Merb in Action

Michael D. Ivey
Yehuda Katz
Ezra Zygmuntowicz

MEAP Unedited Draft

MANNING

Table of Contents

1 Introducing Merb

In the world of Web 2.0 (and with shiny new Web 3.11 just around the corner), frameworks for building database-driven web applications are easy to find in your language of choice. In Ruby alone there's Rails, Camping, Sinatra, Waves, Nitro, Ramaze, IOWA … and the list goes on.

Another of the Ruby frameworks is Merb. Fast, flexible, and very accessible, Merb describes itself as "All you need. None you don't." Merb comes with everything you need to build fast, scalable web applications (and then some), and yet maintains a very high level of flexibility and configurability. Developers choose frameworks for many reasons, but many are drawn to Merb initially because it offers them flexibility in how to approach their applications. While most use Merb in a traditional MVC style, Merb itself allows many approaches and encourages developers to adapt the framework to their application's needs.

Since Merb grew up alongside Ruby on Rails, there's a strong temptation to compare the two. We'll try to avoid explaining Merb in terms of Rails after this chapter. Where it would be helpful, however, we'll call out features of Merb that are implemented differently in Rails, to help ease the transition for those coming from a Rails background.

Before we can understand why Merb matters and why it works the way it does, let's first take a look at the state of things around Rails version 1.1.5.

1.1 Historical Context

Officially announced in October 2006, Merb started life as a small hack. Designed by Ezra Zygmuntowicz to work alongside Ruby on Rails applications, Merb was intended for tasks, like file uploads, where the Ruby on Rails framework underperformed. Just months earlier, Apple had announced that it would include Rails with the next version of its flagship operating system, OSX Leopard, a sign of the framework's maturity. In the years since, Merb has found itselfmaturing from a side-hack to a full-fledged framework in its own right.

As Merb continued to attract users and contributors, people began using it for more than just supplementing existing Rails applications.

NOTE To see how simple Merb was at the beginning, check out the entire source in less than 120 lines: http://pastie.caboo.se/14416

Instead of focusing on making it easy for brand-new programmers to put together a blog or shopping cart in minutes, Merb found itself focusing on modularity, efficiency, speed, and thread-safety. But since Merb took its basic architecture from Ruby on Rails, it shares much of the ease of use that brought so many new web programmers to Ruby in the first place.

As the community continued to grow, much of the ease of use that was missing in earlier versions of Merb was added in to both the core framework and as plugins.

Because Merb values modularity through plugins, that meant that most of the new features forced the framework designers to think intelligently about keeping the core of the code light and easy to extend. In fact, much of what makes up Merb is implemented as plugins on top of the core, but you'll hardly notice unless you start digging in yourself. We'll cover a lot of those details in Part II, when we crack open the hood of Merb and take a look at how it all fits together.

1.2 Ruby

Popular with a devoted group of Japanese programmers since it's inception in the 1990s, Ruby did not attract a lot of attention in the United States until about ten years later. Yukihiro Matsumoto (affectionately known as "Matz") designed Ruby as a language that focused on developer joy and ease of use. Ruby programs are often evaluated not only by how well they perform or meet the design specifications but how aesthetically pleasing the code is. Some Ruby programmers compare the process of writing code to the process of writing poetry.

Newcomers to Ruby are often surprised and delighted to find an open and helpful attitude present in Ruby forums and chat rooms. They are experiencing a defining characteristic of the Ruby Community - the people involved are just plain nice. This attitude is not a spontaneous occurrence, it has been cultivated since the early days of the language. This principle of niceness has its own acronym, MINASWAN, which stands for "Matz Is Nice and So We Are Nice".

With a language focussed on beauty and programmer happiness, and a friendly and helpful community, it's no wonder that we love Ruby as much as we do.

Throughout the book, we assume a certain level of comfort with the Ruby Programming Language. For readers new to Ruby, the appendix contains a basic introduction to Ruby concepts to help with the transition to Merb.

1.3 Why another MVC framework?

With the existence of Ruby on Rails, Camping, Waves, and Nitro, you might be asking yourself: Why another framework? And in particular, why another framework that uses the Model, View, Controller paradigm when Rails is so popular, successful and easy to use.

