in your way as you dig into the most interesting parts of your Rails application.

Problem

You need to authentication to a Rails application... fast.

Solution

Use the `acts_as_authenticated` plugin by to generate the necessary code and database migrations and add them to your rails application. Step-by-step:

Install the plugin:

```
script/plugin install acts_as_authenticated
```

Generate a User model, a Users Controller, a migration, along with tests and a couple of useful library classes.

```
script/generate authenticated User
```

Mix the `AuthenticatedSystem` module into `ApplicationController` instead of `UsersController` so that it is available to all of your controllers. Remove the `include AuthenticationSystem` line from `users_controller.rb`:

```
class UsersController < ApplicationController
  # Be sure to include AuthenticationSystem in Application Controller instead
  include AuthenticatedSystem
  ...
end
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

end

Add it to application.rb:

```
class ApplicationController < ActionController::Base
  include AuthenticatedSystem
  ...
end
```

Add "before_filter :login_required" to each controller that you want to restrict to logged in users. For example:

```
class TodosController < ApplicationController
  before_filter :login_required
  ...
end
```

Run the generated database migration

```
rake db:migrate
```

Start your app.

```
script/server
```

Create the first account by pointing your web browser to <http://0.0.0.0:3000/users/signup>

If you're successful, after signing up, you should see a poem by Ellen P. Allerton -- a bit of whimsy from the plugin's author, Rick Olson.

Rick Olson is a member of the Ruby on Rails Core Team and is known online as "technoweenie." You can discover some incredibly useful plugins and learn a lot by browsing his repository at <http://svn.techno-weenie.net/projects/plugins/>

Discussion

It is easy to assume that authentication on the web is simple because thousands of websites have implemented it. The truth is the possible variations and nuances are manifold.

Can any visitor sign up and create an account? Or are accounts only created by users with administrative privileges? Do a user log in with her username? Or does she log in with an email address? Perhaps either is possible? What sort of confirmation or activation step is necessary?

The purpose of the `acts_as_authenticated` plugin is not to support all these scenarios, but rather to put into place the most common needs for username/password authentication and give you a solid base to build upon.

Helping you stay test-driven

Our favorite thing about the plugin is the quality of the tests it adds to your project. Both the generated model and controller are accompanied by tests that together sport 100% code coverage. When you're ready to start customizing your application's authentication logic, start with the tests, either adding tests to drive new features, or modifying tests to drive changes to the generated code.

You also get some very useful helper methods for you to use in your own tests. Inside of your `lib` directory, you'll notice that the generation process has added `authenticated_test_helper.rb`. It defines a module that is mixed into the two test classes generated (`test/unit/user_test.rb` and `test/functional/users_controller_test.rb` in the example above). Let's look at how to use a few of these helper methods.

login_as: interact with controllers as an authenticated user...

```
def test_index_ok_for_logged_in_user
  login_as :quentin
  get :index
  assert_response :ok
end
```

Please post comments or corrections to the Author Online forum at

<http://www.manning-sandbox.com/forum.jspa?forumID=356>

end

The `login_as` method takes a symbol (`:quentin` in the example above) that is used to lookup a user from the fixture YAML. It then sets that user as the logged in user by setting the `session[:user]` to the ID of the specified user.

...or as an anonymous user

```
def test_index_not_available_to_anonymous_user
  login_as nil
  get :index
  assert_response 401
end
```

When the `login_as` method is passed `nil`, it clears any existing value of `session[:user]`, and thereby lets you simulate an anonymous user.

assert_difference: make sure a new instance of a model was created

```
def test_create_increments_user_count
  assert_difference User, :count, 1 do
    post :create, :user => {:login => 'foo', :password => 'test'}
  end
end
```

The power of Ruby's block syntax is evident in the `assert_difference` method. In this example, the `count` method of `User` is called twice. In between, the content of the block is executed, a POST to the create action in this case. The return values of the two calls are compared to verify that the difference is equal to the value of the third argument to `assert_difference`, or 1 in our example.

assert_no_difference: make sure a new instance of a model wasn't created

```
def test_create_with_invalid_login_does_not_change_user_count
  assert_no_difference User, :count do
    post :create, :user => {:login => nil, :password => 'test'}
  end
end
```

`assert_no_difference` works, just like `assert_difference`, but instead ensures that the return value of the specified method is the same before and after the block executes.

assert_requires_login: verify an action causes a redirect to the login page

```
def test_new_redirects_to_login
  assert_requires_login { |c| c.get :new }
end
```

This test helper demonstrates another elegant use of block syntax. `assert_requires_login` yields a proxy class that wraps `@controller`. The proxy class passes method calls along to the actual controller, and then verifies that the result was a redirect to the login page.

If you're thinking, "wow, some of these test helpers are really handy," you would be right. It is a good idea to mix this module into the `Test::Unit::TestCase` class defined in `test/test_helper.rb`. You can then remove it from the generated tests and use it in any of your tests.

Add account activation with ActiveMailer

If you need to verify that your users have actual working email addresses, a second generator included in `acts_as_authenticated` is there to help you out. Use it like this:

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```
script/generate authenticated_mailer user
```

This piece of the plugin is less polished than the part we discussed already, and you'll need to do some additional work to get the generated code and tests to work properly. The details are beyond the scope of this discussion, but you can dig in on the wiki: <http://technoweenie.stikipad.com/plugins/show/Acts+as+Authenticated>

Use it, then lose it

The `acts_as_authenticated` plugin consists entirely of code generators, so there's really no need to keep it installed once you're happy with the code you've generated. You can remove it like this

```
script/plugin remove acts_as_authenticated
```

The code you generated with it will still be present and working but you won't be transferring an extra directory with every deployment for no reason.

Alternatives

`acts_as_authenticated` is the most commonly used plugin for this task, but there are a few others worth considering.

restful_authentication is by the same author, and seeks to solve the same problem, while following a RESTful paradigm. At the time of this writing, `restful_authentication` is gaining fast in popularity and may eclipse `acts_as_authenticated` as the de facto way to get started with authentication in a new Rails app.

For simple HTTP Basic or HTTP Digest authentication, try the **htpasswd** plugin (<http://wota.jp/svn/rails/plugins/branches/stable/htpasswd/>). It uses standard formats that you may be familiar with if you've worked with Apache.

Acts As Administered Authentication (<http://aaaa.devjavu.com/>) is an adaptation of `acts_as_authenticated` that eschews the "anyone can sign up" approach for a system where user accounts are created by admins.

11.5 Semi-private, personalized feeds

RSS or ATOM feeds are a great way for users to keep form a connection to your site. In the case of a data-driven site, the information in a feed is likely unique or at least personalized for each user. Unfortunately, many feed readers don't support username/password authentication, or don't do so securely.

Problem

You need to offer feeds that need to be personalized and semi-private but still work with typical feed readers.

Solution

Create a secret token associated with each user that can be used to authenticate for feeds only. Here is a sample ActiveRecord migration:

```
require 'digest/sha1'
```

```
class AddUserToken < ActiveRecord::Migration
  class User < ActiveRecord::Base; end
```

```
  def self.up
    add_column :users, :token, :string, :limit => 40
    User.find(:all).each do |user|
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```

        user.token = Digest::SHA1.hexdigest "#{Time.now.to_i}#{rand}"
        user.save
      end
    end

    def self.down
      remove_column :users, :token
    end
  end
end

```

Your User model should automatically create a token when a new user is added:
 require 'digest/sha1'

```

class User < ActiveRecord::Base
  before_create :generate_token
  ...
  def generate_token
    self.token = Digest::SHA1.hexdigest "#{Time.now.to_i}#{rand}"
  end
end

```

You are creating a random 40-character token, using SHA1 as convenient way to do that. You will then add this token to the personalized feed URLs your application exposes and use it to associate the incoming requests for feeds with the right user's data. If your application's actions require a login by default (via a filter in the Application Controller called `login_required`), a comments controller implementing this concept might look like this:

```

class CommentsController < ApplicationController

  skip_before_filter :login_required, :only => [:index]
  prepend_before_filter :login_or_feed_token_required, :only => [:index]

  def index
    @comments = current_user.comments
    respond_to do |format|
      format.html
      format.rss { render_rss_feed_for @comments }
      format.atom { render_atom_feed_for @comments }
    end
  end

  ...
end

```

Here, for the index action only, the controller will bypass the default `login_required` filter, but put add a filter that calls the method `login_or_feed_token_required`. Here is the implementation (this goes in your Application Controller):

```

def login_or_feed_token_required
  if ['rss', 'atom'].include?(params[:format]) && params.has_key?(:token)
    return true if @user = User.find_by_token(params[:token])
  end
  login_required
end

```

We attempt to authenticate using the token if appropriate, and otherwise fall back to the `login_required` method.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

Discussion

The solution relies on providing an alternate means of authentication (a secret token) to each user. This alternate authentication provides only limited rights: read-only access to XML feeds. In the event that the secret is leaked, this approach limits exposure. Contrast that with having a user's password compromised because your application forced him to embed it in a feed URL to use it from a web-based feed reader.

Allow users to regenerate their tokens

Since we want to support users in keeping their data secure, it is a good idea to provide a mechanism for a user to replace her secret token on demand. You can provide this with a controller action like this one:

```
class UsersController < ApplicationController
  ...
  def refresh_token
    @user.generate_token
    @user.save!
  end
  ...
end
```

It is important that the user realize that changing her token will break any externally stored URLs. If you were ambitious, you could devise a system of multiple, independently expireable tokens that can be generated on-demand.

Take the same approach for an API key

You might consider using the same token, or a separate one, as an authentication mechanism for your application's API. If your API provides both read and write operations, this obviously increases risk. Still, it is probably preferable to your user to embed a randomly-generated 40-character string in a script than his password.

Tip: disable the session for feed requests

When you detect that you are handling an RSS or ATOM request, you can be fairly confident that an RSS reader or a script is the requestor. In these situations, you can more efficiently handle the request by turning off the session for that request, like this:

```
class CommentsController < ApplicationController
  ...
  session :off, :only => :index,
             :if => Proc.new { |req| is_feed_request?(req) }
  ...
  def self.is_feed_request?(req)
    ['rss', 'atom'].include?(req.parameters[:format])
  end
end
```

Note that the `is_feed_request` is defined as a class method of the controller. For the method to resolve in the `session :off` statement, it must be a class method.

Alternative approaches

If the data in the feeds is not sensitive, keep it simple and skip authentication for feeds altogether. You also consider making the feed content a shallow, non-sensitive indicator of the content with links back into the site, where the user would have to authenticate normally.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

11.6 HTTP Basic authentication

Sometimes you need some very simple authentication capabilities. HTTP Basic Authentication is a well-known standard supported by practically every web browser and web automation library. This is a simple way to secure an API or application meant to be consumed by simple scripts.

Problem

You need to access the username and password sent via HTTP Basic Authentication in a simple Ruby CGI application.

Solution

Here's how you can extend CGI to allow access to HTTP Basic data:

```
class CGI
  def basic_auth_data
    user, pass = '', ''
    header_keys = ['HTTP_AUTHORIZATION', 'X-HTTP_AUTHORIZATION',
                  'X_HTTP_AUTHORIZATION',
                  'REDIRECT_X_HTTP_AUTHORIZATION']
    header_key = header_keys.find{ |key| env_table.has_key?(key) }
    authdata = env_table[header_key].to_s.split if header_key
    if authdata and authdata[0] == 'Basic'
      user, pass = Base64.decode64(authdata[1]).split(':')[0..1]
    end
    return user, pass
  end
end
#usage
cgi = CGI.new
username, password = cgi.basic_auth_data
```

In this example, we open up the CGI class and add `CGI#basic_auth_data`, a method that can extract the relevant HTTP headers according to the HTTP spec. Note how two of Ruby's language features (open classes and multiple return values) allow us to create nicer usage syntax than we could in less flexible languages.

Discussion

As you can tell from the code above, the username and password is Base64-encoded. This makes the user's credentials extremely easy to access by an attacker sniffing HTTP traffic. Ensuring that HTTP traffic occurs over HTTPS is one way to make this technique more secure.

While a programmatic web client can be written to always send authorization headers, most graphical web browsers prompt for basic authentication upon receiving a response with a 401 status code and WWW-Authenticate header. While testing using a browser, you may not see any authorization headers until you challenge the browser. For example:

```
cgi.out("status" => "AUTH_REQUIRED",
      "WWW-Authenticate" => 'Basic realm="My Application"') { "And you are...?" }
```

This approach can be adapted to integrate with pre 2.0 Rails apps using a similar approach or by leveraging Code Hale's `simple_http_auth` plugin. Rails 2.0 includes this functionality in `ActionController::Base#authenticate_or_request_with_http_basic`.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

11.7 Letting users sign into your application using OpenID

Open ID is an open standard for decentralized single sign-on. It seeks to address the problem that every internet user is familiar with – too many usernames and passwords.

If a site supports Open ID, a user can simply enter her Open ID (janedoe.myopenid.net, for example) and get redirected to her Open ID provider's website, where she agrees to allow the originating site to authenticate before being redirected back to the originating site with a token that proves that the user is the owner of the ID.

This is a simplification of the Open ID protocol. We recommend you familiarize yourself with the details of the interactions. A great resource is the documentation embedded in the ruby-openid gem source code. Check out lib/openid/consumer.rb in particular.

Problem

You need to support single sign-on via OpenID.

Solution

Use the ruby-openid gem to abstract the details of the inter-server communication and initiate and complete the process in controller actions.

```
require_gem 'ruby-openid'

class LoginController < ApplicationController
  def index
    #show the OpenID login form
  end

  def begin
    openid_url = params[:openid_url]
    response = openid_consumer.begin(openid_url)

    case response.status
    when OpenID::SUCCESS
      #send the user off to the OpenID server
      redirect_to response.redirect_url(url_for(:action => :index),
                                         url_for(:action => :complete))
    else
      flash[:error] = "No OpenID server for <q>#{openid_url}</q>"
      redirect_to :action => :index
    end
  end

  def complete
    open_id_response = open_id_consumer.complete(params)
    case open_id_response.status
    when OpenID::SUCCESS
      session[:openid] = response.identity_url #presence of this key
                                              #indicates a logged in user
      redirect_to url_for(:action => 'index')
    else
      flash[:error] = "OpenID auth failed: #{open_id_response.msg}"
      redirect_to url_for(:action => 'index')
    end
  end
end
protected
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```

def open_id_consumer
  @open_id_consumer ||= OpenID::Consumer.new(
    session[:openid_session] ||= {},
    create_open_id_store)

end

def create_open_id_store
  OpenID::FilesystemStore.new("#{RAILS_ROOT}/tmp/openid")
end

end

```

The view for the index action of the controller would look like this:

```

<% form_tag url_for(:action => 'begin') do %>
  <%= text_field_tag 'openid_url' %>
  <%= submit_tag 'Login' %>
<% end %>

```

Taken all together, this code will allow you to check the session for an :opened key and use that to determine whether the user is authenticated. You might define a helper method like so:

```

class ApplicationController < ActionController::Base
  ...
  def authenticated?
    session.has_key? :openid
  end
  ...
end

```

You could also have a User model backed by a database table if you need to store per-user preferences or attributes. You could access it based on the user's OpenID:

```

class ApplicationController < ActionController::Base
  ...
  def current_user
    User.find_by_open_id_url(session[:opened])
  end
  ...
end

```

A common scenario is to offer your visitors the option of logging in using a local username and password, or via OpenID. The code for this approach is beyond the scope of this section, but we encourage you to mix and match the sections of this chapter to create the right solution for your application.

Discussion

The Rails core team is working on a plugin that is compatible with Edge Rails (i.e. Rails 2.0+) written by DHH (`open_id_authentication`). EastMedia, which was integral to the introduction of OpenID to the Ruby world, released a plugin compatible with Rails 1.1+ (`openid_consumer`) and a newer plugin following the RESTful paradigm for Rails 1.2.1+ (`restful_open_id_authentication`).

There are two major differences between the plugins and the example code above. First, the plugins attempt to encapsulate most logic within modules in the plugin, naturally. Second, the plugins all default to using a database store instead of a filesystem store. For an application load-balanced across multiple servers, the filesystem store is inadequate.

OpenID is rapidly gaining support in the industry, with the likes of America Online and Microsoft backing it. Although it is primarily targeted for the public web, the applications for the enterprise are quite interesting.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

A company could run a public OpenID server for just its employees, thereby vouching for the identity of anyone with an OpenID provided by the company. Also, because OpenID libraries are available for practically every web development platform, it has become an option for single sign on in a heterogeneous environment. You might easily layer an OpenID Server atop your ActiveDirectory or proprietary authentication system. If you're interested in this, there is an open source OpenID server written in Ruby available as part of the Apache Heraldry project.

11.8 Summary

Teenagers today are fond of vintage t-shirts, or simply those that appear to be vintage. This style has been dubbed “authenticitude” by marketing wonks – the attitude of authenticity. For Rubyists, though, we think “authenticitude” should be the attitude of treating authentication seriously, and letting your users know that you do.

Whether your application lives in a corporate environment, or on the public Web, whether your users are open standards junkies, newbies, or mindless client applications, we hope this chapter has what you need to implement a great authentication system. Take it seriously. Know your users and let your users know that their data and identities are safe with you.

If you want to wear a faux-vintage t-shirt while you're coding it, that's your business...

12 Searching and Indexing

Throughout this book, and throughout your entire programming life, you’ve been used to dealing with data, as it forms the input and output for all computer programs. Most of the chapters in this book have looked at transforming data from one state to another, but in this chapter we’re going to investigate Ruby’s abilities to let you *search through* data.

Unfortunately, as a language that has only reached maturity in the last few years Ruby is not blessed with hundreds of libraries relating to search. This is not necessarily a bad thing, as search technologies progress quickly, and most of the available Ruby search solutions are up to date and ready to use in production immediately.

In this chapter we’ll look at Ruby-specific techniques to searching and indexing data and examine some solutions to common search-related problems, but first we’ll dive into defining what exactly searching and indexing are as well as define some of the other bits of terminology involved.

12.1 The Principles of Search

Starting from the most elementary point, “searching and indexing” refers to the process of taking a collection of data, processing it to be quick to scan through, and then enabling a program or a user to quickly find small elements of data within the larger set. Google provides the most ubiquitous form of search technology in our lives today. If you want to find a Web page that contains a certain word, or set of words, you go to Google.com, type in your query, and results are returned almost instantaneously.

If we look at how much data Google can search across, it’s somewhere in the range of billions of pages, which take up thousands of gigabytes of space. Google, therefore, certainly can’t scan through every page, byte by byte, looking for results to your query. If it did, it could take days (or longer!) to get your search results, although it would still, technically, be a “search engine.”

The process that takes Google from taking many days to produce search results through to being able to provide results in under a second is “indexing,” and the performance benefits of indexing are so significant that every search library or tool must provide indexing services of one form or another.