NOTE Model, View, Controller is a style of programming that emphasizes breaking your code into three components. The models control access and control data sources. The Views represent the visual representation of the data sent to the user. Finally, the controller stitches the two together, passing useful model data to the views. In the case of web frameworks, the models typically represent access to a database, the views are HTML templates, and the controllers route requests to their appropriate templates.

For many programmers, Rails solves all problems. It provides a useful abstraction that was originally extracted from a real application, BaseCamp, and that provides a toolkit that can get people up and running quickly. However, that ease of use is quickly supplanted by the opinionated nature of the framework. In order to simplify the decisions you'll have to make while using Rails, its framework designers make many decisions for you. That means that overriding those defaults can begin to consume most of a Rails programmer's time. It is said that any non-trivial Rails application involves hacking on the core framework, and your mielage may vary.

On the other hand, Merb is designed around the notion of modularity, to the extent that the core framework itself is designed as a group of modules. Because the framework designers need to keep hooks open for the framework itself, you can be sure that you'll be able to override small and large pieces of functionality as necessary. In other words, Merb is designed to be hackable.

Designed from the ground-up with efficiency in mind, Merb is also substantially faster than its counterparts, being able to spit out an impressive 2,000 requests per second for simple strings, and more than 1,500 requests per second when using templates. At time of press, it is consistently two to five times faster than Rails, with higher benefits coming from applications that make heavy use of partials. In fact, part of Merb's release process is a comprehensive benchmark. If a release of Merb runs slower than the previous release, the core team tweaks it to beat the previous release.

Merb also maintains a healthy agnosticism toward companion libraries. For instance, it works out of the box with both `Test::Unit` and `rspec`, works with any of the `ActiveRecord`, `DataMapper`, and `Sequel` ORMs, and works with any of the myriad JavaScript libraries out there. Merb achieves this by providing basic hooks into the framework, and allowing plugins (like `merb_test_unit`, `merb_activerecord`, etc.) to provide functionality appropriate to the component in question. This goes as far as to generate the appropriate type of stub during model generation, providing a simple mechanism for using Merb that shapes itself around your choices.

Now, let's take a closer look at what these core elements of Merb's philosophy mean in practice.

1.3.1 Hackabilty

We'll take a more complete look at exactly how clean and modular the Merb framework is in Part 2, but suffice it to say that its designers built it with hacking in mind.

Let's take a look at a simple example involving Merb's Controllers and Mailers. Instead of implementing web controllers and mailers separately, with web controllers containing all the logic for handling requests and responses, and mailers containing the logic for sending mail, Merb takes a more modular approach. Merb defines a super-class, `Merb::AbstractController`, which contains all the logic for helpers, layouts, before and after filters, and simple dispatch. `Merb::Controller` inherits from the `AbstractController`. A separate module, called `merb-mailer`, defines a `Merb::MailController` which also inherits from `AbstractController`.

This is an immediate win for Merb, because users of the framework can leverage their knowledge of web controllers to write DRY and modular mailers with all of the features of web controllers. And because both modules inherit from the same module, improvements to the general module automatically percolate down to all inherited modules.

NOTE Ruby on Rails has two separate classes, ActionController and ActionMailer, which are implemented separately and have two different general APIs. While ActionController supports layouts, helpers and instance variable assignment, ActionMailer is much simpler, with none of those features.

Because of this clean approach, it's very easy to override the existing behavior or define new controller types. For instance, another optional module, `merb_parts`, provides support for simple, light, reusable components with full support for helpers, filters, layouts, and content-negotiation.

Merb is also built on top of the Rack interface, which provides a single, unified interface for web servers. This allows Merb to easily support any web server including Mongrel, Evented Mongrel, Thin, FCGI, CGI, and even Webrick. And because the Rack interface is so simple, Merb's console mode is implemented as a Rack stub in under 50 lines of code.

As a result, understanding and hooking into Merb's web server interface is trivial. And get this: Rack's interface allows you to mount multiple applications, which it will try one at a time until one of them does not return a 404.

Finally, the Rack interface made it easy for JRuby, using the Glassfish web server, to connect directly with Merb in just a few dozen lines of code. This means that JRuby's Merb support is rock-solid and won't break in a future version of Merb.

In a later chapter, we'll show you how to use Rack to mount simple Rack applications to handle things like RSS feed generation, so you can still use Ruby, but skip the overhead of Merb for cases that don't need the entire framework. It's features like this that can help you build more efficient apps.

1.3.2 Performance

Designed with performance in mind, Merb has surpassed other Ruby web frameworks, including Rails by leaps and bounds. While Ruby apps are capable of "scaling" just fine by adding additional hardware, the Ruby language can sometimes make it easy to introduce features that prevent efficient scaling. And while it has become conventional wisdom by now that Rails apps can scale by simply throwing hardware at increased demand, the central argument is flawed. People who have come to Ruby have been convinced that since human developers are more unique (and more expensive) than hardware, it's worthwhile to use frameworks that trade development time for hardware costs.

While that might be true if the tradeoff was so neat, it introduces a false dillemma. As Merb has shown, it's possible to build a framework that can scale efficiently while still maintaining simplicity for developers. In real-life terms, a framework that can serve twice as many pages per second requires half as many servers. And the mere fact of needing to introduce additional boxes to handle scaling adds complexity to a deployment, which can eat into the developer savings.

Merb provides performance and scaling that allow you to benefit from the developer ease of Ruby while still maintaining respectable performance stats. Also, because it's a Ruby app, you get the same general deployment and scaling approach as other Ruby frameworks. In other words, the work the community has done to solve the general issues in this area is directly applicable to Merb, and we'll discuss that in much greater depth in Chapter 5, Deployment.

1.3.3 Agnosticism

As the Rails community matured, and the framework found itself in more varied kinds of applications, developers found they didn't always agree with the opinionated choices Rails makes. Lots of developers started to use `rspec`, an alternative testing framework to Rails' built-in support for Test::Unit. New ORMs, including DataMapper and Sequel, came out of the woodwork and began to flourish. And for JavaScript, many Rails developers use jQuery or MooTools rather than the built-in support for Prototype and Scriptaculous.

The problem is, Rails prides itself on its opinionated nature and batteries-included packaging. As a result, users of rspec find themselves out of luck when they generate new controllers (Rails, by default, creates `Test::Unit` stubs). While the creators of `rspec` eventually created packages to hook deep into Rails and replace the built-in support for `Test::Unit`, they use interfaces that are subject to breakage with new versions. Similarly, dropping in a new ORM, like DataMapper, is possible in Rails only if the creators of the ORM make special efforts to support Rails. As more plugins hook deeper into Rails, the larger the chance of conflicts and breakage.

And obviously, many Rails plugins make assumptions about things like template languages. For instance, plugins designed to use Rails' built-in ERB template language may not work correctly with Haml, Markaby, or Liquid.

Merb, on the other hand, takes special care to provide hooks into the base framework, instead of making opinionated assumptions about what collection of tools our users will use. By default, Merb uses Erubis, Rspec, DataMapper, and jQuery (so new users can get a full, batteries-included stack), but nothing in the core framework assumes that stack. In fact, with the exception of Erubis, the "default" Merb stack is simply a set of plugins designed, maintained, and cared for by the Merb core team.

Here's an example of how Merb implements templating, to keep things agnostic: template languages are required to expose a single method (`compile_template`), which takes the path of the template, the name of the method that will be compiled, and the module it will be compiled into. Don't worry about the details; we'll dig into the inner workings of Merb in Part III. The point, however, is that Merb provides very simple points of interaction with other modules, assuming very little so that you can drop in new template languages, ORMs, testing frameworks, or JavaScript libraries with no difficulty at all.

1.3.4 Yikes!

At this point, you might be thinking that Merb forces you to make all sorts of decisions yourself. In the name of modularity, efficiency, and hackability, it seems, we're throwing new developers under the bus.

Don't worry, new Merbivorian.

The Merb maintainers understand that not everyone's ready to dive in and configure every little piece of their new framework. While there might come a day where being able to fine-tune your app becomes important, you can start your journey with a Batteries Included version of Merb, which includes merb-core (the core framework) as well as merb-more (generators, mailers, alternate template languages, authentication, caching, etc.). From your perspective, you'll just need to do `gem install merb`, and for now, not need to worry about how everything fits together.

The core team also maintains a series of high-quality plugins, including plugins for Ruby ORMs, support for Test::Unit, and some additional helpers that are not included in `merb-more`. You can see how everything fits together in Figure 1.1.