Indexing technologies can be extremely advanced, but at their most basic work in the same way as an index in a book. For example, if you turn to the index of this book and search for a particular term, you know roughly where in the index to look for your term since the index is in alphabetical order. Furthermore, once you find your desired term, you’re provided with a set of page numbers to refer to. This contrasts significantly with having to read through every page to find something, and so computers use similar techniques. A search engine’s indexer will make a distinct note of words and phrases on a page, and then link those terms along with a unique ID that references those terms to the page it is indexing. Later on, the query engine can quickly look up the IDs of pages that match the terms provided by a user’s search query.

In this chapter we’re going to look at solving search related problems in Ruby from two angles. Firstly, we’re going to look at standalone libraries and techniques available to Ruby developers, and will walk through how to index content using two Apache Lucene-based libraries, Ferret and Solr, as well as a performance-driven Ruby-only library called FTSearch. Secondly, we’ll look at integrating search features with other technologies, and look at searching the Web, searching databases, and adding indexing and search features to Ruby on Rails applications.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

It is important to note that we are only going to cover search libraries that are under active development and that have been updated in the last year. There are several, older libraries that you may hear about, but we feel it's important to focus on up-to-date, well supported tools so that the level of documentation and support you would expect is still present.

12.2 Standalone and High Performance Searching

In this section we're going to look at generic, standalone searching and indexing scenarios, with the simplest problem being that to index, and then query, a corpus of documents that we can provide. This contrasts with the latter half of this chapter where we will look at how to use and integrate search techniques in busier situations, such as on the Web or within a database-driven Web application.

12.2.1 Standalone Indexing and Search with Ferret

Ferret is a Ruby implementation of Apache Lucene, an open-source search and indexing library written in Java. Lucene is incredibly popular in the open source world and a significant amount of software, as well as libraries, use it for implementing search systems large and small. Lucene is just a indexing and search library, and so does not include any features relating to obtaining or specially parsing content. These features are provided by other libraries, or by the software using Lucene. As a Ruby implementation of Lucene, Ferret shares the same characteristics.

Ferret will not crawl the Web for you, download e-mails, or index different types of content in unique ways. Instead, you have to use Ferret from your Ruby programs in a generic way. Like Lucene, Ferret can be agnostic to different data formats, and as long as you can extract the textual content of the data you wish to index, Ferret can handle it. Ferret works with the concepts of *documents* and *fields*, where documents represent individual groups of content to index (such as a single Web page, an e-mail, or the lyrics to a song) and fields are more detailed elements of data with documents (such as dates, author information, and other metadata).

In this section we're going to look at using Ferret to index and search through documents we provide it.

Problem

We wish to be able to index, and then search via query, an arbitrary set of documents (that may or may not contain multiple “fields” of metadata, such as titles, descriptions, and author information) quickly and efficiently. We do not care if the index is usable only from Ruby.

Solution

Three solutions to the problem are presented. The first implements a basic text-only indexing and searching system. The second looks at indexing and searching through content that contains metadata and multiple fields. The third solution looks at storing an index to disk and then loading it from another program (necessary so that your index can persist).

Each solution assumes that you have installed the `ferret` gem. This is very simple to do on a system running Ruby and Rubygems, with `gem install ferret`. Let's take a look at an example of a Basic Document Search in Listing 12.1.

Listing 12.1: Basic Document Search

```
require 'rubygems'
require 'ferret'
```

```
index = Ferret::Index::Index.new
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```

index << "This is a test document to test Ferret."
index << "Here is another test document."
index << "And now, for something totally different!"

query = "test"

index.search_each(query) do |id, score|
  puts "Found #{query} in document #{id} (scoring #{score})"
end

```

Listing12.2: Multi-field Document Search

```

require 'rubygems'
require 'ferret'

# Begin to build a set of fields for content to use
fields = Ferret::Index::FieldInfos.new(:store => :yes, :index => :yes)

# Give a boost to titles, as their content is likely to confidently reflect
the document
fields.add_field(:title, :boost => 10)

# No boost for general content
fields.add_field(:text)

# Give a big boost to author information in terms of searches
fields.add_field(:author, :boost => 20)

# Build a local index object
index = Ferret::Index::Index.new(:field_infos => fields)

# Add some example documents to the index
index << { :title => "My Test Document",
           :text  => "This is a test to demonstrate Ferret",
           :author => "Fred Bloggs" }

index << { :title => "Irrelevant Title",
           :text  => "Another test document",
           :author => "Anon" }

index << { :title => "Third Document",
           :text  => "This document contains nothing of interest",
           :author => "Fred Bloggs" }

# Perform a query
puts "What do you want to search for?"
query = gets.chomp
search_results = index.search(query).hits

# Sort the search results by score descending and show the results
search_results.sort_by{ |h| h.score }.reverse.each do |hit|
  puts "Found '#{index[hit.doc][:title]}' by '#{index[hit.doc][:author]}'
(score: #{hit.score})"
end

```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

Listing 12.3: Separate Indexer and Query Client Programs

Indexer

```
require 'rubygems'
require 'ferret'

fields = Ferret::Index::FieldInfos.new(:store => :yes, :index => :yes)
fields.add_field(:title, :boost => 10)
fields.add_field(:text)
fields.add_field(:author, :boost => 20)

# Create an index on the filesystem (stored in /tmp/our_index)
index = Ferret::Index::Index.new(:path => '/tmp/our_index',
                                :field_infos => fields)

# Add some example documents to the index
index << { :title => "My Test Document",
           :text  => "This is a test to demonstrate Ferret",
           :author => "Fred Bloggs" }

index << { :title => "Irrelevant Title",
           :text  => "Another test document",
           :author => "Anon" }
```

Query Client

```
require 'rubygems'
require 'ferret'

index = Ferret::Index::Index.new(:path => '/tmp/our_index')

# Perform a query
puts "What do you want to search for?"
query = gets.chomp
search_results = index.search(query).hits

# Sort the search results by score descending and show the results
search_results.sort_by{ |h| h.score }.reverse.each do |hit|
  puts "Found '#{index[hit.doc][:title]}' by '#{index[hit.doc][:author]}'
(score: #{hit.score})"
end
```

Discussion

The three solutions above are all very similar, but just show different approaches and levels of complexity. All of them use a Ferret index, with the first two solutions simply storing the index in memory for immediate use only, and the final solution storing and loading the index to and from the disk.

The Basic Process

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

The first solution shows how simple it is to create an index, by creating a new object from the `Ferret::Index::Index` class:

```
index = Ferret::Index::Index.new
```

Next, we supply the index with multiple “documents” to index. We used strings, but these can be almost any form of data in Ruby that can translate to a string (such as an array or a hash).

```
index << "This is a test document to test Ferret."
index << "Here is another test document."
index << "And now, for something totally different!"
```

Whether using an in-memory or on-disk index, simply “pushing” documents to the index causes them to be indexed immediately.

Lastly, we query the index using the `search_each` method, a method that performs a search and then iterates over each result, passing in the document ID of the matching document, along with a quality score, each time:

```
index.search_each(query) do |id, score|
  puts "Found #{query} in document #{id} (scoring #{score})"
end
```

We did not give our documents ID numbers, but Ferret did this for us in the order that we supplied the documents. For example, the first solution performs a query of, simply, “test,” which nets the following results:

```
Found test in document 0 (scoring 0.70710676908493)
Found test in document 1 (scoring 0.5)
```

Since the word “test” wasn’t mentioned in the third document (that is, the document that has an ID of 2, due to zero-indexing), it’s not returned as a result.

Metadata / Fields

The second solution shows how this can be extended with an option to include information about a set of “fields” that exist on the supplied documents. We define these fields first into a group using class `Ferret::Index::FieldInfos`, and then pass that composite object through to the index as an option.

Defining fields to be indexed and managed separately by Ferret is easy. First, you create the `FieldInfos` object that will hold all of the information:

```
fields = Ferret::Index::FieldInfos.new(:store => :yes, :index => :yes)
```

This constructor takes many different options, but the important ones are `:store` and `:index`. These options act as the default choice for all the fields we define from here on out. `:store` lets us choose if the index will store the *actual content* of a field (or whether it should be compressed and/or processed then discarded entirely) and `:index` specifies whether a field should be indexed at all. In our case we want the default to be yes to both.

Next in the solution we moved on to actually defining three fields, a “title” field, a “text” or content field, and an author field.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```
fields.add_field(:title, :boost => 10)
fields.add_field(:text)
fields.add_field(:author, :boost => 20)
```

Adding the fields is as easy as calling `add_field` on the `FieldInfos` object. Then you can specify the field name, along with any options. In this case the default options of `:store => :yes` and `:index => :yes` are used on all of the fields, but on two of the fields we provide a “boost.”

Boosting is useful when you have columns that contain data that’s more important than that in other columns. In this case, for example, we give the title of a document more importance in the rankings than the plain content (in field ‘text’). We then give the author name even more importance than the title, meaning that if one document contains “Fred” in the title, and another was written by someone named “Fred”, then the latter document would probably score more highly.

Once the fields are defined in the `FieldInfo` set, we can define the index in a similar way to in our first example, but with the addition of passing through the field data:

```
index = Ferret::Index::Index.new(:field_infos => fields)
```

Then, to add documents with defined fields, we can use hashes:

```
index << { :title => "My Test Document",
           :text  => "This is a test to demonstrate Ferret",
           :author => "Fred Bloggs" }
```

As the fields are delimited in this sample “document,” Ferret knows how to handle it in relation to the fields defined in the index.

Running this example and providing it with a test query shows us how the boosts affect things:

```
What do you want to search for?
> document
Found 'Third Document' by 'Fred Bloggs' (score: 1.97577941417694)
Found 'My Test Document' by 'Fred Bloggs' (score: 0.922030389308929)
Found 'Irrelevant Title' by 'Anon' (score: 0.065859317779541)
```

Notice that even though each document contains the word “document,” the document that ranks highest is the one with the word in both the title and the content, whereas the second result lacks the word in the content, and the third document, with an extremely low score, merely contains it in the “text” field.

Storage

The third solution demonstrates how to store and retrieve an index from disk. Ferret makes it extremely simple, with it only being necessary to specify the pathname within the construction of the index object:

```
index = Ferret::Index::Index.new(:path => '/tmp/our_index',
                                :field_infos => fields)
```

If the directory specified using the `:path` parameter doesn’t exist, it’ll be created, as it is in the first example of this third solution.

In the second example, the index is loaded in much a similar fashion:

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```
index = Ferret::Index::Index.new(:path => '/tmp/our_index')
```

Because the index should already exist, and the field information pre-defined, we don't need to construct and pass through the field information this time, as it's already part of the index's structure.

The rest of the third solution then uses the same querying code as used earlier.

FURTHER INFORMATION

While the queries we performed in the solutions were simple, single-word queries, Ferret has support for complex queries, much as you'd expect from a search tool. You can search for phrases, perform boolean operations (e.g. "fred OR martha" or "foo AND bar"), and use wildcards (e.g. "fre*").

You can learn more about the query syntax supported by Lucene at <http://lucene.apache.org/java/docs/queryparsersyntax.html>

You can also learn more about how to use Ferret by looking at the official tutorial at <http://ferret.davebalmain.com/api/files/TUTORIAL.html>

Next, we're going to look at a true Apache Lucene install, installed and made available remotely by another Apache product, Solr.

12.2.2 Integrating with Solr, an external, non-Ruby Lucene-based search engine

In the last section we looked at using Ferret, a Ruby implementation of Apache Lucene, to index and search data. As we discovered, Ferret and Lucene are simply indexing and searching libraries, and the ability to parse data to index, as well as interpret the results for searches, rests with the client application.

In this section, we're going to look at Lucene from a different angle, using the Solr search server. Solr, another Apache project, is an open source search server that uses Lucene. Whereas Lucene is only a searching and indexing library, Solr provides higher-level features such as XML and JSON HTTP based APIs, replication, caching, and a Web-based administration interface. If Lucene is the "guts" of a searching and indexing system, Solr provides the friendly face necessary to use it at a higher level.

Whether you choose to use Ferret or Solr depends on your preferences and the fit between your requirements and the pros and cons of each technology. Even though both provide Lucene-based functionality, the interface used with each is so radically different that careful consideration is required. Solr wins out if you need features like replication, the ability to rapidly scale, the ability to easily access indexes over a network, or you want to provide the same index to multiple applications, including non-Ruby applications. Ferret wins out if you want a simple, single-machine, Ruby-only solution, as it can be installed in one step using RubyGems, whereas Solr requires you to install several pieces of software just to get started.

The solution covered in this section expects that you have Apache Solr installed and running correctly. The installation of Solr is beyond the scope of this chapter, but the official home page is at <http://lucene.apache.org/solr/> and information about its dependencies, such as Java 1.5 (or higher), and how to download Solr, is available at the main tutorial provided on the site.

Problem

We wish to be able to use a Solr installation to index and query a set of documents, whether the Solr server is local or remote. This will give us the ability to perform searches over HTTP and be able to use Solr's features to gain access to a more robust, scalable, cross-application search solution.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

Solution

In our solution in Listing 12.4, we will assume Solr is installed and running, and that access to the admin interface is possible at `http://localhost:8983/solr/admin/`. If the server is running on a different machine or port, simply replace references of `localhost` to the relevant hostname, IP address and/or port in the code.

Listing 12.4: MySolr Class

```
require 'net/http'
require 'uri'
require 'open-uri'

# Very basic client library for Solr
class MySolr
  def initialize(server)
    @server = server
  end

  def <<(doc)
    # Put together the XML for Solr in a basic fashion
    xml = %Q{<add><doc>}
    doc.each do |key, value|
      xml << %Q{<field name="#{key}">#{value}</field>}
    end
    xml << %Q{</doc></add>}

    # Send the request
    make_xml_request xml
  end

  # Commit any new documents to the index
  def commit
    make_xml_request %Q{<commit waitFlush="false"
                        waitSearcher="false" />}
  end

  # Delete an item from the index
  def delete(id)
    make_xml_request %Q{<delete fromCommitted="true">
                        <id>#{id}</id></delete>}
  end

  # Perform a query / search on the Solr server
  def query(q)
    params = %Q{?q=#{q}&wt=ruby}
    uri = @server + "select/" + params
    eval(open(uri).read)['response'] rescue nil
  end

  private

  # Makes an HTTP POST request to Solr and sends the data we supply
  def make_xml_request(body)
    uri = URI.parse(@server + "update")
    res = Net::HTTP.new(uri.host, uri.port).start do |http|
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```

    http.post(uri.path, body, { 'Content-Type' => 'text/xml' })
  end
  raise "Error - #{res.class}" unless res.class == Net::HTTPOK
end
end

```

A class is specially provided here, as current libraries available are focused on integration with Ruby on Rails, rather than for use directly from Ruby. As such, this basic library demonstrates how Solr works at a deeper level, by making the HTTP requests directly. Once you become familiar with Solr, you may choose to take a different approach, or move to using one of the Rails / ActiveRecord based solutions.

This class is used by the remainder of this solution, so include it at the top of the following items of code (Listings 12.5 and 12.6) or “require” it in.

Listing 12.5: Adding and Indexing Documents

```

# Create an index object using the MySolr class
index = MySolr.new("http://localhost:8983/solr/")

# Add documents to the index
index << { :id    => 1,
           :name  => "My Test Document",
           :text  => "This is a test to demonstrate Solr" }

index << { :id    => 2,
           :name  => "Irrelevant Title",
           :text  => "Another test document" }

# Commit the documents to the index
index.commit

```

Listing 12.6: Querying the Index

```

results = index.query('title')

# Print the results
puts "#{results['numFound']} result(s) found!"
puts
results['docs'].each do |result|
  puts "Document ID: #{result['id']}"
  puts "      Title: #{result['name']}\n\n"
end

```

Running this code after the indexing routine above should get a result like the following:

```

1 result(s) found!

Document ID: 2
      Title: Irrelevant Title

```

Listing 12.7: Deleting Items from the Index

```

index.delete(1)
index.delete(2)
index.commit

```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

Discussion

As a remotely accessible search engine, most of the code that makes up the `MySolr` library is concerned with making HTTP connections and sending XML data around. This contrasts with our earlier experiments with Ferret which work in a very natural, Ruby coded way. Solr, on the other hand, accepts XML instructions over HTTP, although its internal operation is somewhat similar to that of Ferret.

The `MySolr` class is not to be considered a particularly reliable Solr client library, although it works well for our demonstrations here, and to see how to integrate with Solr at the HTTP level. One major flaw with the library is that it doesn't construct XML documents in a reliable way, although the intricacies of building XML documents is too long-winded and beyond the scope of this chapter.

The solution depends on a Solr server being installed and presently running at the time of execution. It also expects you to be using the "example" Solr server that's built by default when you install Solr. This "example" server includes a schema with pre-defined field types that we use in the solution (namely, "id", "name", and "text"). If you wanted to build your own schema from scratch, you would need to cater for this in the documents you index from Ruby, and ensure that they only present the legitimate fields to Solr, otherwise an error will result.