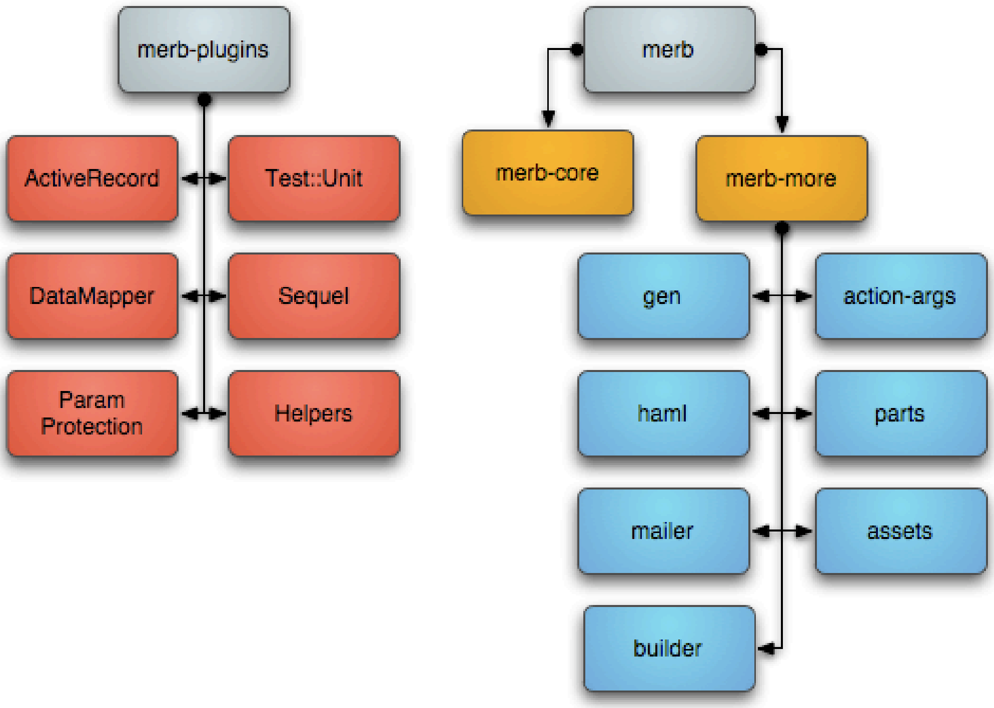


Figure 1.1 Merb components

As we move forward, we'll highlight what component part of Merb contains each feature we discuss.

1.4 A taste of Merb

Nothing gives a better taste of a framework than code. Over the next few pages we're going to show you some pieces of Merb applications. If this feels rushed, that's because it is. These examples aren't designed to teach you how to do these steps. We just want to give you a general feeling for Merb. Where appropriate, we'll note the chapters or pages you can reference for more information.

1.4.1 What you return is what you get

Whether it's Ruby or Java or C#, whether it's a Web framework or a GUI toolkit, with many frameworks you end up writing code that doesn't match the idioms of the language. Your application looks like others written with that framework, but may not "feel like Ruby".

Merb's controllers try to be good Ruby citizens. When you instantiate one it is in a consistent state. All of the controller's dependencies are passed in via the constructor. Most of the methods are public. In short, Merb controllers act like Ruby objects instead of pieces of the framework. Because of this, Merb controllers are easy to test and they're easy to re-use. Here's how you might create a new controller instance in a test.

Listing 1.1 Creating a controller and calling an action

```
@controller = Cats.new(@mock_request)
@controller._dispatch("index")
```

That's all there is. (You can't call the action directly with `@controller.index` because Merb runs filters before and after the action method.) Merb is very tolerant when it comes to what gets passed in as the request, too: as long as it acts like a Merb::Request, you can pass in a mock or a test stub.

Now let's look inside that example controller and see how the response body is generated. This is another place where Merb's simplicity is sure to make Ruby programmers feel at home.

Listing 1.2 Returning a response

```
class Cats < Application
  def index
    "I know about these cats: " + find_all_cats.join(", ")
  end
end
```

Merb uses the return value of the action as the response body. This means that whatever data you would get from calling `@controller._dispatch("index")` is the same data that will be sent to the web browser. Merb controllers behave like other Ruby objects would.

Of course, we need more than just strings, and no one wants to generate HTML by hand, so let's look at just one of Merb's options for rendering templates.

1.4.2 Templates and web services

Chances are you've either built a web service, or thought about building one. With Web 2.0 data exchange is on everyone's mind. Merb gives you some simple and powerful tools for adding RESTful web service support alongside a traditional application.

NOTE Representational State Transfer (REST) is an architectural style for building applications, originally explained by Roy Fielding. See http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm for his original paper. REST also describes an implementation of that architecural style that maps HTTP verbs to standard actions, as implemented in Rails. Merb supports Rails-style REST and more general RESTful concepts.

Let's return to our application that manages a database of cats. This time, we'll assume we've done a lot more on it, and have an index and show action, both working with our models and rendering HTML.

Listing 1.3 Cats controller

```
class Cats < Application
  def index
    @cats = Cat.find_all
    display @cats
  end

  def show
    @cat = Cat.find(params[:cat_id])
    display @cat
  end
end
```

We'll go into much more detail on the `display()` method in chapter XX. For now, just know that if it finds a template for the action and the current content type, it will render it. Otherwise, it will try and coerce the object into that content type.

Since we haven't changed any of the standard settings, this controller only supports HTML output. Suddenly we realize that our cat database needs XML support, so we can share data with a hot new pet aggregator. Because Merb is ready to handle multiple content types, we only have to add one line.

Listing 1.4 Adding XML support

```
class Cats < Application
  provides :xml

  # ...

end
```

Once we tell Merb that our controller handles XML, we're done.

(Mostly. There's a little more to the story, which we'll cover later, but for this high-level picture, it really is that simple.)

1.4.3 When things go wrong

We can't expect everything to always go our way. Users give us bad data, databases crash, things fall apart. In Ruby, when things go wrong, we're used to raising exceptions. Merb brings that philosophy to dealing with error handling inside controllers.

Let's handle the case where the cat the user asked for doesn't exist. Our show action from the previous section would look like this.

Listing 1.5 Cats#show with error handling

```
class Cats < Application
  def show
    @cat = Cat.find(params[:cat_id])
    raise NotFound unless @cat
    display @cat
  end
end
```

Now, if the cat record isn't found, the end-user will see a "404 Not Found" error. Merb defines custom exceptions for each of the HTTP status codes. If the HTTP spec ever adds payment rules, you can `raise PaymentRequired` as part of your business plan.

1.4.4 Web parameters and method arguments

There will always be a little bit of a disconnect between application programming and the web, but Merb has a few ways to lessen it. Parameterized actions (part of `merb-more` in `merb-action-args`) makes your actions look like traditional Ruby methods, including default values for optional arguments. Compare the example below with the previous example.

Listing 1.6 Cats#show with action args

```
class Cats < Application
  def show(id)
    @cat = Cat.find(id)
    raise NotFound unless @cat
    display @cat
  end
end
```

Instead of using a magic `params` hash, Merb passes `params[:id]` into the id parameter of the show action. This is another way Merb lets you treat your controller classes like a regular Ruby class.

Now that we've gone through a quick tour of Merb, let's take a look at some of the use cases.

1.5 When to use Merb

When should you use Merb? Whenever possible, we hope! While Merb is primarily appropriate for building web applications, that's not even a real limitation (see future note/example re: non-web Merb app). Inside merb-core you'll find everything you need to build simple web applications, including basic authentication and X. Most Merb applications involve a database and take full advantage of the various ORM plugins. Merb shines for non-database driven applications, though. Imagine an API that aggregates 3 different feeds into a unified format. Because you only include what you need, there's no ORM or database layer involved at all. Merb is especially well-suited for building APIs where there isn't a web front-end, using the web as a transport layer. Because you can use just merb-core, you can strip out everything you don't need, and build a small, light-weight application. Another application that's common is using Merb to upload files into an existing Rails application.

When shouldn't you use Merb? If you want transparent Javascript updates generated by your Ruby code, you'll notice that Merb doesn't offer any of that, by design, and instead encourages you to use the methods described as "Unobtrusive Javascript". If you're looking for the same level of ease and simplicity as you find in Rails, Merb may not make you happy. Merb trades some of the simplicity and convenience for flexibility.