From the client's point of view, using `MySolr` looks similar to how we used the Ferret library earlier. We create an index and simply push documents on to it:

```
index = MySolr.new("http://localhost:8983/solr/")
index << { :id    => 1,
          :name  => "My Test Document",
          :text  => "This is a test to demonstrate Solr" }
```

However, querying is quite different. Solr accepts queries over HTTP to a URL like so:

```
http://hostname:port/solr/select?q=query
```

If you have Solr running, you can use such a URL and see the results come back in XML format. For our purposes, however, we add an extra option that makes Solr return the results in a "Ruby friendly" format, by using a URL like so:

```
http://hostname:port/solr/select?q=query&wt=ruby
```

This technique is apparent in the query method of `MySolr` when we're putting the URL together:

```
params = %Q{?q=#{q}&wt=ruby}
uri = @server + "select/" + params
```

Solr will return a string that can be "evalled" by Ruby that looks similar to this:

```
{ 'responseHeader'=>{ 'status'=>0, 'QTime'=>0, 'params'=>{ 'q'=>'title', 'wt'=>'ruby' } }, 'response'=>{ 'numFound'=>1, 'start'=>0, 'docs'=>[ { 'name'=>'Irrelevant Title', 'id'=>'2', 'sku'=>'2', 'popularity'=>0, 'timestamp'=>'2007-07-12T03:56:49.618Z' } ] } }
```

We evaluate this and return the relevant section from the query method like so:

```
eval(open(uri).read)[ 'response' ] rescue nil
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

This line downloads the results (using `open-uri`'s convenient technique), evaluates the Ruby-friendly text response, and then returns only the “response” section, as the “responseHeader” section only contains information about the request and how long it took to complete.

The information that gets returned to the main client is then a hash that looks like this:

```
{'numFound'=>1, 'start'=>0, 'docs'=>[{'name'=>'Irrelevant  
Title', 'id'=>'2', 'sku'=>'2', 'popularity'=>0, 'timestamp'=>'2007-07-  
12T03:56:49.618Z'}]}
```

With this information it's rudimentary to walk through the information and present the results.

You can learn more about Solr's HTTP API calls and how to index data and configure Solr at the Solr Wiki at <http://wiki.apache.org/solr/FrontPage>

Later in this chapter we will look, briefly, at how to use Solr from a much higher level from a Ruby on Rails application, where the `acts_as_solr` plugin takes care of all of the intricacies of XML and HTTP for us.

12.2.3 Ultra-fast indexing and searching of a corpus / set of files with FTSearch

The previous two sections have looked at Apache Lucene based searching and indexing tools. In this section we're going to look at a more grassroots, specialized, and high performance library called FTSearch.

FTSearch is a performance-focused search library written primarily in Ruby (with a little C, for performance) by Mauricio Fernandez. FTSearch began life when Fernandez wanted to create a tool that could search Ruby's documentation faster than the existing *ri* tool. He decided to start by writing a basic suffix-array based full-text search engine, and his first pure Ruby implementation - achieved in merely 300 lines - was already many times faster than *ri*'s existing search.

With further work, and the addition of a little C code, FTSearch became so fast that Fernandez's tests instead began to use the source code for Linux than the trivial amount of Ruby documentation available. At the end of 2006, Fernandez's benchmarks showed FTSearch was outperforming Ferret (and therefore the Lucene approach in general) by between 3 and 30 times.

These characteristics, coupled with FTSearch's current lack of many features Ferret and Solr provide, make FTSearch ideally equipped for situations where you have a massive set of text files (such as source code) and all you need to do is be able to index and perform queries on their raw contents very rapidly to find matches.

Problem

You wish to be able to index, and then search via query, an arbitrary set of documents or files extremely quickly, without using an Apache Lucene derivative. Raw indexing and query performance are the overriding considerations as opposed to features.

Solution

Our solution involves an indexer (Listing 12.8), which will define the fields, iterate the files and index them, and a query script (Listing 12.9) that will query the index we created.

Listing 12.8: Indexer

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```

# Add ./lib and ./ext/ftsearch to the path search for "require"s
$:.unshift *%w{lib ext/ftsearch}

# Load the necessary parts of the FTSearch library
require 'ftsearch/fragment_writer'
require 'ftsearch/analysis/simple_identifier_analyzer'
require 'ext/ftsearch/ftsearchrt'

# Define a set of fields for the index to use, as with our Ferret example
field_infos = FTSearch::FieldInfos.new
field_infos.add_field(:name => :uri, :analyzer =>
  FTSearch::Analysis::SimpleIdentifierAnalyzer.new)
field_infos.add_field(:name => :content, :analyzer =>
  FTSearch::Analysis::WhiteSpaceAnalyzer.new)

# Create an index thta uses the fields defined above
index = FTSearch::FragmentWriter.new(:path => "testindex",
  :field_infos => field_infos)

# Get a list of files to index
files_to_index = Dir["data/**/*.*"]

# Go over each file and add it to the index
files_to_index.each do |file|
  puts "Adding to index: #{file}"
  index.add_document(:uri => file, :content => File.read(file))
end

# Write index to disk from memory
index.finish!

```

Listing 12.9: Querying the Index

```

# Add ./lib and ./ext/ftsearch to the path search for "require"s
$:.unshift *%w{lib ext/ftsearch}

# Load the necessary parts of the FTSearch library
require 'ftsearch/suffix_array_reader'
require 'ftsearch/fulltext_reader'
require 'ftsearch/document_map_reader'
require 'ftsearchrt'

# Set up readers for each stage of the searching and ranking process
fulltext_reader = FTSearch::FulltextReader.new(:path =>
  "testindex/fulltext")
suffix_array_reader = FTSearch::SuffixArrayReader.new(fulltext_reader,
  nil, :path => "testindex/suffixes")
doc_map_reader = FTSearch::DocumentMapReader.new(:path =>
  "testindex/docmap")

# Get a query from the user
puts "What do you want to search for?"
query = gets.chomp

# Get the raw "hits" for the query in the index

```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```

hits = suffix_array_reader.find_all(query)
puts "#{hits.size} found"

offsets = suffix_array_reader.lazyhits_to_offsets(hits)
puts "#{offsets.size} documents found"

# Sort the documents based on weights of fields
# This gives the first field "uri" more weight than the content
sorted = doc_map_reader.rank_offsets(offsets, [1000000, 10000])

# Print the sorted results
puts "Results:"

sorted.each do |doc_id, score|
  puts "Document ID: #{doc_id}"
  puts "      Score: #{score}"
  puts "      Filename: #{doc_map_reader.document_id_to_uri(doc_id)}\n\n"
end

```

If the source code to the Linux 0.01 kernel (<http://www.kernel.org/pub/linux/kernel/Historic/linux-0.01.tar.gz>) is extracted into the “data” directory, and the above code is run and a query of “Linus” is supplied, the results should be like the following:

```

2 found
2 documents found
Results:
Document ID: 68
      Score: 2
      Filename: data/linux/kernel/vsprintf.c

Document ID: 28
      Score: 1
      Filename: data/linux/include/string.h

```

If you want to run your own tests, make sure to provide a large corpus, such as the large collection of source code found in the Linux kernel. FTSearch’s techniques are not suited to small sets of data, and as FTSearch is suited to indexing and searching large sets of data, it makes sense to test it out!

Discussion

Before we look at how the FTSearch library is used, it’s necessary to cover its installation process.

Installation

As a pre-release library, FTSearch requires a manual installation. The library, in its current state, can be installed using the Darcs package manager (<http://darcs.net/>), like so:

```
darcs get http://eigenclass.org/repos/ftsearch/head/
```

Of if you don’t want to download and install Darcs, you can mirror or manually download all of the files directly from <http://eigenclass.org/repos/ftsearch/head/>

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

Once you have all of the files, the README file explains how to compile the C portion of the library. On most systems this is as easy as performing:

```
cd ext/ftsearch && ruby extconf.rb && make
```

Once these steps are complete, the examples in the solution will work.

Usage

The FTSearch solution is split into an indexer and a program that performs queries upon the index. The indexer looks very similar to the Ferret indexer. Fields are defined, again using a `FieldInfos` class:

```
field_infos = FTSearch::FieldInfos.new
field_infos.add_field(:name => :uri,      :analyzer =>
                    FTSearch::Analysis::SimpleIdentifierAnalyzer.new)
field_infos.add_field(:name => :content, :analyzer =>
                    FTSearch::Analysis::WhiteSpaceAnalyzer.new)
```

Notice that we specify different “analyzers” to be used on the two fields. The content is indexed with each word being parsed once broken up by whitespace. Filenames, however, are divided up by other separators, and the “simple identifier analyzer” simply looks for groups of alphanumeric characters, whether separated by whitespace or not. This will allow us to search for words within filenames with the query client.

The rest of the indexer simply retrieves a list of files from the “data” subdirectory and provides them to the index object to be indexed. Finally, the `finish!` method is called on the index to write the data to disk.

The querying program looks a little more complex. Due to FTSearch’s young age, it’s necessary to do more work than you do with Ferret. We have to require quite a few library files (which, in future, would hopefully be replaced with a single `require “ftsearch”!`) and create instances of readers for the different type of files in the index. Usually these functions would be abstracted away within a library, and it’s anticipated that this will happen with FTSearch in future.

Once the reader objects are prepared and the user has supplied a query, the first step is to get a list of all of the “raw hits” for the query:

```
hits = suffix_array_reader.find_all(query)
```

With the list of raw hits in the bag, it’s time to refer them to the documents, and then sort our results by the scores:

```
offsets = suffix_array_reader.lazyhits_to_offsets(hits)
sorted = doc_map_reader.rank_offsets(offsets, [1000000, 10000])
```

The second argument to `rank_offsets` is an array with weights for each field used in the index. Whereas with Ferret the “boost” values could be provided up-front and stored in the index, with FTSearch it is currently necessary to provide these figures at the time of search. Therefore, we’re giving words within the filename a “boost” of 1,000,000, and words within the files themselves a boost of only 10,000.

Displaying the sorted results is easy, as `rank_offsets` returns an array containing document IDs and scores:

```
sorted.each do |doc_id, score|
  puts "Document ID: #{doc_id}"
  puts "      Score: #{score}"
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```
puts "    Filename: #{doc_map_reader.document_id_to_uri(doc_id)}\n\n"
end
```

The FTSearch installation comes with a README file and several excellent examples, all of which are more in-depth than the simple solution demonstrated here. By reading through the examples, the full power of FTSearch becomes apparent. A FTSearch-vs-Ferret benchmarking tool is also included so you can see FTSearch's significant speed advantages for yourself.

Next we're going to change gears significantly, move away from the dry, technical world of directly interfacing with indexing and querying routines, and rely on more automated indexing solutions.

12.3 Search Integrated with Other Technologies

In this section of the chapter we're going to look at more complex examples than we have till now. Whereas we have focused so far on indexing and querying various collections of local data, we are now going to look at how we can use and integrate searching and indexing techniques with other technologies, such as the Web and database-driven applications.

12.3.1 Web Search using a basic API with structured results (Yahoo!)

In the eyes of many, search means "Google." Of course, we know there's a lot more to search than that, but it's impossible to escape that, to many people, "search" relates to the ability to search the Web. This isn't a bad assumption to make, however, as the Web, coupled with search engines like Google and Yahoo!, provides a relatively device and format agnostic way to search large sets of data. In this chapter we're going to look at how you can perform your own Web searches with Ruby on both Google and Yahoo! using two different techniques, a search API and "scraping."

In this solution we'll query Yahoo and process the results it returns in a basic XML format, although a similar approach can be used for many systems that provide basic APIs where the query can be specified on a URL and the results are returned in a structured format (such as searches made with Amazon.com's developer APIs, for example).

Problem

You wish to be able to perform Web searches using search engines, such as Yahoo!, that return structured data.

Solution

Let's use the Yahoo! Web Search API to search for the 5 top results for "ruby":

```
require 'open-uri'
require 'rexml/document'

# Specify some parameters for the search
results = 5
query    = 'ruby'
appid    = 'YahooDemo'

# Construct the URL using our parameters and retrieve the XML data
url = "http://api.search.yahoo.com/WebSearchService/V1/webSearch"
url += "?query=#{query}&appid=#{appid}&results=#{results}"
xml = open(url).read
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```
# Print search results extracted from XML
REXML::Document.new(xml).elements.each('ResultSet/Result') do |result|
  puts result.text('Title') + " => " + result.text('Url')
end
```

The result of running this program looks something like this:

```
Ruby Programming Language => http://www.ruby-lang.org/en
Ruby (programming language) - Wikipedia, the free encyclopedia =>
http://en.wikipedia.org/wiki/Ruby_programming_language
Ruby Central => http://www.rubycentral.com/
Ruby Annotation => http://www.w3.org/TR/ruby/
Ruby Programming Language => http://ruby-lang.org/
```

These results show the results (at the time of writing) for a Yahoo! Web Search using the query “ruby.”

Discussion

The Yahoo! Web Search API represents, to us as programmers, an excellent Web search API because the results are returned in a programmer-friendly XML format. This format has been defined by Yahoo! and will remain consistent and documented (at Yahoo’s developer site).

Yahoo! provide access to their Web Search API via simple URLs. The first half of our program is concerned entirely with loading up the libraries we need to use (code omitted below) and building up one of these URLs:

```
results = 5
query    = 'ruby'
appid    = 'YahooDemo'
url = "http://api.search.yahoo.com/WebSearchService/V1/webSearch"
url += "?query=#{query}&appid=#{appid}&results=#{results}"
```

The basic parameters required by Yahoo! are a number of results to return (in this case, 5), an application ID, and a query (in this case, “ruby”). There are about ten others to access advanced features like removing adult sites, choosing sites from certain countries, and so forth, and these are covered in detail at <http://developer.yahoo.com/search/web/V1/webSearch.html>.

The application ID in our example is “YahooDemo,” which Yahoo! provides for test purposes only. It should work for a few queries, but if you want to use the Yahoo! Web Search API seriously, you’ll need to apply to Yahoo for your own application ID and agree to their terms and conditions. You can learn more about this, and the rest of Yahoo!’s APIs at <http://developer.yahoo.com/search/>.

After the URL has been put together, we use `open-uri`’s useful `open` method to download the XML:

```
xml = open(url).read
```

The resulting XML itself is too long and complex to print in this book, but its format becomes apparent from the code we use to parse it.

```
REXML::Document.new(xml).elements.each('ResultSet/Result') do |result|
  puts result.text('Title') + " => " + result.text('Url')
end
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

This code uses Ruby's standard XML process library, REXML, to iterate over each "Result" element of the XML, and then to print out the contents of the inner "Title" and "Url" elements in each case:

```
Ruby Programming Language => http://www.ruby-lang.org/en
Ruby (programming language) - Wikipedia, the free encyclopedia =>
http://en.wikipedia.org/wiki/Ruby_programming_language
Ruby Central => http://www.rubycentral.com/
Ruby Annotation => http://www.w3.org/TR/ruby/
Ruby Programming Language => http://ruby-lang.org/
```

Next we're going to move on to a rougher approach to search, a Google "screen scraping" approach.

12.3.2 Web Search using a scraping technique (Google)

Let's face it, Google is the grand daddy of search, so it's natural to want to be able to use its results in a programmatic fashion. Or how about other sites like the Internet Movie Database (IMDB) or a typical e-commerce site? Unfortunately we need to resort to the dark magic of "scraping."

Until the end of 2006, Google provided access to a SOAP-based search API, but this has been withdrawn from use for all except existing users. As this system is now deprecated and closed to the public, we cannot go into its operation. Google replaced the SOAP-based API with an "AJAX-based" one to be used directly on Web pages, but this API is not of significant use to us.

It should be noted that "scraping" search results in an automated fashion is against Google's terms of service, although it is generally allowed for non-automated, personal use. We expect you to use any code provided in this section with the best of faith and in compliance with Google's terms of service (or any other site you choose to scrape).

Problem

You wish to be able to perform Web searches and extract data from search engines or other Web sites that don't present their results in a structured way. This forces us to use a "scraping" technique.

Solution

Let's scrape Google for results for the search query "ruby":

```
require 'rubygems'
require 'scrubyt'

query = "ruby"

results = Scrubyt::Extractor.define do
  fetch 'http://www.google.com/'
  fill_textfield 'q', query
  submit

  result('/html/body/div/div/div', { :generalize => true }) do
    link_title '/a[1]'
    link_url '/a/@href'
  end
end

puts results.to_xml
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

The result from this program, after a whole collection of raw debugging information, should look something like this:

```
<root>
  <result>
    <link_title>Ruby Programming Language</link_title>
    <link_url>http://www.ruby-lang.org/</link_url>
  </result>
  <result>
    <link_title>Ruby Home Page - What's Ruby</link_title>
    <link_url>http://www.ruby-lang.org/en/20020101.html</link_url>
  </result>
  <result>
    <link_title>Ruby (programming language) - Wikipedia, the free
    encyclopedia</link_title>

<link_url>http://en.wikipedia.org/wiki/Ruby_programming_language</link_url>
  </result>
  <result>
    <link_title>Ruby - Wikipedia, the free encyclopedia</link_title>
    <link_url>http://en.wikipedia.org/wiki/Ruby</link_url>
  </result>

  [.. extra results removed to preserve space ..]
</root>
```

Note that this solution depends on the `scrUBYt!` library, available as a gem and easy to install using:

```
gem install scrubyt
```

On some platforms you may also need to install an extra gem (only do this if there is an error):

```
gem install ParseTreeReloaded
```

Discussion

As Google doesn't supply a programmer-friendly API, we had to resort to scraping Google's data from their regular HTML pages. As explained earlier, automating this process on a mass scale is against Google's terms and conditions, so tread with caution. That said, you will be able to use this same technique with other Web sites and search engines, and the scraping library, `scrUBYt!`, comes with many examples of using the library to scrape sites like IMDB, Yahoo Finance, Amazon.com, and so on.

`scrUBYt!` (<http://scrubyt.org/>) is a Ruby library developed by Peter Szinek that makes it easy to automate and process data on the Web. The conciseness of the main part of our code reinforces this:

```
results = Scrubyt::Extractor.define do
  fetch 'http://www.google.com/'
  fill_textfield 'q', query
  submit
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```

    result('/html/body/div/div/div', { :generalize => true }) do
      link_title '/a[1]'
      link_url '/a/@href'
    end
  end
end

```

The steps involved should be clear. As `scrUBYt!` features its own DSL (Domain Specific Language), we define an “extractor” through the `Scrubbyt::Extractor` class, and then the fun can begin. The first step of the extraction process is to fetch the Google.com homepage:

```
fetch 'http://www.google.com/'
```

That done, we can fill out the search form with our desired query and submit the form:

```
fill_textfield 'q', query
submit
```

In this case, it’s easy to see by looking at the source code that the Google search form uses a text field with a name of “q” to accept the user’s query, so we use `fill_textfield` to do this job for us, and submit the form.

Once the form is submitted, a results page will be returned to us, and we need to define a pattern to match against each result. `scrUBYt!` is powerful enough to allow you to specify example results, obtained by hand, and then work out the XPath rules to scrape that data from a page. In this case, however, to ensure that the example works I have specified the XPath rules explicitly:

```

result('/html/body/div/div/div', { :generalize => true }) do
  link_title '/a[1]'
  link_url '/a/@href'
end

```

The result block gives us access to each result (found with an XPath query of “/html/body/div/div/div”) and allows us to define which elements of the result we want to extract. In our case we’re extracting the title and URL of the links, although you could extract the result descriptions too, with an extra rule. These methods are just simple patterns, and could use any name, other than `link_title` and `link_url`. `scrUBYt!`’s DSL is clever enough to work out these are names we are giving to the respective elements, and will use these names in the resulting XML. For example, this code is just as valid and would result in XML output using slightly different element names:

```

result_title '/a[1]'
result_url '/a/@href'

```

The resulting XML that comes from the following line can then be processed using any XML library, such as `REXML` or `SimpleXML`:

```
results.to_xml
```

Next we’re going look at indexing and searching data in a more traditional, local form once again, upon databases used by our applications.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

12.3.3 Indexing and searching data in Rails applications with Ferret and Solr

Ruby on Rails is a Web application framework that makes it easy to develop database-driven Web applications using Ruby. One of the key ways it makes life easier is by splitting up applications into three main areas of concern: Models, Views and Controllers (also known as the MVC pattern).

Rails predominantly uses a library called ActiveRecord that provides an object-oriented interface to data. Instead of using SQL to work on database entries directly, Rails programmers can define classes that represent “models” (and, in turn, database tables) in their application and operations performed upon those models are represented within the database automatically.

For example, if you were developing a blogging application with Rails, you might have model classes called things like User, Post, Category, Tag, and so on. You might then perform operations upon them like so:

```
fred = User.new
fred.age = 20
fred.name = "Fred Bloggs"
fred.save
```

The code in this example would create a new row in the `users` table of the database and assign values to the various columns.

In the solution covered in this section we’re going to assume that you either have a basic understanding of how to develop a Rails application, or that you’ve visited <http://www.rubyonrails.org/> and reached a stage where you can continue reading here.

Why use Ferret or Solr from Rails?

Rails and ActiveRecord provide an abstraction between classes, objects, and data stored in a database. Rails supports a number of different database systems, such as MySQL, Postgres, Microsoft SQL Server, and Oracle. Unfortunately, however, all of these database systems work in different ways.

Some of the database engines Rails supports include advanced full-text search features and others do not. Furthermore, different data and table types within each engine can have different search characteristics. This has meant that Rails and ActiveRecord have not been able to provide a generic, easy way to perform full-text searches on tables within Rails applications, instead leaving users to perform sloppy, non-portable SQL hacks like so (this example is for MySQL):

```
results = Post.find(:all, :conditions => "title LIKE '%search phrase%'")
```

The downside to the previous code is that it forces the database to go through every row in the “posts” table to find an entry with a title containing the necessary word or phrase. As we discussed in 12.1 “The Principles of Search,” this is considered to be “search” but without any indexing process it’s extremely inefficient and slow, particularly on larger datasets.

To get around the performance problems, therefore, it’s possible to use the Lucene-inspired Ferret library that we looked at in 12.2.1 to index data that’s in our database, and then query the index when we want to perform searches rather than the database directly. Once Ferret returns the correct document IDs, then we can use standard ActiveRecord methods to extract the data from the live database.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

Problem

You wish to be able to index, and then search via query, data stored in a single Rails model using a Ferret or Solr-based (along with the various benefits this allows, as discussed in section 12.2.2.) index.

Solution

We are going to focus on a Ferret based solution, but then also take a quick look at how to change the solution to work using Solr. As the interfaces between the two plugins involved is so similar, a full, second solution for Solr is not necessary, so the differences are covered in the Discussion section only.

Model

In `app/models/post.rb` (or whichever model you please):

```
class Post < ActiveRecord::Base
  has_many :comments
  belongs_to :user

  acts_as_ferret :fields => {
    :title => { :boost => 10 },
    :content => { }
  }
end
```

Querying the Index

This will work from models, controllers, and views:

```
Post.find_by_contents("test").each do |post|
  result_title = post.title
  result_score = post.ferret_score
end
```

Discussion

The solution is short, but there's a lot of background to it. Let's dig deep and see if it's really as simple as it looks! Let's assume you have a Rails application that implements a blogging tool, and features models such as `User`, `Post`, and `Comment`, relating to users of the application (people who are posting entries), the posts made, and comments made by visitors, respectively.

We would consider it useful for a blogging application to provide general search features over at least the posts within the system, and possibly the comments too. We don't want to resort to messy techniques, such as using `LIKE`, as covered in the introduction to this section, as we want to be able to support powerful search features like wildcards and phrase searches.

Note that this solution relies on having Ferret installed (as explained in 12.2.1), and the `acts_as_ferret` Rails plugin which can be installed simply:

```
gem install acts_as_ferret
```

Once `acts_as_ferret` is installed, just add this line to the bottom of your `config/environment.rb` file, and the remainder of the solution will work:

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```
require 'acts_as_ferret'
```

Once `acts_as_ferret` is installed and included into our Rails application, it becomes extremely trivial to use Ferret's features on our models. For example, our basic `Post` model (located in `app/models/post.rb`) will look somewhat like this:

```
class Post < ActiveRecord::Base
  has_many :comments
  belongs_to :user
end
```

Adding a single line gives us searching and indexing facilities:

```
class Post < ActiveRecord::Base
  has_many :comments
  belongs_to :user
  acts_as_ferret :fields => [:title, :content]
end
```

The `acts_as_ferret` line throws a whole set of gears into action that take care of indexing and tracking the “title” and “content” fields/attributes on our `Post` objects. A more powerful, style to follow, however, would be this:

```
acts_as_ferret :fields => {
  :title => { :boost => 10 },
  :content => { }
}
```

This style is more complex, but allows us to specify options for each field, much as we did in 12.2.1. Similar to our example in 12.2.1 we apply a “boost” to a particular field. This means that words contained within the title of a post are considered more important than those in the content. Since titles are usually more concise and targeted than the content of blog posts, this is a good use of the boost technique.

You can also specify whether Ferret should store the data it's indexing or not by using the `:store` option, much as we did in 12.2.1 also (by default, storing is off):

```
acts_as_ferret :fields => {
  :title => { :boost => 10, :store => :yes },
  :content => { }
}
```

Anyway, let's add some posts to be indexed to our database (you can do this using `script/console` to make it easier):

```
Post.create(:title => "Test Post", :content => "A test document!")
Post.create(:title => "Another Post", :content => "More test stuff!")
Post.create(:title => "Third Post", :content => "End of testing")
```

`acts_as_ferret` takes care of the indexing automatically, so you can immediately start making queries, using the `find_by_contents` method that `acts_as_ferret` adds to models it's supporting:

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```
Post.find_by_contents("test").each do |post|
  puts "#{post.title} - #{post.ferret_score}"
end
```

With our example data, we get:

```
Test Post - 1.0
Another Post - 0.025548005476594
```

Naturally, more complex examples are possible, as you get the full range of boolean, wildcard and other searches Ferret supports:

```
Post.find_by_contents("te*").each { |p| puts p.title }
```

It's also possible to query the index without actually returning each matching object. This could be useful if you just wanted to count the number of results, do pagination, or just get access to the metadata involved:

```
Post.find_id_by_contents("test*")
```

Which results in an array like the following:

```
[3, [{:score=>1.0, :model=>"Post", :id=>"1", :data=>{}},
{:score=>0.0157371964305639, :model=>"Post", :id=>"2", :data=>{}},
{:score=>0.0157371964305639, :model=>"Post", :id=>"3", :data=>{}}]]
```

`find_by_contents` also supports the `:limit` and `:offset` features that will be familiar from other finders:

```
Post.find_by_contents("", :limit => 1, :offset => 1)
```

This will return not the first result, but the second result, and this can also be useful for pagination.

You can learn a lot more about how `acts_as_ferret` works from the official wiki at http://projects.jkraemer.net/acts_as_ferret/wiki

Solr

`acts_as_solr` provides a very similar interface to `acts_as_ferret`, and its API documentation is available at http://api.railsfreaks.com/projects/acts_as_solr/

`acts_as_solr` is installed in the same way as most Rails plugins, using the `script/plugin` tool (requires Subversion to be installed):

```
ruby script/plugin install
svn://svn.railsfreaks.com/projects/acts_as_solr/trunk
```

`acts_as_solr` can use a remote Solr install (as we did in 12.2.2, you should even be able to use the same example install that we used previously) but it also comes with its own version of Solr, which makes things easier. To run it just type:

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```
cd vendor/plugins/acts_as_solr/solr/  
java -jar start.jar
```

Solr will then be running permanently, and you can continue developing on another shell/terminal.

Before we change any code in the Rails application, it's necessary that the configuration file in `db/solr.yml` is updated to use the correct address/port number for the currently running version of Solr. Update this for both production and development environments before continuing. The address of the Solr server will have been printed on-screen when you ran the Solr server.

Changing the `Post` model from the Ferret example is as easy as going to this:

```
class Post < ActiveRecord::Base  
  has_many :comments  
  belongs_to :user  
  
  acts_as_solr :fields => [{:title => {:boost => 10}}, :content]  
end
```

The syntax is a little different for specifying the boost (and any other options you might choose to use), but the technique is almost the same.

Querying is also similar:

```
Post.find_by_solr("test")
```

Or to get a full set of results:

```
Post.find_by_solr("test").records.each { |p| puts p.title }
```

12.4 Summary

This chapter has covered a tight niche, in terms of Ruby. As we've seen, there are only a handful of search libraries and techniques developed in Ruby so far, but all of them are reasonably powerful and ready to be used in production scenarios.

First we looked at the general principles of searching and indexing, before moving on to look at some high performance, stand-alone solutions in the guises of Ferret, a Ruby Apache Lucene port, Solr, an HTTP-based search server and interface to Lucene, and FTSearch, a high performance suffix-array based indexer and search library developed by Mauricio Fernandez. We then moved on to looking at search solutions that integrate with other technologies, such as `acts_as_ferret` and `acts_as_solr`, Rails plugins that bridge the gap between Rails applications and search libraries. Lastly, we looked at Sphinx, an indexer for "full text" collections, and a Ruby library to interface with it.

This chapter has shown you how to get your bearings with the search technologies available to you in Ruby. It's also shown you some further technologies and libraries you can explore and to look out for in the future.

In the next chapter we're going to move on to looking at document processing and report generation.

13 Document Processing and Reporting

Though we'd all love to be video game programmers or lead developers for the Next Big Web 2.0 multimedia experience, most programmers have less glamorous problems to solve. This inevitably includes processing documents exported from any number of sources, ranging from legacy systems to industry standard SQL databases. Of course, collecting the data and translating it into a useable form is only half of the challenge. Aggregated data then needs to be manipulated, analyzed, and formatted in ways that clearly communicate what that data actually represents. The field of reporting ranges from quick sales reports for a small business to massive statistical analysis for the enterprise.

There are any number of canned solutions for processing one form of document and translating it to another or running 'standard reports' against data. For many uses, these applications are the way to go. However, these solutions aren't golden hammers. When developing custom software, it's often desirable to get exactly the kinds of results you're looking for, instead of just a pre-built solution that may or may not be what you need. For this kind of problem, Ruby really shines.

In this chapter, we'll be showing several solutions to common reporting problems. In it, we'll cover processing and formatting libraries such as FasterCSV and PDF::Writer, but we'll be also looking at the lightweight reporting system, Ruport. Through these solutions, you'll be able to see how rolling your own solutions isn't as scary as it sounds. By providing a solid but malleable foundation to work from, Ruport provides a way to quickly build your own customized reporting applications.

We'll start with some simple CSV processing showing off FasterCSV's feature set, then show how to use Ruport's code generator to trivially script away a database backed report that automatically emails its results when run. We'll then show how to use Ruby's text processing capabilities to compare a CSV file to a non-standard data format. After covering the ins and outs of data manipulations, we'll dive into printable documents, showing how to leverage Ruport and PDF::Writer to generate attractive customized PDF reports. Finally, we'll talk about how to optimize for performance when dealing with large data sets in Ruport.

13.1 Processing CSV Data

Problem

One of the most common tasks in document processing is handling CSV data. Used as a generic text format for tabular data from any range of sources, the CSV format is simple and fairly easy to process. However, Ruby's standard library for handling CSVs is a bit awkward, and also very slow. The following example shows how to use FasterCSV for reading and writing CSV data, and also shows off some of the features which have made it the new defacto standard among Rubyists.

Here we'll show a simple tool that does currency conversion based on CSV data, to demonstrate many key features of FasterCSV.

Solution

Many websites will allow you to download a CSV table of exchange rates for world currencies. These might come in any number of forms, but to build a fairly general conversion tool, all that is needed is a set of currency codes and their conversion ratios. The data which this example is based off of looks like this:

```
Currency,Code,USD/1 Unit,Units/1 USD
Andorran Franc,ADF,0.1339,7.4659
Andorran Peseta,ADP,0.005372,186.167
Utd. Arab Emir. Dirham,AED,0.2723,3.6741
Afghanistan Afghani,AFA,0.01998,50.25
Albanian Lek,ALL,0.01136,91.678
Armenian Dram,AMD,0.002895,345.41
NL Antillian Guilder,ANG,0.5682,1.8
```

The following small utility performs conversions to and from any of the currencies listed in the file, using USD as the base currency for interchange.

```
class CurrencyConverter

  def self.load_data(file,currency="USD")
    @to_us = {"USD" => 1}
    @from_us = {"USD" => 1}

    FasterCSV.foreach(file, :headers => true, :converters => :numeric) do
|r|
      @from_us[r["Code"]] = r["Units/1 USD"]
      @to_us[r["Code"]]   = r["USD/1 Unit"]
    end
  end

  def self.convert(amount,options={})
    from = options[:from] || "USD"
    to   = options[:to]   || "USD"

    amount * @to_us[from] * @from_us[to]
  end

  def self.high_low_report
    high, low = @from_us.partition { |code,ratio| ratio < 1 }
    write_file high, "high.csv"
    write_file low, "low.csv"
  end

  private

  def self.write_file(data,filename)
    FasterCSV.open(filename,"w") do |csv|
      csv << ["Code","USD/1 Unit"]
      data.each { |record| csv << record }
    end
  end
end
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

end

As you can see, nothing about the implementation is particularly surprising. Two lookup tables are generated for rates when `CurrencyConverter.load_data()` is called. It is then possible to make arbitrary currency conversions.

For example, to convert 100 USD to Japanese Yen:

```
CurrencyConverter.convert 100, :to => "JPY"
```

You can actually go in reverse as well (back to USD):

```
CurrencyConverter.convert 12240.0, :from => "JPY"
```

The implementation also allows for non-US to non-US conversion:

```
CurrencyConverter.convert 100, :from => "GBP", :to => "JPY"
```

In addition to this conversion feature, you can see that this little utility also offers a simple reporting facility. It will determine which currencies are higher and which are lower in value than the USD and generate two CSV files, "high.csv" and "low.csv"

Providing that you have loaded a CSV with the exchange ratios, generating this report is easy:

```
CurrencyConverter.high_low_report
```

This outputs files like this:

high.csv:

```
Code,USD/1 Unit
XPT,0.0007819
OMR,0.386
GRD,0.7345
ITL,0.7345
DEM,0.7345
```

low.csv:

```
Code,USD/1 Unit
XCD,2.725
SAR,3.7509
RUB,25.6764
NOK,5.8243
NIO,18.743
```

As you can see, FasterCSV makes the task of CSV processing nearly trivial. Of course, it's worth discussing in a little more detail how the library works as well as the advantages it offers over Ruby's built in CSV standard library.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

Discussion

Why FasterCSV?

As the name implies, FasterCSV is much faster than Ruby's standard CSV processor. In most cases, it is actually about 800% faster. However, it has more advantages than just speed alone.

If we look back at the currency conversion code, the CSV processing seems to take a back seat, letting us focus on the actual problem we're trying to solve.

The reason it seems this way is because there are a lot of nice things that FasterCSV is handling for us.

Graceful header handling

If you work with Ruby's standard CSV processor, there is no direct support for CSV headers. This means that instead of saying `r["Code"]`, you'd be saying `r[2]` in our example, or rolling your own mapping of names to indices.

Whenever you give any of FasterCSV's reading methods the option `:headers => true`, it will try to use the first line of a CSV as the headers, and automatically allow you to refer to cells by column name in addition to their ordinal position.

Because CSVs might have non-unique column names, FasterCSV supports minimum ordinal indices. This simple IRB session shows how a row with two "a" columns can still be nicely handled using this feature:

```
>> a = FasterCSV.read("simple.csv", :headers => true)
>> a[0].to_a
=> [["a", "1"], ["b", "2"], ["c", "3"], ["a", "4"]]
>> a[0]["a"]
=> "1"
>> a[0]["a", 1]
=> "4"
```

Rubyists often enjoy using symbols for keys rather than strings. FasterCSV allows arbitrary conversions of data, and has a built in which makes this trivial:

```
>> a = FasterCSV.read("simple.csv", :headers => true,
?>                               :header_converters => :symbol)
>> a[0].to_a
=> [[:a, "1"], [:b, "2"], [:c, "3"], [:a, "4"]]
>> a[0][:a]
=> "1"
>> a[0][:a, 1]
=> "4"
```

In addition to header conversion, FasterCSV allows you to do conversions on your entire dataset, which we'll now take a look at.

Intelligent Data Conversion Filters

One thing you'll notice from our calculation code is that we never explicitly converted the ratio fields to `Float` objects. However, it's pretty clear why we didn't have to, if you look at the main loading code.

```
FasterCSV.foreach(file, :headers => true, :converters => :numeric) do |r|
  #...
end
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

The `:converters => :numeric` code tells FasterCSV to use a built in formatter to inspect the fields and convert numeric values to their proper ruby objects (`Fixnum`, `Float`, etc.)

This is something handy, as the alternative would be to do explicit `to_f` calls for each field that needs to be converted. Other built in converters are available, such as `:date`, `:datetime`, `:integer`, and `:float`

It is actually quite easy to build your own converters as well, if needed. This is left as an exercise to the reader, although the process is essentially as simple as passing Proc objects which accept a field as an argument and do any necessary manipulations.

Stream writing with I/O like semantics

You might have noticed that `FasterCSV.open()` looks a whole lot like `File.open()`.

```
FasterCSV.open(filename, "w") do |csv|
  csv << ["Code", "USD/1 Unit"]
  data.each { |record| csv << record }
end
```

The primary difference here is that instead of a file object, a FasterCSV object is yielded. This object will automatically convert arrays into CSV rows, doing any necessary escaping.

If we had wanted to output to a string instead of a file object, you can just use `FasterCSV.generate`, meaning our original code could be re-written as:

```
csv_string = FasterCSV.generate do |csv|
  csv << ["Code", "USD/1 Unit"]
  data.each { |record| csv << record }
end
```

This especially comes in handy when generating CSVs in the context of web applications, where you may wish to build files on the fly for download without ever storing them server side.

For the most part, the fact that FasterCSV in general acts a lot like a Ruby I/O object makes life a lot easier, since it is a very rare case to be working with CSV data without the need for I/O operations. Here we have only scratched the surface, of course.

Discussion Summary

Almost all programmers will encounter CSV processing jobs from time to time. Though we didn't cover all the edge cases, most jobs will involve some of the basic techniques shown here. If you'd like to get started with FasterCSV, you can find it on RubyForge distributed as a gem. Since it is pure ruby, it should work without modification on all platforms.

13.2 Generating and Emailing Daily Reports

Problem

A common task in reporting is to generating scheduled reports. Though these take on several flavors, they usually share the same core process: Query a database for records which match a given day or time period, process the report, and then shelve it off somewhere for use or archival. Here we'll look at a basic sales report to show how to solve this type of problem in Ruport.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

Solution

We use Rupert's code generator, `rope`, to script away the boilerplate code for us and cut down on hand configuration. This tool is part of the `ruport-util` package, which is officially maintained by the Ruby Reports core team.