1.6 Summary

Merb was created out of a belief that Ruby on Rails has a number of excellent ideas, but that it can be sorely lacking in implementation, especially in the core areas on hackability, performance and vendor lockin. Merb is built around a lot of the same core concepts, like MVC architecture, convention over configuration, and a batteries-included stack, but makes it much simpler to override and swap out parts of the stack that aren't working.

In the next few chapters, we'll get you up and running with Merb. We think you'll be pleased with how simple getting started can be, and if you've used Rails, you're sure to experience a pleasant sense of déjà vu. That's because many of the most pleasing parts of getting up and running with Rails are almost identical in Merb.

Let's go!

2 Getting started with Merb

In the previous chapter, we touted all of the benefits of Merb's modularity. You might be thinking that the best way to get up and running would be to install Merb Core and add features from Merb More and plugins as you need them. And in fact, that is how advanced Merb developers typically architect new applications. We'll start with something simpler.

Antoine de Saint Exupéry (author of The Little Prince) said, "Perfection is attained not when there is no longer anything to add, but when there is no longer anything to take away." (Terre des Hommes, 1939.) That's the approach we'll take as we begin to explore the Merb framework. Instead of forcing you to learn how all the pieces fit together, and what modules specific pieces of functionality fit into, we'll treat Merb as a monolithic whole. As you get more advanced, you will get proficient at configuring your own distribution of Merb, perfect for your application.

In Part II of this book, we'll dig deep into the Merb framework, in a sort of motorcycle manual to the internals. When you emerge, you will understand all of the cogs and wheels of the internals of the framework, proud to call yourself a master. For now, grasshopper, let's dip your toes into the pool of knowledge. The best way to learn is to blur the lines a bit between the modules.

This chapter will get our toes wet by building a simple sandwich shop application. We'll start by building some static pages, and then start the process of creating menus in the database with DataMapper. We'll finish the project in Chapter 3, when we take a look at DataMapper in more depth.

But before we can do that, we'll need to get Merb installed.

2.1 Getting ready for Merb

Getting up and running with Merb requires that you have a basic installation of Ruby. First, let's cover installing standard Ruby, or Ruby 1.8.6, currently maintained by Yokihuru Matsomoto and available from ruby-lang.org. Later on in this chapter, we will cover getting up and running with JRuby, Rubinius, and Ruby 1.9, which all have their own caveats.

NOTE The version of Ruby that Merb is known to be most stable on is Ruby 1.8.6. Ruby 1.8.7 includes some experimental features of the 1.9 development branch, and is not considered as stable by Merb developers.

The following sections cover installing Ruby on various platforms, including some prerequisites that require compliation. You can skip to the section that covers your platform.

2.1.1 *Installing Ruby on Windows*

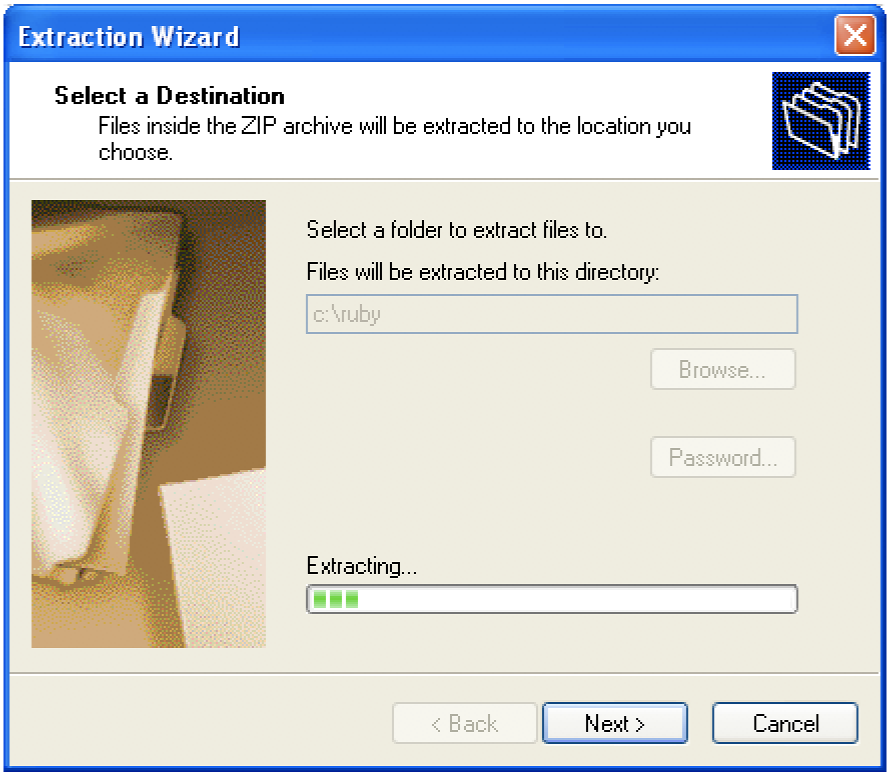
The easiest way to install Ruby on Windows is the Ruby One-Click package. You can download it from Ruby's official site at http://www.ruby-lang.org/en/downloads. Installing the package will get you up and running with Ruby and Rubygems, and you'll be able to follow along with the command-line instructions in this book with no further configuration.