The following commands create a basic code skeleton, and then create templates for our report and its renderer.

```
$ rope store_reports
$ cd store_reports
$ rake build report=daily_sales
$ rake build renderer=sales
```

This report will be run against a MySQL database and will be emailed automatically upon generation. The following configuration is the minimum necessary to do that.

config/environment.rb

```
require "ruport"

# Uncomment and modify the lines below if you want to use query.rb
#
Ruport::Query.add_source :default, :user => "root",
                                   :dsn  => "dbi:mysql:storefront"

Ruport::Mailer.add_mailer :default, :host => "mail.adelphia.net",
                                    :address => "gregory.t.brown@gmail.com"
```

We set up our main report to execute a simple query, and tell the report to render with the `Sales` renderer. The report generates text and PDF output, and emails both upon execution.

lib/reports/daily_sales.rb

```
require "lib/init"
require "lib/renderers"

class DailySales < Ruport::Report

  renders_with Sales

  def renderable_data

    query %q{ select product,quantity,customer_name from sales
              where sale_date = ? order by quantity }, :params => [today]

  end

  def today
    Date.today.strftime('%Y-%m-%d')
  end

end
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```

DailySales.generate do |report|
  report.save_as("sales_report.pdf")
  report.send_to("gregory.t.brown@gmail.com") do |m|
    m.subject = "Sales Report for #{report.today}"
    m.text     = report.to_text
    m.attach  "sales_report.pdf"
  end
end

```

Our renderer for this report is fairly simple. It simply calculates the total number of items sold in a day, and the amount of customers. The formatters display this information as well as the table that is returned from the SQL query.

lib/renderers/sales.rb

```

require "lib/init"

class Sales < Ruport::Renderer
  stage :report_table

  module Helpers

    def today
      Date.today.strftime("%m/%d/%y")
    end

    def total_sales
      data.sigma("quantity")
    end

    def total_customers
      data.column("customer_name").uniq.length
    end

  end

end

class Text < Ruport::Formatter::Text

  renders :text, :for => Sales

  def build_report_table
    output << "Sales Report for #{today}\n\n"
    render_table(data)
    output << "\n\nTotal Sales: #{total_sales}  " <<
      "Total Customers: #{total_customers}"
  end

end

class PDF < Ruport::Formatter::PDF
  renders :pdf, :for => Sales

```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```

def build_report_table

  title = "Sales Report for #{today}\n"

  render_table(data, :formatter => pdf_writer,
               :table_format => { :title => title })

  pad(10) do
    add_text "Total Sales: #{total_sales}  " <<
      "Total Customers: #{total_customers}", :justification =>
:center
    end
    render_pdf
  end
end
end

```

The report is invoked by running the following command:

```
$ rake run report=daily_sales
```

You can see the text output below, which is sent as the body of the email:

Sales Report for 07/17/07

```

+-----+
|  product  | quantity | customer_name |
+-----+
| Tollbooth |         1 | Joe Rasta     |
| Tomato    |         1 | Al Green      |
| Vacuum Cleaner |       3 | Joe Loop      |
| Vacuum Cleaner |       5 | Al Green      |
+-----+

```

Total Sales: 10 Total Customers: 3

The PDF output is sent as an email attachment, and looks like this:

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

Sales Report for 07/17/07		
product	quantity	customer_name
Tollbooth	1	Joe Rasta
Tomato	1	Al Green
Vacuum Cleaner	3	Joe Loop
Vacuum Cleaner	5	Al Green

Total Sales: 10 Total Customers: 3

By using rope, you're automatically given a common structure for your application, that is ideal for this sort of problem. At this point, the only task remaining is to hook up scheduling software (such as cron), but that is left as an exercise to the reader.

Discussion

For now, let's look at the various parts of this application in more detail.

When you run the rope command, you're actually generating a simple skeleton for Ruport applications. The files generated are listed on the screen when the command is run:

```
$ rope store_reports

creating directories..
store_reports/test
store_reports/config
store_reports/output
store_reports/data
store_reports/data/models
store_reports/lib
store_reports/lib/reports
store_reports/lib/renderers
store_reports/templates
store_reports/sql
store_reports/log
store_reports/util
creating files..
store_reports/lib/reports.rb
store_reports/lib/helpers.rb
store_reports/lib/renderers.rb
store_reports/lib/init.rb
store_reports/config/environment.rb
store_reports/util/build
store_reports/util/sql_exec
store_reports/Rakefile
store_reports/README
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

In our report, we needed to configure our database and mailer information, which was done in *config/environment.rb*. For both of these, we actually showed the most basic form of configuration. We'll cover some additional useful configuration options here.

Authenticated SMTP Sessions

The following configuration can be used to connect to a mail server which requires user authentication:

```
Ruport::Mailer.add_mailer :default, :host => "mail.adelphia.net",
                                   :address => "test@test.com",
                                   :user => "joe", :password => "secret",
                                   :auth_type => :login
```

As you can see, it's simply a matter of providing a username and password and specifying that the mail server uses login authentication.

User Authentication with a remote database

The following configuration allows you to specify a database username and password and a remote location of your database.

```
Ruport::Query.add_source :default, :user => "joe",
                                   :password => "secret",
                                   :host => "192.168.1.101",
                                   :dsn => "dbi:mysql:storefront"
```

Setting your database driver

If you are working with a database other than MySQL, you can set the :dsn accordingly. For example, working with ODBC would just be:

```
:dsn => "dbi:odbc:storefront"
```

You can consult the RubyDBI documentation to find out what dsn to use for your data source.

Why :default ?

You can see that for both the Mailer and Query configurations, we've called our configurations :default. This is because Ruport will use these sources if others are not specified, but can actually handle multiple sources. For example, you could easily define a test database:

```
Ruport::Query.add_source :test, :user => "root"
                               :dsn => "dbi:mysql:storefront_test"
```

In our report, we could rewrite our query to use this test database.

```
query %q{ select product,quantity,customer_name from sales
          where sale_date = ? order by quantity }, :params => [today],
          :source => :test
```

Mailer works in a similar fashion, and can be configured through the `Report#use_mailer()` method if needed.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

We'll now take a look at Report objects and how they tie in with the rest of Ruport.

Report: Simplified Interface to Ruport

The main purpose of Ruport's Report class is to keep your report definitions from looking like a shell script. It's entirely possible to live without it, and when working within other frameworks, it is even advisable to do so. However, for standalone reports with reasonably basic needs, Report provides a handy baseclass that simplify various common Ruport tasks.

If we look back at our report definition, you'll see it is more-or-less split into two parts.

Report Definition

The first part simply ties a query's result set to a renderer, and provides a helper method for formatting dates

```
class DailySales < Ruport::Report

  renders_with Sales

  def renderable_data

    query %q{ select product,quantity,customer_name from sales
              where sale_date = ? order by quantity }, :params => [today]

  end

  def today
    Date.today.strftime('%Y-%m-%d')
  end

end
```

The `renderable_data` method will pass its return value as the `:data` option to the renderer specified by `renders_with()` when you generate your report.

Report Generation

The second part of our definition is our generation code, which sends an email of PDF and text output when it is run.

```
DailySales.generate do |report|
  report.save_as("sales_report.pdf")
  report.send_to("gregory.t.brown@gmail.com") do |m|
    m.subject = "Sales Report for #{report.today}"
    m.text     = report.to_text
    m.attach "sales_report.pdf"
  end
end
```

One thing you might notice is that when you save files to disk, Report ensures that the proper formatter is called. This means that the following code:

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```
report.save_as("sales_report.pdf")
```

is equivalent to

```
File.open("sales_report.pdf","w") { |f| f << report.to_pdf }
```

This should work with arbitrary format definition, so that `save_as("foo.something")` will be equivalent to:

```
File.open("foo.something","w") { |f| f << report.to_something }
```

Easily Swap Data Sources

By structuring your reports this way, you gain the ability to swap out data sources very easily. Often in my work I will find myself working off of a CSV dump to design my report, and then later hook it up to the actual database. As long as the data is represented in the same way, Ruport doesn't care where it is coming from.

This means that if you had a CSV dump of your sales table, you could rewrite just the `renderable_data` method and not need to touch anything else:

```
class DailySales < Ruport::Report

  renders_with Sales

  def renderable_data
    Table("products.csv")
  end

  def today
    Date.today.strftime('%Y-%m-%d')
  end

end
```

With that in mind, we'll show how to avoid raw SQL by using ActiveRecord instead. The change is surprisingly simple:

Using ActiveRecord as a data source

We can get our model hooked up fairly easily via a simple rake task

```
$ rake build model=sale
```

We won't need to change anything for this report, but you can see the model definition in *data/models/sale.rb* is quite basic:

```
class Sale < ActiveRecord::Base

  acts_as_reportable

end
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

This allows us to use a method called `report_table` to get a `Ruport::Data::Table` back from an ActiveRecord find. Thus, our new, ActiveRecord backed report definition looks like this:

```
require "lib/init"
require "lib/renderers"
require "data/models"

class DailySales < Ruport::Report

  renders_with Sales

  def renderable_data

    Sale.report_table(:all,
      :only => ["product", "quantity", "customer_name", "sale_date"],
      :conditions => ["sale_date = ?", today],
      :order => "quantity")
  end

  def today
    Date.today.strftime('%Y-%m-%d')
  end

end
```

The `Report.generate` code and renderer needn't be changed at all.

Quick Look at the Renderer for this report

You'll notice that our renderers don't look any different when we use them in the context of rope, they're just the same as you'd use them anywhere else.

The only particularly interesting bit about these is that they make use of a helper module to encapsulate some tasks common to the formatters.

```
module Helpers

  def today
    Date.today.strftime("%m/%d/%y")
  end

  def total_sales
    data.sigma("quantity")
  end

  def total_customers
    data.column("customer_name").uniq.length
  end

end
```

This allows us to call these methods in our formatters and fit them into our format as needed. There is nothing especially magic going on here, the Helpers module is just mixed into the formatters at render time.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

Discussion Summary

You've seen here how rope provides a way to partition your code so it can easily be reused without having to write a ton of boilerplate code. The common task of writing a quick report that's capable of being run by a scheduler and delivered to the people that need it should be easy, and rope tries to make sure that it is.

13.3 Comparing Text Reports to Well Formed Data

Problem

Although it is desirable to report against well formed data, it's not always an option. Most systems we deal with day to day make use of relational databases or at least offer standardized export formats, but this isn't true for our legacy systems. Luckily, Ruby is an excellent text processing language, so it can handle most obscure data formats with ease. We'll show here how to do a basic comparison report between a standard CSV and a messy export from a legacy system, making use of Ruport for our data manipulations and formatted output.

Essentially, this report consists of a comparison between our legacy data:

Transaction Report 1/1'6 Through 11/16'6				Page 1
11/16'6				
ETS Checking				
Date	Num	Description	Category	

BALANCE 12/31'5				
1/2'6	38-487	misc adj	Misc	
1/3'6	38-490	John Parlian	Sales	
1/3'6	38-491	Tolland	Sales	
1/3'6	38-492	New Alliance...	Misc	
1/4'6	38-494	James Farall...	Misc	
1/4'6	38-495	Returned Check	Misc	
1/4'6	38-496	Craig Winter...	Dinners	
1/4'6	38-497	York	Dinners	
1/4'6	38-498	York	Misc	
1/4'6	38-499	York	Sales	
1/4'6	38-500	York	Sales	
1/4'6	38-501	Wooster	Sales	
1/5'6	38-502	Tolland	Sales	
1/5'6	38-503	Bankcard Ser...	Dinners	

And a well formed CSV dump from the database:

```
Date,Num,Description,Category
1/2/2006,38487,misc adj,Misc
1/2/2006,38489,John Parlian,Salary
1/3/2006,38490,John Parlian,Sales
1/3/2006,38491,Tolland,Sales
1/3/2006,38492,New Alliance...,Msc
1/4/2006,38493,James Farell...,Salary
1/4/2006,38494,James Farall...,Misc
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```

1/4/2006,38495,Returned Check,Misc
1/4/2006,38497,York,Misc
1/4/2006,38498,York,Misc
1/4/2006,38499,York,Sales
1/4/2006,38500,York,Sales
1/4/2006,38501,Wooster,Sales
1/5/2006,38502,Tolland,Sales

```

Solution

The following report is quite simple, but useful. At work, we use this to catch synchronization issues with a two way bridge between a legacy system and a standard SQL server. It parses both files to create `Table` objects, which can then be compared to see which records are missing from each, and which records are different based on their record number.

```

require "rubygems"
require "ruport"
require "ruport/util"

module Analysis

  module_function

  def process_text_file(filename)
    Table(%w[Date Num Description Category]) do |t|
      File.foreach(filename) do |r|
        next unless r =~ /\A\d+\.\d+\.\d+\s{4}\d/
        row = r.split(/\s\s+/)
        row[0].sub!(/\d+/) { |m| "#{m[1..-1].to_i + 2000}" }
        row[1].delete!("-")
        row[-1].chomp!
        t << row
      end
    end
  end

  def missing_data(options = {})
    from, compare, by = options.values_at(:from, :compared_to, :by)
    keys = compare.column(by) - from.column(by)
    compare.sub_table { |r| keys.include?(r[by]) }
  end

  def unmatched_data(options = {})
    grouping = Grouping(options[:old] + options[:new], :by => options[:by])
    Table(options[:old].column_names + ["file"]) do |t|
      grouping.each do |n, g|
        if g.length == 2 && g[0] != g[1]
          t << g[0].to_hash.merge("file" => "old", "Num" => n)
          t << g[1].to_hash.merge("file" => "new", "Num" => n)
          t << []
        end
      end
    end
  end
end

```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

end

```
table_from_txt = Analysis.process_text_file("checking.txt")
table_from_csv = Table("checking.csv")
```

The final report generation is fairly simple:

```
puts "The following data was missing from the Text file\n"
puts Analysis.missing_data( :from => table_from_txt,
                           :compared_to => table_from_csv,
                           :by => "Num")

puts "\nThe following data was missing from the CSV file\n"
puts Analysis.missing_data( :from => table_from_csv,
                           :compared_to => table_from_txt,
                           :by => "Num" )

puts "\nThe following data did not match in the two files\n"
puts Analysis.unmatched_data( :old => table_from_txt,
                             :new => table_from_csv,
                             :by => "Num" )
```

This outputs a nicely formatted text report that helps show what data is out of sync between the two files:

```
The following data was missing from the Text file
+-----+
| Date | Num | Description | Category |
+-----+
| 1/2/2006 | 38489 | John Parlian | Salary |
| 1/4/2006 | 38493 | James Farell... | Salary |
+-----+
```

```
The following data was missing from the CSV file
+-----+
| Date | Num | Description | Category |
+-----+
| 1/4/2006 | 38496 | Craig Winter... | Dinners |
| 1/5/2006 | 38503 | Bankcard Ser... | Dinners |
+-----+
```

```
The following data did not match in the two files
+-----+
| Date | Num | Description | Category | file |
+-----+
| 1/3/2006 | 38492 | New Alliance... | Misc | old |
| 1/3/2006 | 38492 | New Alliance... | Msc | new |
| 1/4/2006 | 38497 | York | Dinners | old |
| 1/4/2006 | 38497 | York | Misc | new |
+-----+
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

Though the solution for this is basically straightforward, we're playing with some of the most powerful features in Ruby and Ruport. We'll take some time to go through some of the more interesting parts in depth, so that you can use these tricks in your own reports.

Discussion

Parsing Records

The first task we must deal with before doing any comparisons or formatting is parsing the text file input. Our `process_text_file()` method handles this, building up a table of data as it iterates through the lines of the text file.

An individual record in our text file looks like this:

```
1/2'6      38-487      misc adj                      Misc
```

To keep things simple, we tell our processor to skip any rows that don't have this basic format.

```
next unless r =~ /\A\d+\ \/ \d+' \d+ \s{4} \d/
```

The actual check is fairly simple, in english, you can read this as:

Starting at the beginning of the line, match one or more digits followed by a / character, followed by one or more digits, followed by a ' character, followed by four spaces, followed by a digit. This means that the pattern is really only matching the following part of the above record:

```
1/2'6      3
```

As there are only three types of data present in our file, header data, empty lines, and records, this check turns out to be sufficient. Only records will match this pattern. By jumping to the next line if this check fails, we can immediately skip processing headers and blank lines.

If we've matched a record, we then process it to make it into well formed data for comparison. Line by line, it looks like this:

Split by at least two spaces:

```
row = r.split(/\s\s+/)
row #=> ["1/2'6", "38-487", "misc adj", "Misc\n"]
```

Convert the year into a 4 digit year:

```
row[0].sub!(/\d+\/) { |m| "/#{m[1..-1].to_i + 2000}" }
row[0] #=> "1/2/2006"
```

Remove the dashes from the 'Num' column

```
row[1].delete("-")
row[1] #=> "38487"
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

Remove the newline from the 'Category' column.

```
row[-1].chomp!  
row[-1] #=> "Misc"
```

This provides us with a nicely formed record, looking something like this:

```
row #=> ["1/2/2006", "38487", "misc adj", "Misc"]
```

You'll notice that each field is now in a comparable format to the records in our CSV data. We append the record to the table we're building up, and the final result of the `process_text_file()` method is to return the `Table` object with all the processed records aggregated. Because the CSV is well formed, there is no need to manipulate the data, and it is loaded via a simple call:

```
table_from_csv = Table("checking.csv")
```

With both files processed, we can begin our comparison report.

Finding missing data

Figuring out what data is in one file but not the other is simple, because all records have a unique primary key in their 'Num' field. Our `missing_data()` method definition follows:

```
def missing_data(options = {})  
  from, compare, by = options.values_at(:from, :compared_to, :by)  
  keys = compare.column(by) - from.column(by)  
  compare.sub_table { |r| keys.include?(r[by]) }  
end
```

We invoke this method twice, once to show what is missing from the text file, and once more to show what is missing from the CSV file.

In either case, what this does is a simple set difference on the values in the Num field to find out the keys that are present in one file but not the other.

```
keys = compare.column(by) - from.column(by)
```

In our "missing from text file" report, this results in keys being set to the two values which are not present in the text file's 'Num' field, which are:

```
["38949", "38503"]
```

We then create a sub table from the CSV data, including only those records which are missing from the text file:

```
compare.sub_table { |r| keys.include?(r[by]) }
```

The result is our familiar text table output, looking like this:

```
+-----+
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

Date	Num	Description	Category
1/2/2006	38489	John Parlian	Salary
1/4/2006	38493	James Farell...	Salary

The same process is repeated to find the records which exist in the text file but not in the CSV file. As you can see, it is fairly trivial to do this kind of filtering so long as you have a unique key to work with.

The more interesting part of this report is seeing which records exist in both files, but do not match. We use a Grouping object in `unmatched_data()` to simplify things. The very first line joins the two tables together and then groups them:

```
grouping = Grouping(options[:old] + options[:new], :by => options[:by])
```

The real purpose of this grouping is to collect the data in groups by their 'Num' field. This leaves us with two possible group lengths, either 1 or 2. The length 1 groups are ones that exist in only one file, and the length 2 groups are ones which exist in both files. As we're building up our table for our report, we immediately reject the data which isn't in pairs, and then compare the first record to the second record to see if they are equal. If they aren't, we add them to our report.

```
Table(options[:old].column_names + ["file"]) do |t|
  grouping.each do |n,g|
    if g.length == 2 && g[0] != g[1]
      t << g[0].to_hash.merge("file" => "old", "Num" => n)
      t << g[1].to_hash.merge("file" => "new", "Num" => n)
      t << []
    end
  end
end
```

In the above, we are yielded the group name and actual group object for each iteration. Our group name represents the 'Num' value for the records, and our group object contains our tuple of records, with the old values preceding the new values. We simply append these values to the table we are building up, adding an empty row as a spacer for each group. The resulting table looks familiar:

Date	Num	Description	Category	file
1/3/2006	38492	New Alliance...	Misc	old
1/3/2006	38492	New Alliance...	Msc	new
1/4/2006	38497	York	Dinners	old
1/4/2006	38497	York	Misc	new

This allows us to inspect the data very easily and see the rows with unmatched values, and finishes off all the interesting implementation details of the actual report. However, it's worth making a comment about the structure of our report definition, as it deviates a little from the norm.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

Modular Report Definition

Our solution finds itself very near the crossroads between quick one-off scripting and a more structured application. Ruby's object oriented model is very pleasant to work with, but to a minimalist, might seem a little excessive for a problem like this. For this reason, we use a modular design.

By using `module_function`, you can create modules which encapsulate functions that can called directly on the module itself, rather than needing to be mixed into another object. This means that we don't need to define a class to hold our functions in, the module alone will do.

This provides a namespace for our code to live in, which means that we can reuse bits of this code in other scripts without worrying about a method name like `process_text_file` clashing with other definitions.

I find this sort of structure is ideal for scripts that are around this size and complexity. It makes the code still easy to test, something you throw away with one-off scripts, and still prevents you from having to manage state, which is inevitable if you use an object oriented solution.

It's very common for me to reuse bits and peices of my reporting code, so structuring your scripts in this way may make it easier for you to do so without having to think much about design.

Discussion Summary

As you can see, it is quite easy to quickly parse a non-standard text format and then compare it to other data using Ruby's regular expressions and Ruport's table manipulation capabilities. Though this report was somewhat basic, the general approach will be useful for more complex applications as well.

13.4 Custom Printable Documents

Problem

Generating customized printable reports is something most programmers will need to do from time to time. Though Ruport handles the most common cases without any modification, you'll often need to extend its PDF formatter to get the results you need. We'll walk through a common PDF formatter extension to show how to begin customizing your reports.

Solution

The following renderer and formatter takes a CSV and then does a simple grouping operation on it. The result is then displayed on a landscaped page, with a header and information panel in addition to the table of grouped data.

```
require "rubygems"
require "ruport"

class PatientNotes < Ruport::Renderer

  prepare :report
  stage :header, :info_panel, :table
  finalize :report

  class PDF < Ruport::Formatter::PDF

    renders :pdf, :for => PatientNotes
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```

def prepare_report
  options.paper_orientation = :landscape
end

def build_header
  draw_text "FooBar Enterprises", :font_size => 36, :left => 50
  hr
end

def build_info_panel
  info = "daily record for FooBar Enterprises," <<
        "prepared by #{options.preparer}\n\n" <<
        "please process the following orders for shipping.\n" <<
        "if there are any questions, email the sales department"

  rounded_text_box(info) do |o|
    o.radius = 3
    o.width = 300
    o.height = 80
    o.heading = "purchase log for #{date.today.strftime('%m/%d/%y')}"
    o.font_size = 12

    o.x = 450
    o.y = 580
  end
end

def build_table
  move_cursor -50
  render_grouping data, :style => :separated,
    :table_format => { :width => 700 },
    :formatter => pdf_writer
end

def finalize_report
  render_pdf
end

end

end

```

The following code shows how this renderer is invoked:

```

t = Table("products.csv")
puts PatientNotes.render_pdf(:data => Grouping(t, :by => "Name"),
                             :preparer => "Sam Jackson")

```

Our input CSV is fairly simple:

```

Name,Recipient Name,Recipient Location,Order Number,Quantity,Unit
Price,Total
Accordian,Joe Gainsville,"43 Orange Street

```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

Smockton, VT 01010",10123,3,10.25,30.75
 Toy Piano,Mark Union,"100 Telpha Lane
 Silverberg, MD 02020",10124,1,200.00,200.00
 Kite,Al Hooligan,"50 Staley Road
 Sandspring, MI 03030",10125,2,50.00,100.00
 Toy Piano,Joe Gainsville,"43 Orange Street
 Smockton,VT 01010",10126,2,200.00,400.00
 Kite,Ralph Eggert,"109 Salt Rock Road,
 Trist, CA 04040",10127,1,50.00,50.00
 Kite,Allen Spitz,"300 Telian Court
 Apartment 3A
 East Bay, FL 05050",10128,5,50.00,250.00

After grouping and applying the formatting we've specified, we get nice PDF output like this.

Purchase Log for 07/23/2007						
Daily Record for FooBar Enterprises, prepared by Sam Jackson						
Please process the following orders for shipping. If there are any questions, email the Sales Department						
Name	Recipient Name	Recipient Location	Order Number	Quantity	Unit Price	Total
Accordian	Joe Gainsville	43 Orange Street Smockton,VT 01010	10123	3	10.25	30.75
Toy Piano	Mark Union	100 Telpha Lane Silverberg, MD 02020	10124	1	200.00	200.00
	Joe Gainsville	43 Orange Street Smockton,VT 01010	10126	2	200.00	400.00
Kite	Al Hooligan	50 Staley Road Sandspring, MI 03030	10125	2	50.00	100.00
	Ralph Eggert	109 Salt Rock Road, Trist, CA 04040	10127	1	50.00	50.00
	Allen Spitz	300 Telian Court Apartment 3A East Bay, FL 05050	10128	5	50.00	250.00

Although it definitely involves rolling up your sleeves, you can see that it isn't terribly difficult to get really fine grained control over your document rendering.

Discussion

Most customized formatting jobs in Ruport begin with defining a renderer. This class is responsible for defining the process that should be carried out by formatters, and also serves as the interface to rendering your data in whatever formats you wish to support. For this particular application, our definition is quite simple:

```
class PatientNotes < Ruport::Renderer

  prepare :report
  stage :header, :info_panel, :table
  finalize :report

  # ...

end
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

With the above definitions, when told to render a specific format, the following hooks will be called on the formatter, in order:

- * prepare_report
- * build_header
- * build_info_panel
- * build_table
- * finalize_report

Any hooks which are not implemented are simply ignored. This allows the formatters to retain some deal of independence from the renderers.

This is actually necessary because the Eenderers do not need to know about the formatters at all when they are defined. If you take a look at our formatter definition, you'll see the following definition:

```
class PDF < Ruport::Formatter::PDF
  renders :pdf, :for => PatientNotes
  # ...
end
```

It is this callback which allows us to render our results as such:

```
puts PatientNotes.render_pdf(:data => Grouping(t, :by => "Name"),
                             :preparer => "Sam Jackson")
```

This turns out to be a handy feature, as there is no need to follow a specific convention, format names are simply labels. So for example, if we had used:

```
renders :landscaped_pdf, :for => PatientNotes
```

Our call would look like this:

```
puts PatientNotes.render_landscaped_pdf(:data => Grouping(t, :by => "Name"),
                                         :preparer => "Sam Jackson")
```

Stylistically it is best to have the format name be the same as the extension on your outputted files, but there is no requirement to do that.

You may have noticed that we nested our formatter within our renderer definition, e.g.

```
class PatientNotes < Ruport::Renderer
  class PDF < Ruport::Fromatter::PDF
  end
end
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

This is also just a matter of convenience. It allows us to wrap our formatter in a namespace, with the fully qualified name `PatientNotes::PDF` and prevents it from clashing with other formatter extensions. However, there is nothing wrong with defining the formatters outside of your renderers, especially if you wish to use the same formatter for multiple renderer classes.

Now that we've gone through the basic structure of a formatting extension, we can walk through each of the stages of the report and take a look at what's going on.

Setup

The first thing you should notice is that we are working on a subclass of `Ruport::Formatter::PDF`, which allows us to gain access to a number of helper methods to do customized PDF output.

Because this particular report has many columns, we want a landscaped report. We set an option for this in our `prepare_report` hook

```
def prepare_report
  options.paper_orientation = :landscape
end
```

It's important to set this option before calling any methods that would draw on the PDF object. Once you begin drawing, you cannot change the paper orientation. For this reason, our `prepare` hook is the best place to put this instruction to be sure it will be called before anything else.

Company Header

Once we've established the paper orientation, we can begin drawing our report. Starting with the header, we need our company name in large font followed by a horizontal rule:

```
def build_header
  draw_text "FooBar Enterprises", :font_size => 36, :left => 50
  hr
end
```

We're doing a bit of measured text here, describing a specific distance from the left margin in pixels. We could have also specified this in terms of ruler measurements, so long as we convert the values before passing them. This requires us to make use of a `PDF::Writer` helper. For example, if we wanted to place our text two inches from the left margin, we could do something like this:

```
def build_header
  draw_text "FooBar Enterprises", :font_size => 36,
    :left => pdf_writer.in2pts(2)
```

Of course, you don't always need to be so specific about where text is placed on the screen, this was only necessary for our needs. If you want Ruport's PDF formatter to handle flow control for you, the `add_text()` method works fine.

Sometimes you'll have an image logo instead of or in addition to a text company header. For this, take a look at Ruport's `center_image_in_box`, which allows you to use absolute positioning of images as well as handle automatic scaling.

Info panel overlay

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

In our report, we overlay a rounded text box with some additional information. This text box has some dynamic elements to it, but they're mostly trivial. Let's take a quick look at the stage so we can talk about what's going on with it.

```
def build_info_panel
  info = "Daily Record for FooBar Enterprises," <<
        " prepared by #{options.preparer}\n\n" <<
        "Please process the following orders for shipping.\n" <<
        "If there are any questions, email the Sales Department"

  rounded_text_box(info) do |o|
    o.radius = 3
    o.width = 300
    o.height = 80
    o.heading = "Purchase Log for #{Date.today.strftime('%m/%d/%Y')}"
    o.font_size = 12

    o.x = 450
    o.y = 580
  end
end
```

The `rounded_text_box()` method is fairly straightforward. We provide the body text info as an argument and then set the formatting details and our text box header via the block. All of the above fields (with exception of heading) are required to create a rounded text box, so be sure to include them in your reports. If you have to draw several similar boxes, you can of course create a helper method that simplifies things.

The more interesting part of this code is that we make use of formatting options, allowing us to pass in extra data at rendering time to populate the dynamic parts of our report.

If you look back at our code that actually renders the report, you'll see we pass an option as `:preparer => "Sam Jackson"`. In the code above, this is represented by `options.preparer`. Ruport's formatting option system is very simple, and this is one of three ways to use it.

Another way you can invoke this is to use the block form of rendering to access the options object directly:

```
PatientNotes.render_pdf do |r|
  r.data = Grouping(t, :by => "Name"),
  r.options.preparer = "Sam Jackson"
end
```

The options object is shared by the renderer and whatever formatter it invokes, so you can actually make use of these in both the formatter and renderer as needed. They can also be accessed like an indifferent hash, e.g. `options["preparer"]`, which can come in handy for highly dynamic reports.

The last way to set an option is to use the block form with a declaration of an option in the renderer, e.g.

```
class PatientNotes < Ruport::Renderer
  option :preparer
end
```

This allows you to simplify our earlier example to look like this:

```
PatientNotes.render_pdf do |r|
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```

    r.data = Grouping(t, :by => "Name"),
    r.preparer = "Sam Jackson"
end

```

Finally, if you want to be sure an error is raised when this option is not set, you can use `required_option()` instead of `option()` in your renderer definition. This will effect both the hash form of option handling as well as the block form, and will make it so that an invocation like this:

```

PatientNotes.render_pdf(:data => Grouping(t, :by => "name"))

```

will raise a `RequiredOptionNotSet` error. Whether you choose to use this feature or not is entirely up to you, and will probably depend on your needs.

Table rendering

The data object we're passing in is a `Ruport::Data::Grouping`, and because this already has a built in formatter that will work for our needs, we don't need to reinvent the wheel. We can make use of a simple renderer helper to call that formatter. The only caveats are that we need to pass our `PDF::Writer` object into it to make sure that renders on our object instead of creating a new PDF, and we need to manually position it on our document. The following code does exactly that:

```

def build_table
  move_cursor -50
  render_grouping data, :style => :separated,
    :table_format => { :width => 700 },
    :formatter => pdf_writer
end

```

The `move_cursor` call lets us move our drawing cursor down the page 50 pixels, leaving sufficient room between the start of the table and the end of the header content. When we call `render_grouping`, we simply pass it our data object, and then tell it to use the separated grouping style. We also set some `PDF::Writer` objects via `:table_format`. Any attributes you specify via this method are used to set accessors on the underlying `PDF::SimpleTable` object, which means that you have full control over the table generation. For more details on this, check the [Ruport API Documentation for Ruport::Formatter::PDF](#) and possibly the `PDF::Writer` documentation.

Finishing up

Many of Ruport's formatters build up their output as they go along. Because of the way PDF documents are structured and `PDF::Writer` is implemented, this is not the case with the PDF formatter. For this reason, we must explicitly tell the PDF object to render itself when all the stages have completed. We do this via our `finalize` hook:

```

def finalize_report
  render_pdf
end

```

This simply calls `output << pdf_writer.render`, but will help ensure that your code will stay working even if the underlying PDF implementation changes. Typically, when you can use a Ruport helper instead of a `PDF::Writer` method, its a better idea, as those calls are officially supported and maintained.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

However, you may end up having to reuse old PDF::Writer code or squeeze more formatting power out of Ruport than what's needed for the typical report. If that's the case, you'll want to take a look at our `pdf_writer_proxy` program, which forwards all requests not directly understood by Ruport's formatter to the underlying PDF::Writer object. At the very least, this will save you some typing, and more likely, will make it easier to get the best of both worlds between high and low level functionality.

Discussion Summary

We've walked through a non-trivial PDF report that hits some of the common elements you will need: Custom headers, text boxes, and tabular output. Though it can get to be a little low level, you can more-or-less accomplish anything you might need with PDF::Writer and a little help from Ruport. There are copious PDF examples for Ruport distributed with the source, but you may be able to get by with what you've seen here for the most common cases. One thing worth mentioning is that PDF::Writer has some notable performance issues, and you may wish to read the recipe on report performance tweaking if you plan to deal with large datasets

13.5 Reporting against large data sets

Problem

It's no secret that Ruby is a slow language. Combine this with the common task of generating reports based on anywhere from 10 thousand to 10 million records, and it seems as if you're asking for trouble. However, in many cases, it is still possible to squeeze out enough performance to make it worthwhile to stick to Ruby. The following example shows how to build a row based processor to generate Text, HTML, and PDF reports for high data volumes without eating up endless resources.

Solution

Although Ruport is most at home with processing and formatting tables, it isn't impossible to work in terms of rows. In fact, Ruport has a built in row rendering system that can be easily customized as needed.

For this task, we have trivial data, but it spans over 20000 lines. Here is a small sample of what it looks like:

```
date,chart,amount
2/20/2007,453175.2S,$325.00
2/20/2007,453175.2S,$300.00
2/20/2007,453175.2S,$250.00
11/17/2006,233089,$58.00
11/17/2006,233089,$58.00
11/17/2006,233089,$-
11/17/2006,233089,$58.00
11/17/2006,233089,$58.00
1/22/2007,233089,$84.00
1/22/2007,233089,$84.00
1/22/2007,233089,$84.00
```

The following code uses Ruport, PDF::Writer, and FasterCSV to process this data and do the necessary format conversions.

```
require "rubygems"
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```

require "ruport"
require "pdf/writer"
require 'fastercsv'

class MyPDF < Ruport::Formatter::PDF

  renders :pdf, :for => Ruport::Renderer::Row

  def build_row

    pdf_writer.start_new_page if cursor < 50

    pad(5) do
      ypos = cursor
      draw_text data[0], :y => ypos
      draw_text data[1], :y => ypos, :left => 150
      draw_text data[2], :y => ypos, :left => 250
    end

    horizontal_rule
  end

end

pdf = PDF::Writer.new
renderer = Ruport::Renderer::Row
widths = [10,12,10]

File.open("out.txt", "w") do |text|
  File.open("out.html", "w") do |html|
    html << "<table>"
    FasterCSV.foreach("big.csv") do |r|
      next unless r[2] =~ /\d|(amount)/

      html << renderer.render(:html, :data => r)

      text << renderer.render(:text, :data => r,
        :max_col_width => widths, :ignore_table_width => true)

      renderer.render(:pdf, :data => r, :formatter => pdf)
    end
    html << "</table>"
  end
end

pdf.save_as("out.pdf")

```

If you noticed, it seems like we've unraveled the system a bit, creating our own custom PDF renderer, using ordinal values instead of column based access, and even resorting to using some of the methods of Ruport's dependencies directly. The reason for this can be summed up in two words: Speed and Memory.

On my machines, this takes about 1 minute to run. Better than 75% of that time is spent generating the PDF, which is over 500 pages long. Since this is all done in pure Ruby, speed usually comes at the cost of

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

elegance. We'll now take a closer look at the different techniques used to get a feel for how to attack similar problems when you encounter them.

Discussion

Why use a row processor?

When dealing with smaller data sets, it's often convenient to think and work in terms of tables rather than rows. However, this becomes less and less feasible the larger your data set gets. This is a problem where with every new row, your cost increases in both time and space. By cutting the tables out of the equation, we can process, manipulate, and format each row as we receive it, and then let garbage collection pick up for the discarded ones as needed.

That's why in the code below, you'll notice that we only need to store one row at a time, instead of having the whole result set in memory.

```
FasterCSV.foreach("big.csv") do |r|
  next unless r[2] =~ /\d|(amount)/

  text << renderer.render(:text, :data => r,
    :max_col_width => widths, :ignore_table_width => true)

  html << renderer.render(:html, :data => r)

  renderer.render(:pdf, :data => r, :formatter => pdf)
end
```

The drawbacks of row based processing is that it makes for uglier code. If we weren't trying to be conservative about resources, the above could be written like this:

```
table = Table("big.csv")
table.reduce { |r| r.amount =~ /\d/ }

text << table.to_text
html << table.to_html
pdf << table.to_pdf
```

However, nothing above is particularly efficient, and falls down and dies under high data volumes. You can see in the latter case that column names are automatically detected, but that in the row processor, we need to check for them explicitly:

```
next unless r[2] =~ /\d|(amount)/
```

Though it's ugly, this lets us filter a specific column without applying the filter to the header row. In cases where simple pattern matching like this won't work, you may need to take a different approach.

Rendering data by row in Ruport

Because our data isn't being rendered as a table, but rather as a series of rows, we needed to roll up our sleeves a bit here. Ruport provides a base row renderer for HTML, Text, and CSV, but not PDF. It also requires a little tweaking to make use of the Text formatter.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

HTML

For HTML, you'll notice that we don't need to provide any special directives:

```
html << renderer.render(:html, :data => r)
```

HTML turns out to be an excellent format for streaming, because we don't need to worry about column widths, or page breaks, or anything like that for the basic cases. We really just need to generate something like this for each row:

```
<tr><td>1/22/2007</td><td>233089</td><td>$84.00</td></tr>
```

This is exactly what the above line of code does, with some whitespace considerations for easier hand editing. It is implemented very efficiently, and if we were only rendering HTML in this report, it would take less than 5 seconds to generate the HTML from our CSV.

Text

You'll notice that the text formatting isn't nearly as straightforward. We need to make use of fixed column widths and turn off table width detection.