NOTE Several times throughout this book, we will refer to Ruby packages that need to be compiled. If you are using Windows, this is a blessing and a curse. If the Ruby package in question has been ported to Windows (such as the popular HTML parsing library called Hpricot), installation is a cinch, because all of the hard work that can plague Unix systems has already been performed for you. Unfortunately, it also means that you'll need to wait for new packages to get ported, which can sometimes take a while. For now, you don't need to worry about that. All of the dependencies for Merb, its optional modules, and its official plugins are available for Windows.

2.1.2 *Installing Ruby binaries on Windows*

The One-Click package tends to lag behind more recent releases of Ruby, so if you're interested in staying up to date, you can also install the precompiled binaries of Ruby, which can also be downloaded at http://www.ruby-lang.org/en/downloads.

1. Once you have downloaded the package, you will need to unzip it. An easy place to unzip it into is C:\ruby. We will assume that's where you unzipped it to for the rest of these instructions.


Extraction Wizard

Select a Destination

Files inside the ZIP archive will be extracted to the location you choose.

Select a folder to extract files to.

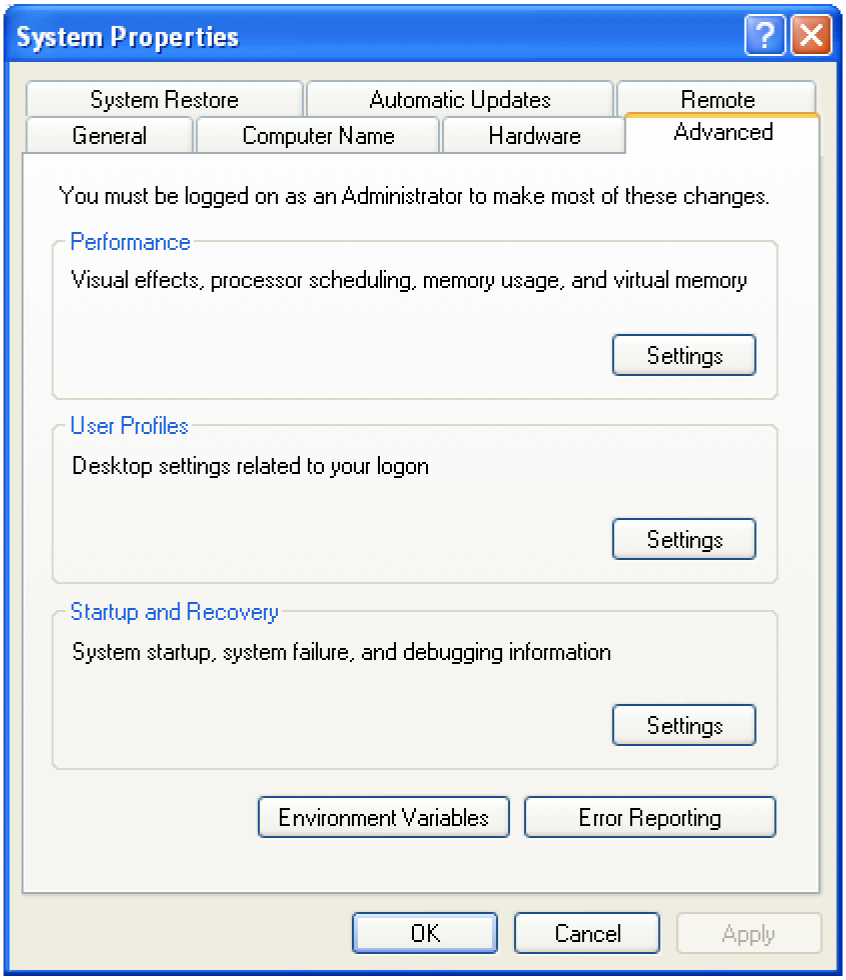
Files will be extracted to this directory:

c:\ruby

Extracting...