```
text << renderer.render(:text, :data => r,  
  :max_col_width => widths, :ignore_table_width => true)
```

The problem with doing row based text output is that in order to format the rows properly, we need to set fixed widths for the columns. For example, if we did not specify column widths, we'd likely end up with something like this:

```
| a | this is a long field |  
| bc | this is short |  
| d | this is another long field |
```

By specifying fixed widths, we can get better looking output, something more like this:

```
| a | this is a long field      |  
| bc | this is short          |  
| d | this is another long field |
```

In the code above, `:ignore_table_width => true` simply tells Ruport not to try to truncate the table to fit the console, making it suitable for output to file.

As another minor performance note, the text generation code might have read a little more clearly if we embedded the widths directly:

```
text << renderer.render(:text, :data => r,  
  :max_col_width => [10,12,10], :ignore_table_width => true)
```

However, the problem here is that Ruby would create a new array each time a new record was rendered, and since these values do not change from row to row, we can speed things up a little by defining these values before entering the loop.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

PDF

Ruport is capable of generating very nice tables using `PDF::SimpleTable`. However, this tool is notoriously slow and becomes painful to work with at a relatively small amount of pages. The reason for this is similar to why tabular output of text is tricky to do: Every cell needs to be inspected in order to figure out the right column widths, and even worse for PDFs, tables need to properly span across pages, re-printing the column names when needed.

Our approach here is to create a very simple, fixed width PDF output that properly spans pages and is easy to read. The following chunk of code does exactly that.

```
class MyPDF < Ruport::Formatter::PDF

  renders :pdf, :for => Ruport::Renderer::Row

  def build_row

    pdf_writer.start_new_page if cursor < 50

    pad(5) do
      ypos = cursor
      draw_text "Date: #{data[0]}", :y => ypos
      draw_text "Chart: #{data[1]}", :y => ypos, :left => 150
      draw_text "Amount: #{data[2]}", :y => ypos, :left => 250
    end

    horizontal_rule
  end

end
```

Using the above formatter, we end up with simple output which looks like this:

date	chart	amount
2/16/2007	170831	\$210.00
2/16/2007	170831	\$270.00
12/29/2006	450316	\$330.00
12/29/2006	450316	\$330.00
12/2/2006	439626S	\$58.00
12/2/2006	439626S	\$58.00
12/18/2006	288087S	\$150.00
12/18/2006	288087S	\$150.00

You may have more complex needs for your reports, and you can certainly tweak the formatter a whole lot more before it becomes too costly.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

You'll notice that we manually check to see if we need to advance the page before rendering a new row. This makes sure that we prevent records from spanning multiple pages, and provides us a buffer zone between the end of the page and our last record on the page.

Another thing to keep in mind is that PDF is not a format that can easily be streamed, and even if it were possible, the PDF::Writer API does not provide us a way to do it. This is why you'll notice that we need to instantiate the PDF::Writer object outside of the main processing loop:

```
pdf = PDF::Writer.new
```

We then pass this to our row renderer so that it uses this object instead of creating a new PDF::Writer object with a single row each time:

```
renderer.render(:pdf, :data => r, :formatter => pdf)
```

Finally, in the end, we can save the full PDF once it has been generated:

```
pdf.save_as("out.pdf")
```

In practice, I've found this approach to scale fairly well, so long as you stick to primitive drawing methods, text rendering, and manual page advancement. It is when you need to either inspect the entire data source before producing a report, or when you need to do complex checks to determine whether or not your given data will fit properly on a page or in a column that things start to slow down greatly.

Discussion Summary

The general patterns shown here should help when you need to squeeze a little more speed out of a report or conserve some memory. The CSV file in this example can be replaced with any streaming data source, including result sets from database or another source such as a file processed with StringScanner.

The key things to remember are true for most applications, but it's worth mentioning them succinctly:

- Avoid loading full data structures in memory if they can be streamed
- Use lightweight datastructures if possible
- Avoid repeatedly creating new objects with the same values in loops
- Use fixed column widths if possible in formatted output
- Disable as much auto-detection as possible and tune for your specific problem
- Use lower level tools when necessary to avoid overhead

With this in mind, if you're generating batch reports at night, or have processes that can run for a few minutes without causing problems, it's entirely feasible to use Ruby to process large data sets. If you're not in that boat, you'll need to consider ways to parallelize your task or think about writing some C extensions. However, for a lot of common cases, this general approach should do the trick.

13.6 Summary

From the five recipes here, you have hopefully gained a solid base in building reporting applications with Ruby and Rupert, and should be able to adapt the examples as needed to solve your actual problems. Rupert is ultimately more about helping you define your reporting process and helping with the low level bits than it is solving any particular reporting problem, the general approach here will be sufficient to get you well on your way with your work.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

A Installing Ruby

Installing Ruby is a fairly trivial affair, however, there are a few things worth paying attention to. You will want to use Gems extensively, and not all installations include RubyGems by default. You may also want to use Gems that take advantage of C extensions, for example MySQL drivers and the Mongrel Web server, which require that you have the proper development libraries and tools. In this appendix we provide you with the simplest steps for installing a fully functional and up to date version of Ruby 1.8 on Windows, OS/X and Linux.

Once you get Ruby up and running, head over to the last section where we provide useful tips for improving IRB (the Interactive Ruby Interpreter) and accessing documentation for all Gems installed on your machine.

A.1 Windows

If you're using Windows, you have three options to choose from. You can get the latest version of Ruby directly from the ruby-lang.org Web site, with downloads for either Ruby 1.8 or 1.9. When you download the Ruby interpreter it includes command line tools like RDoc, IRB and RI, but does not include RubyGems or many of the other libraries you need to get started.

In our experience, hunting down and installing libraries like RubyGems, FCGI or OpenSSL is not a fun experience, and we've got better things to do. A better alternative is to use the *One-Click Ruby Installer*, available from rubyinstaller.rubyforge.org. It includes many of the common libraries you need to get started, libraries for accessing various Windows APIs, a good programmer's text editor (Scite) and a PDF copy of the book *Programming Ruby*, 1st Edition.

If you're interested in building Rails applications, consider InstantRails, available from instantrails.rubyforge.org. InstantRails includes all the same libraries as One-Click Installer, and also adds recent version of Rails, Mongrel, MySQL and Apache. It's the quickest way to install a working environment for developing and deploying Web applications.

Before you get started, we recommend upgrading to the latest version of RubyGems, which offers important performance and usability improvements:

```
$ gem update -system
```

Now you're all set and ready to go.

A.2 OS/X

OS/X 10.5 (Leopard) comes with Ruby 1.8.6 pre-installed and also includes Rails, Mongrel, Capistrano and a few other Gems.

Before you get started, we recommend upgrading to the latest version of RubyGems, which offers important performance and usability improvements:

```
$ sudo gem update --system
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

The different release schedules mean that Leopard does not include the most recent versions of Rails and Mongrel, not a problem since those are safe to upgrade using `sudo gem update`.

OS/X 10.4 (Tiger) includes Ruby 1.8.2, and many of the libraries you will want to use no longer support 1.8.2, for example, Rails requires 1.8.4 or later. The easiest way to upgrade to a more recent version of Ruby is using the *Ruby One-Click Installer for OS/X* available from rubyosx.rubyforge.org.

Alternatively, you can install Ruby using either MacPorts or Fink. For example, to install Ruby with RubyGems using MacPorts:

```
$ sudo port install ruby rb-rubygems
```

Ruby support on OS/X is excellent, in spite of that, we did run into a couple of gotchas. Occasionally, Ruby Gems that use C extensions will fail to recognize libraries installed outside the main directories, for example, libraries installed using MacPorts.

These are easy to fix by passing specific compile/build options to `gem install`, for example:

```
$ sudo gem install oniguruma -- --with-opt-lib=/opt/local/lib
```

MySQL adds another twist. By default C extensions are built as universal binaries, but MySQL ships with per-architecture binaries. You can force Ruby to build extensions for a particular architecture by setting the ARCHFLAGS environment variable. To install MySQL on the Intel architecture:

```
$ sudo -s
$ ARCHFLAGS="-arch i386" gem install mysql -- --with-mysql-dir=/usr/local/mysql
```

On the PowerPC, replace “-arch i386” with “-arch ppc”.

A.3 Linux

Installing Linux on RedHat/Fedora couldn't be easier. Just go into the terminal and use the package manager:

```
$ sudo yum install ruby ruby-devel rubygems gcc
```

The `ruby` package provides the generic command line utilities, while `ruby-devel` provides header files, and `gcc` the compiler, both of which are necessary for installing Gems that use C extensions, such as Mongrel and MySQL.

Most likely this will install an older version of RubyGems. We recommend upgrading to the latest version of RubyGems, which offers important performance and usability improvements:

```
$ sudo gem update --system
```

Installing Linux on Ubuntu/Debian takes a bit more effort. For starters, the `ruby` package will only install the Ruby interpreter but none of the command line tools (RDoc, IRB, etc). Use the meta-package `ruby-full` to install all the relevant command line tools.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

To build Gems that use C extensions you'll need both **ruby1.8-dev** and if you don't already have them, the various build tools like GCC and Make, provided by the **build-essentials** package:

```
$ sudo apt-get install ruby-full ruby1.8-dev libopenssl-ruby build-essential
```

Do not install the **rubygems** package. This package is setup up in such a way that it allows using installed Gems, but the only way to install new Gems is using **apt-get**. Unfortunately, many of the Gems you will want to use, and several Gems we cover right here in the book, are not available as Debian packages.

Instead, we recommend that you install the latest version of RubyGems by downloading it from the RubyForge project:

```
$ curl -OL http://rubyforge.org/frs/download.php/29548/rubygems-1.1.0.tgz
$ tar xzf rubygems-1.0.1.tgz
$ cd rubygems-1.0.1
$ sudo ruby setup.rb
$ sudo ln -s /usr/bin/gem1.8 /usr/bin/gem
```

Remember that certain Gems use C extensions, which in turn require header files to compile. For example, to install the Ruby MySQL Gem it is not enough to have MySQL installed on your machine, you must also install the MySQL developer package:

```
$ sudo apt-get install libmysqlclient15-dev
$ sudo gem install mysql
```

Next, we'll cover some tips for setting up your working environment to use Ruby Gems and IRB effectively.

A.4 More Tips

Now that you have Ruby installed, it's time to customize your environment. We're going to cover three simple setups that we find indispensable in every environment. They'll make it easier to run Ruby scripts, enhance the Interactive Ruby Interpreter and let you access documentation for the various Gems you install.

Requiring RubyGems with RUBYOPT

One thing we find annoying about Ruby is that it still treats RubyGems as an optional extension that you have to install separately. And since it's optional, you also have to **require 'rubygems'** in any program that uses Gems.

You can get around this by setting the RUBYOPT environment variable to the value "rubygems". On Windows you can set this environment variable using the Control Panel, the One-Click Ruby Installer will automatically do that for you.

On OS/X, Linux and other flavors of UNIX, you can add this line to your profile:

```
export RUBYOPT=rubygems
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

Improving IRB with Wirble

If like us you work from the command line and use IRB (the Interactive Ruby Interpreter) or the Rails console, which itself uses IRB, we highly recommend installing Wirble. Wirble adds tab-completion, history, a built-in `ri` command and colored output.

Once you install Wirble (`gem install wirble`), create a file called `.irbrc` in your home directory and add the following lines to it:

```
require 'rubygems'
require 'wirble'
# start wirble (with color)
Wirble.init
Wirble.colorize
```

Don't add the last line if you don't like colors in your console, or if you are running from the Windows command line, which doesn't support ANSI colors.

Once you have Wirble installed you can use tab-completion on Ruby classes, modules and methods. For example, begin by typing `Obj` and click the Tab key once to expand it to `Object`. Continue by typing a period (`.`) and click the Tab key again to reveal all the methods on the Object class.

Not sure what a class or method does? You can get help from within IRB by running the `ri` method with a class, module or method name. For example:

```
>> ri "MatchData"
>> ri "Object.dup"
>> ri "open"
```

You can also use the Up/Down arrow keys to go back in history.

Using the Gem Server

As intuitive as Ruby is, you won't get far without knowing the APIs and that means looking up the documentation. Ruby comes with two tools that let you access documentation using the command line or your Web browser.

You can use the command-line tool `RI` to request information about any class, module or method, from both the core library and any installed Gem. For example:

```
$ ri Enumerable
$ ri lambda
$ ri String.to_i
```

You can use the Gem server to access documentation of all installed Gems from your Web browser. This method is much easier for navigating through the documentation. In addition to class and method documentation it includes the README file, which often contains valuable information about the Gem and its usage, alongside links to the official Web site, list of dependencies and any command-line scripts.

Start by running the Gem server from the command line:

```
$ gem server
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

And point your browser to <http://localhost:8808>. You now have all your Gem's documentations available at your fingertips.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

Licensed to Keith Steward <paypal1@keithsteward.com>

B JRuby

JRuby is an implementation of the Ruby language that runs on the Java Virtual Machine. The main benefits of using JRuby are the ability to mix Ruby and Java code, the performance benefits gained from running on the JVM, and the ability to deploy Web applications to Java servers like Tomcat and J2EE. This appendix provides an overview that will help you get started exploring the many facets of JRuby.

B.1 Installing and Using JRuby

The steps required to install JRuby are the same on all operating systems. First, make sure you have Java installed and that the `java` command is accessible from the path. Next, download the most recent version of JRuby available from dist.codehaus.org/jruby/, expand the archive to a directory of your choice, and set the path to point to the `bin` directory that contains the various JRuby executables (`jruby`, `jirb`, etc). Verify that JRuby is installed correctly by running `jruby --version` from the command line.

For example, on Linux you could download and install JRuby 1.1 like this:

```
$ curl -OL http://dist.codehaus.org/jruby/jruby-bin-1.1.tar.gz
$ tar -xz < jruby-bin-1.1.tar.gz
$ mv jruby-1.1 /opt/jruby
$ echo "export PATH=$PATH:/opt/jruby/bin" >> .profile
```

To run Ruby programs using JRuby, use the `jruby` command, for example:

```
$ jruby myprog.rb
```

You can also use `jirb`, the JRuby Interactive Interpreter, then same way you would use `irb` (see Appendix A for instructions on how to install Wirble).

JRuby includes a recent version of RubyGems, and you can start installing and managing Gems using the `gem` command. Make sure to read the next section to avoid conflicts between the Ruby and JRuby Gem repositories.

If you need specific JVM settings you can pass them along to JRuby using command line options with the prefix `-J`, or by setting the environment variable `JAVA_OPTS` (`VM_OPTS` on Windows). For example:

```
$ export JAVA_OPTS=-Xmx512m
$ jruby -J-server myapp.rb
```

Next, we're going to tell you how to use JRuby and Ruby side by side on the same machine. Whether you installed both, or already have Ruby on your machine (some operating systems come with Ruby pre-installed), you'll want to read the next section to avoid common setup issues.

B.2 JRuby and Ruby Side By Side

Ruby and JRuby maintain separate Gem repositories and any Gem you install for one will not be available for the other. While most Gems are identical across all platforms, Gems that use C extensions install different code depending on the target platform. On Windows machines (`mswin32`) they will use DLLs and on most

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

other platforms compile shared libraries during installation. Gems that use C extensions will not install on JRuby. Fortunately, many popular Gems like Mongrel, Hpricot and RMagick were ported over to JRuby and install using Java libraries instead of C extensions.

RubyGems handles this for you by installing the right Gem for the target platform, so `gem install mongrel` will work the same way for all Ruby platforms and for JRuby. There are a few cases where you will be using different Gems, for example, *mysql* for Ruby but *jdbm-mysql* for JRuby.

Many Gems install command line scripts, for example Rake provides `rake` while RSpec uses `spec`. Installing the Gem on both platforms will result in two command line scripts, in two different locations, both available from the path. This may be a source of confusion at first, but is easy to work around using the `-S` command line argument, as the following example illustrates:

```
$ ruby -S spec specs/*
$ jruby -S gem install rspec
$ jruby -S spec spec/*
```

This is also the suggested way to install Gems when using Ruby and JRuby side by side. If Ruby's `gem` command shows up first on the path, you can run it directly to install Gems in the Ruby repository, and use `jruby -S gem` to install Gems in the JRuby repository, and vice versa.

When working on Rails applications, many of the scripts you will be using (for example, `script/server`, `script/plugin`) are used from the current directory not the path. In this case you cannot use the `-S` option nor do you need to. Simply run the script with the right interpreter, for example:

```
$ ruby script/server
$ jruby script/server
```

If you're writing Ruby code that behaves differently on each platform, you can detect when it runs in JRuby by looking at the value of the `RUBY_PLATFORM` constant. This constant holds a string that depends on the target platform, for example, "i386-mswin32" or "universal-darwin9.0". Since Java code behaves the same way on all operating systems, JRuby always sets this constant to "java". You can still determine the underlying operating system by checking the value of `Config::CONFIG['host_os']`.

Next, let's see how we can use JRuby to build Ruby applications that use Java libraries and tightly integrate with Java code.

B.3 Mixing Ruby And Java

As you can imagine, JRuby is all about easy Java/Ruby integration. You can access any Java class or interface available in the class path through the `Java` module. For example:

```
>> foo = Java.java.lang.String.new('foo')
=> #<Java::JavaLang::String:0xe704bd @java_object=foo>

>> bar = Java.java.util.HashMap.new
=> #<Java::JavaUtil::HashMap:0xcd022c @java_object={}>
```

Calling Java methods is just as easy:

```
>> foo.toString
=> "foo"
>> foo.to_string
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```
=> "foo"
>> foo.to_s
=> "foo"
```

As you can see from this example, you can call methods using either Java or Ruby naming conventions, either `toString` or `to_string`. Since these are also Ruby objects, you can call their Ruby methods as well, in this case `to_s`.

The conversion works both way, for example, passing Ruby objects to Java:

```
>> array = Java.java.util.ArrayList.new([1, "foo", Object.new])
=> #<Java::JavaUtil::ArrayList:0xc7c7bc @java_object=[1, foo,
#<Object:0xa0a9a>]>

>> array.to_s
=> "[1, foo, #<Object:0xa0a9a>]"
```

Primitive types like strings and integers automatically convert to the right Java type. Since Ruby objects are also Java objects, you can define Ruby classes that extend Java classes. Java interfaces are exposed as modules, so to implement an interface simply `include` it:

```
class MyIterator
  include Java.java.util.Iterator
  def hasNext
    false
  end
end
```

If you need to cast arrays, use the `to_java` method, as this example illustrates:

```
>> ['foo', 'bar'].to_java
=> #<#<Class:01x96f94>:0x7a00b @java_object=[Ljava.lang.Object;@9c5304>

>> ['foo', 'bar'].to_java(Java.java.lang.String)
=> #<#<Class:01xf3941>:0x8fc43e @java_object=[Ljava.lang.String;@9cce04>
```

As with Java, you can use `import` to save yourself typing the full package name:

```
>> import Java.java.util.Hashtable
=> Java::JavaUtil::Hashtable
>> Hashtable.new
=> #<Java::JavaUtil::Hashtable:0x3584f9 @java_object={}>
```

To access Java libraries, add them to the classpath using the `-J-cp` command line argument, by setting the CLASSPATH environment variable, or simply by requiring them:

```
>> require '/usr/share/ant/lib/ant.jar'
>> Java.org.apache.tools.ant.Project
=> Java::OrgApacheToolsAnt::Project
```

The Java system properties are always available from ENV_JAVA, for example:

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```
>> ENV_JAVA['java.runtime.version']  
=> "1.5.0_13"
```

We showed you how to use Java code from Ruby, next we'll show you how to mix Ruby code into Java applications using the scripting support provided by Java 6.

B.4 Scripting With Ruby

With JRuby you can also run Ruby scripts and use Ruby libraries inside your Java applications. You will need Java 6, which adds scripting support, the JRuby libraries in the classpath, and `javax.script.EngineManager` to create a `ScriptEngine` that can evaluate Ruby code.

This simple example shows how you can load a Ruby file and call one of its methods:

```
ScriptEngine jruby = new ScriptEngineManager().getEngineByName("jruby");  
jruby.eval(new FileReader("myscript.rb"));  
String hello = (String) jruby.eval("hello_world");  
System.out.println("Ruby says " + hello);
```

The last JRuby feature we're going to cover is the ability to package Ruby applications as WAR files and deploy them directly into your Java Web server.

B.5 Deploying Web Applications

You can use JRuby to deploy Web applications in much the same way we cover in Appendix C, but you can also use it to package your application as a WAR file and deploy it to any Java Web server like Tomcat or J2EE.

Start by installing the Warbler Gem (`jruby -S gem install warbler`) and head over to the root of your Rails project to package it as a WAR:

```
$ jruby -S warble war
```

Warble creates a WAR file that places the core of your Rails application in the WEB-INF directory, including Rails, its plugins and any of your Gem dependencies. Public files go in the root of the WAR, and you can use additional Java libraries that are copied over from the lib directory to WEB-INF/lib. The result is a self-contained, ready to run, application.

If you need to customize the WAR file to your specific needs, start by creating a new configuration using the `warble config` command and edit the file `config/warble.rb` before running `warble war` again.

C Deploying Web Apps

As you can expect from Ruby, a language that connects with so many Web developers, there is no shortage of options for deploying Web applications. So which one is right for you? We're going to start by briefly reviewing the available options, before narrowing it down to one particular architecture that works best across the board, from local use during development and all the way up to a server farm. We'll show you how to deploy for this architecture using Apache 2 and Nginx as the front-end Web servers, Thin and Mongrel as the back-end application servers.

C.1 A Brief Overview of Deployment Options

The original model for deploying applications behind Web servers was CGI. When using CGI, the Web server starts a new process to handle each incoming request. It may be good enough for simple and oft-used scripts, but if you're using a Web framework or opening database connections, the cost of setting these up for each request will quickly bring your server to its knees. Since most Web applications fall into the later category, we'll turn our attention to better performing options.

To work around this limitation, modern Web servers start the application once and keep that process alive, dispatching incoming requests as they come along. One approach that emerged early on consists of the Web server and application running in separate processes, using protocols like FastCGI or SCGI to connect the two. A fair number of Web servers support FastCGI, SCGI or both, either natively or through add-ons, and on the Ruby side you'll find both the `ruby-fcgi` and `scgi` gems. In spite of availability, we do not recommend either option. As it stands, FastCGI and SCGI fell behind and are harder to setup and administer, and offer lackluster performance compared to the alternatives.

The second approach involves a Web server that can run Ruby code in the same process. Most Web servers support this model through modules, plugins or components, although with limited availability: the choice of language dictates the choice of Web servers able to run it, and few are able to run Ruby code in the same process. As we'll see in a minute, this is not necessarily a problem.

Apache uses modules for running applications in the same process, the most popular being `mod_php`. You won't hear much about `mod_ruby`, its development stagnated, in no small part due to its inability to deal with Rails applications. Java Web servers use components, packaged and deployed as WAR files. These must be written in Java and use the Servlet API, but fortunately do support Rails applications using JRuby to run Ruby applications on the JVM, and Warble to package Rails applications as WAR files. We show you how to do that in Appendix B.

There are three Web servers designed specifically for running Ruby applications. Webrick is a pure-ruby implementation bundled as part of the core Ruby library. When you're running Rails in development mode, you'll notice that it uses Webrick by default. Webrick's best feature is being available everywhere Ruby is, but don't expect much in terms of performance or scalability.

Mongrel is a lightweight Web server that incorporates a native library – C code on most platforms, Java code when running on JRuby – to handle the CPU intensive portion of HTTP processing, offering simple setup and configuration with real-world performance. Thin is another lightweight Web server that users the Mongrel HTTP processing library in combination with Event Machine, a high performance I/O network library, offering better throughput and scalability.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

Mongrel and Thin are both viable options, and while Thin has the edge on performance and scalability, Mongrel has been around for longer, has better tooling support and more extensive documentation. You can understand how difficult it was for us to choose which one server to cover. Fortunately, they're similar enough in principles and basic usage, which we decided to base our examples on Thin, and highlight the differences in a sidenote.

Although Thin and Mongrel are excellent choices for handling the dynamic portion of the application, and support a variety of frameworks (Rails, Merb, Capmingle, to name but a few), they do not offer the same capabilities you would expect from a full-fledged Web server. Neither one is a Web server you would expose directly to the Internet. Rather, we're going to delegate that task to a more capable front-end Web server, and configure it to reverse proxy into our Thin/Mongrel application servers. We'll explain the basics of this architecture in the next section.

C.2 Reverse Proxying

Proxy servers are commonly used as out-bound gateways; they handle traffic emanating from clients on the local network, directed at servers on the Internet. Reverse proxy servers act as gateways for in-bound traffic, responding to requests coming from clients on the Internet and dispatching them to various applications deployed on the local network.

Reverse proxies are compelling for several reasons. To clients they look like a single Web server, and provide a central location for handling encryption, access control, logging, virtual hosting, URL rewriting, static content caching, throttling, load balancing, and everything else a Web server is tasked with. That leaves the back-end Web servers to deal exclusively with the business logic, simplifying management and configuration when you have many different applications deployed throughout the network. The two communicate with each other using the HTTP protocol.

One obvious benefit of this architecture is the ease of scaling, from a single instance used during development all the way to a large-scale server farm. You can start small, deploying the front-end Web server and a handful of application servers on the same machine. Since the intensive portions of the workload are handled by these application servers, scaling is a matter of adding more machines and distributing the application servers across them. A single front-end Web server can handle massive traffic before it reaches its scalability limits. Beyond that, you can start looking at load-balancing proxy servers like Varnish, Pound or PenBalance, or even go down the route of dedicated hardware appliances.

Another benefit is the variety of options available, and the ability to mix and match them to create a best-of-breed configuration. Standardizing on the HTTP protocol allows you to pick from any number of front-end Web servers and load balancers, and just as easily mix in different back-end Web applications, from the simplest ones all the way to mainframes. Because we're exercising management and control through the front-end Web server, we don't have to standardize on a single provider for our Web applications, and easily mix languages and platforms, running Ruby side by side with PHP, Python, J2EE and .Net.

The last benefit is application isolation. Since all back-end applications run independent of each other – the shared nothing architecture – we can add new applications without impacting existing ones. For example, we can roll out a new Rails 2.0 application and run it alongside an older Rails 1.1 application, without having to migrate the older but fully functional code, just because we're using the newer framework for future development.

The ability to scale with ease, mix and match best of breed solutions, and roll out new applications alongside legacy ones, makes reverse proxy our favorite deployment model. So now let's look at actual deployment. We'll start by showing you how to setup Thin and Mongrel for the application servers, and then proceed to cover Apache and Nginx for the front-end Web servers.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

PERFORMANCE BENEFITS OF A FRONT-END WEB SERVER

A well-turned front-end Web server does more than just shield your Web application from the Internet and provide a central point for management and control. Here are five ways in which it can speed up your Web application:

- It can serve static content directly, freeing your application to deal with dynamic content.

- It can compress responses before sending them to the client, cutting down bandwidth and improving response time.

- It can buffer responses on behalf of slow clients. Slow clients keep the application busy waiting to transmit the full response, buffering free the application to cater to the next incoming request.

- Acting as a proxy server it adds another layer of caching between client and server. Make sure to mark responses that are publicly cacheable by using the right Cache-Control directive.

- By keeping connections open without tying up processing thread, it can take advantage of HTTP keep alive and scale to a larger number of concurrent connections.

C.3 Setting Up Thin

True to its name, Thin is light Web server that's incredibly easy to setup (visit code.macournoyer.com/thin/ for more information). We're going to start from an existing Rails application and work our way to having an operating system service that manages multiple application instances.

If you don't already have Thin, now is the time to install it by running `gem install thin`. We're going to be conservative and start out by testing a single instance of the application. From the root directory of your Rails application issue the following command:

```
$ thin start -e production --stats /stats
```

Thin is quick enough that you'll want to use it in development instead of Webrick, and it knows that, so it defaults to run in development mode. Here we're dealing with deployment, so we force Thin to run in production mode just to make sure we got the configuration right. It's easier to check for issues now than later on, when we run Thin as a background service.

Next, point your Web browser to <http://localhost:3000>. We added the `--stats` option so you can also navigate to <http://localhost:3000/stats> and investigate HTTP requests and responses. A common issue with reverse proxy configuration is forgetting to forward an essential request header, and the `--stats` option helps you spot these problems.

Once you verified that everything works as expected it's time to create a configuration file. We'll start with a simple configuration that will run three instances of Thin that listen on ports 8000 through 8002:

```
$ thin config -s 3 -a 127.0.0.1 -p 8000 -e production -C myapp.yml
```

In our configuration, the front-end Web server and Thin all run on the same machine, and since we don't want to expose Thin to clients directly, we just told it to only accept requests using the loopback address 127.0.0.1. If you're running Thin on a separate machine from the front-end Web server, use an IP address that is directly accessible to the front-end Web server, the default (0.0.0.0) will work if you don't care which IP address receives incoming requests.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

We now have a configuration file called myapp.yml, let's see what it looks like:

```
---
pid: tmp/pids/thin.pid
log: log/thin.log
timeout: 30
port: 8000
max_conns: 1024
chdir: /var/www/myapp
environment: production
max_persistent_conns: 512
daemonize: true
address: 127.0.0.1
servers: 3
```

If we wanted to add more servers, change the port or IP assignment, or any other configuration option, we could run the command with different options, or edit this YAML file in a text editor.

Later we're going to run Thin as a service. We're going to have one Thin service per machine, and that service may run any number of applications, so each configuration file must point to the root directory of the application it runs. If you move the application to a different location, make sure to change the value of the chdir configuration property.

Next, let's start Thin using this configuration:

```
$ thin start -C myapp.yml
```

To check that it's running, point your browser to each of the ports, <http://localhost:8000> through 8002, or use the `lsof` command:

```
$ lsof -i tcp@127.0.0.1 -P
COMMAND PID USER  FD  TYPE   DEVICE  SIZE/OFF  NODE NAME
ruby    3321 assaf   3u  IPv4  0x4e74270      0t0  TCP localhost:8000
(LISTEN)
ruby    3324 assaf   3u  IPv4  0xaf1e66c      0t0  TCP localhost:8001
(LISTEN)
ruby    3327 assaf   3u  IPv4  0xb0a0e64      0t0  TCP localhost:8002
(LISTEN)
```

Getting Thin to run as a service depends on the operating system you're using. On Linux you can do that with a single command:

```
$ sudo thin install
```

This command creates a new directory for storing the configuration files (`/etc/thin`) and adds a script in `/etc/init.d` to control the Thin service. When you start the Thin service, it picks all the configurations it finds in the `/etc/thin` directory. If you have more than one application, make sure they all use different port ranges. If you run the same configuration on multiple machines, you could keep the configuration file in the application and link to it from the `/etc/thin` directory:

```
$ sudo ln -s myapp.yml /etc/thin/myapp.yml
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

Last, we're going to setup the service to start and stop automatically, and get it started. On RedHat/CentOS:

```
$ sudo /sbin/chkconfig --level 345 thin on
$ thin start --all /etc/thin
```

On Debian/Ubuntu:

```
$ sudo /usr/sbin/update-rc.d thin defaults
$ thin start --all /etc/thin
```

Setting Up Mongrel

Mongrel and Thin are similar enough that we'll follow the same workflow and just highlight where they differ (visit mongrel.rubyforge.org for more information about Mongrel).

To run a cluster (several instances) of Mongrel, you'll want to `gem install mongrel mongrel_cluster`. We'll start with a single instance in production mode:

```
$ mongrel_rails start -e production
```

To create a configuration that runs three instances on ports 8000 through 8002:

```
$ mongrel_rails cluster::configure -e production -N 3 -p 8000 -a 127.0.0.1 -c $PWD
```

You need to point Mongrel to the root of your Rails application explicitly, which we did here using the `-c $PWD` argument. This command will create a new configuration in `config/mongrel_cluster.yml`.

When run as a service, Mongrel picks up all the configuration files it finds in the `/etc/mongrel_cluster` directory, so either move your configuration file there, or create a symbolic link:

```
$ sudo mkdir /etc/mongrel_cluster
$ sudo ln -s config/mongrel_cluster.yml /etc/mongrel_cluster/myapp.yml
```

To start and stop the service from the command line:

```
$ mongrel_cluster_ctl start
$ mongrel_cluster_ctl stop
```

To deploy Mongrel as a service on Linux, find the `resources/mongrel_cluster` file in the Mongrel gem directory and copy it over to the `/etc/init.d` directory (hint: use `gem contents mongrel_cluster`). Make it an executable with `chmod +x` and configure it to start/stop at the appropriate run levels using `/sbin/chkconfig --level 345 mongrel_cluster on` (RedHat/CentOS) or `sudo /usr/sbin/update-rc.d mongrel_cluster defaults` (Debian/Ubuntu).

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```
On Windows, start with gem install mongrel_service and install a service by running mongrel_rails service::install which takes the same command line arguments or configuration file.
```

Now that we've got Thin up and running, let's configure the front-end Web server using either Apache or Nginx.

C.4 Setting up Apache Load Balancing

With Thin up and running we can turn our attention to the front-end Web server. To setup Apache for reverse proxy and load balancing, you need Apache 2.1 or later, loaded with `mod_proxy`, `mod_headers` and `mod_proxy_balancer`. These modules are included and activated in the default Apache configuration.

We're going to create a new configuration file for our application, `myapp.conf`, and place it in the Apache configuration directory. We'll start by defining the proxy balancer to point at the three Thin instances:

```
<Proxy balancer://myapp>
  BalancerMember http://127.0.0.1:8000
  BalancerMember http://127.0.0.1:8001
  BalancerMember http://127.0.0.1:8002
</Proxy>
```

Put this at the head of the configuration file. If you add more instances (ports), move Thin to a different server (IP address), or need to fine tune how the workload is distributed (e.g. assigning different loadfactor to each instance), this will be the place to make these changes.

Next, and still using the same configuration files, we're going to create a virtual host that handles incoming request using the proxy load-balancer:

```
<VirtualHost *:80>
  ServerName example.com
  ServerAlias www.example.com
  DocumentRoot /var/www/myapp/public

  # Redirect all non-static requests to application,
  # serve all static content directly.
  RewriteEngine On
  RewriteCond %{DOCUMENT_ROOT}%{REQUEST_FILENAME} !-f
  RewriteRule ^/(.*)$ balancer://myapp%{REQUEST_URI} [P,QSA,L]
</VirtualHost>
```

Our application serves dynamic contents, but also includes static content like images, CSS stylesheets, JavaScript and so forth. Apache can handle these efficiently and without burdening the back-end server, so we tell Apache to serve any file it finds in the document root directory, and dispatch all other requests to the back-end server. Rails puts all the static content under the public directory, so point DocumentRoot there. Never point it to the root directory of the Rails application, unless you want people to download your application's source code and configuration files.

The reverse proxy forwards requests to the back-end application using HTTP, so when configuring a virtual host to handle HTTPS request, it must inform Rails the original request came over HTTPS by setting the forwarded protocol name. You can do that by adding this line to the virtual host configuration:

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```
RequestHeader set X_FORWARDED_PROTO 'https'
```

Now you're ready to restart Apache (`sudo httpd -k restart`), open the browser to `http://localhost` and watch the application in action.

Before we complete, here's another trick to add to your arsenal. Apache includes a Load Balance Manager that you can use to monitor the state of your workers and change various load balancing setting without restarting Apache. The following configuration will enable local access to the Load Balancer Manager on port 8088:

```
Listen 8088
<VirtualHost *:8088>
  <Location />
    SetHandler balancer-manager
    Deny from all
    Allow from localhost
  </Location>
</VirtualHost>
```

Setting Up Nginx

Nginx is an up and coming Web server, not as popular as Apache, and unfortunately not as well documented, but offers better performance and scalability. At its core, Nginx has a smaller memory and CPU footprint and uses non-blocking I/O libraries, so it can scale to larger workloads and maintain a higher number of concurrent connections. We recommend giving Nginx a try, and will show you how configure it as we did for Apache (you can read more about it at nginx.net).

We're going to create a new configuration file for our application, `myapp.conf`, and place it in the Nginx configuration directory. Some setups of Nginx will include all the configuration files placed in a certain directory (typically, `/etc/nginx/enabled`), if not make sure to include it from within the `http` section of the main configuration file.

We'll start by defining the upstream Web server that points to our three Thin instances:

```
upstream myapp {
  server 127.0.0.1:8000;
  server 127.0.0.1:8001;
  server 127.0.0.1:8002;
}
```

Next, and still using the same configuration files, we're going to define a virtual host that handles incoming request by dispatching them to the upstream server:

```
server {
  listen 80;
  server_name example.com;
  gzip on;
  proxy_buffering on;
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

```

location / {
    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header Host $http_host;

    root /var/www/myapp/public;
    if (!-f $request_filename) {
        proxy_pass http://myapp;
        break;
    }
}

```

When dealing with HTTPS request, add this line to the list of forwarded headers:

```
proxy_set_header X-Forwarded-Proto 'https';
```

If you're running Nginx and Thin on the same machine, you can take advantage of UNIX sockets, by configuring the upstream server to use them:

```

upstream backend {
    server unix:/tmp/thin.0.sock;
    server unix:/tmp/thin.1.sock;
    server unix:/tmp/thin.2.sock;
}

```

You will also need to configure Thin to use sockets instead of IP ports:

```
$ sudo thin config -s 3 -S /tmp/thin.sock -e production -C /etc/thin/myapp.yaml
```

In this appendix we covered the various options for deploying Ruby-based Web applications, focusing specifically on the reverse proxy architecture. The benefits of this architecture is the ability to scale from a single-server deployment to a cluster of application servers, and the ease of mixing different Web applications and languages. We showed you how to set up a cluster using two different lightweight Ruby servers, Thin and Mongrel, and on the front-end how, using the popular Apache Web server and the lighter and faster Nginx.