Figure 2.1 Extracting the Ruby zip file

2. In order to use the binary, you will need to set your path to include the new C:\ruby folder, so you can reference the Ruby binary from anywhere in your system. To do this, open up the control panel and select System. Next, select the "Advanced" tab.



You must be logged on as an Administrator to make most of these changes.

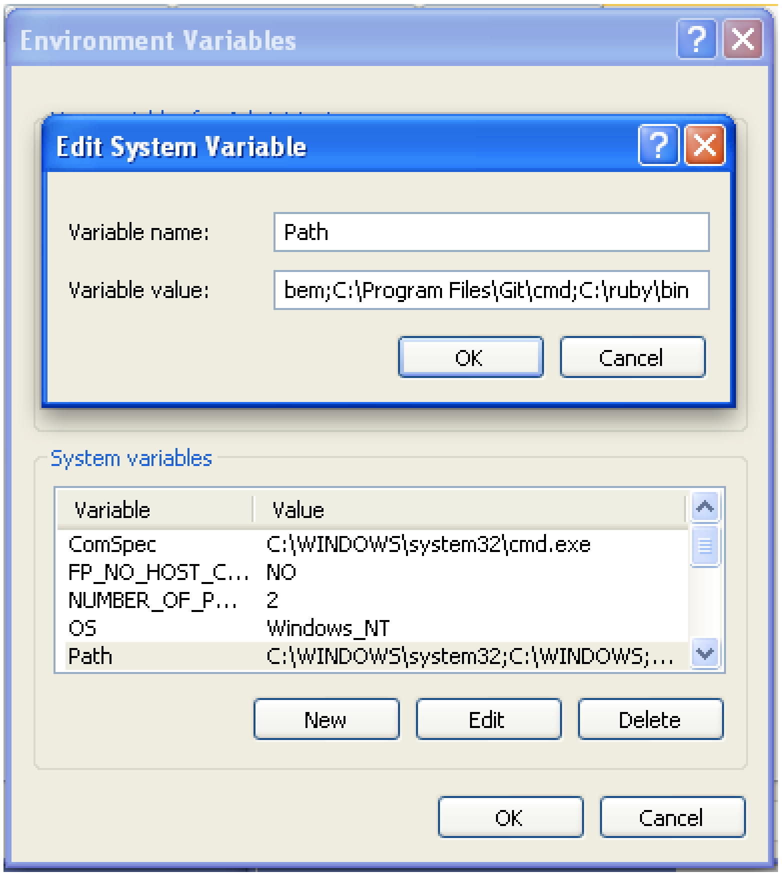
Visual effects, processor scheduling, memory usage, and virtual memory

Desktop settings related to your logon

System startup, system failure, and debugging information

Figure 2.1 Viewing the Advanced Tab

3. At the bottom of the window, you will see an “Environment Variables” button. Click it.
4. You will see a window with two grids. The bottom grid will be marked “System Variables”. One of the variables will be called “Path”. Click “Edit”.
5. In the “Variable Value” box at the bottom, add `;C:\ruby\bin`.


Environment Variables

Edit System Variable

Variable name: Path

Variable value: bem;C:\Program Files\Git\cmd;C:\ruby\bin

System variables

Variable	Value
ComSpec	C:\WINDOWS\system32\cmd.exe
FP_NO_HOST_C...	NO
NUMBER_OF_P...	2
OS	Windows_NT
Path	C:\WINDOWS\system32;C:\WINDOWS;...

Figure 2.1 Setting the path

6. Now that you have Ruby installed, you'll need to install RubyGems, Ruby's package manager. Head over to http://rubyforge.org/frs/?group_id=126&release_id=21058. You'll want to grab the .zip of the newest release of Rubygems (the one at the top). Extract it to c:\ruby\rubygems.
7. Open up the Start Menu and select "Run". Type `cmd`
8. You will be at a command prompt. Change the directory (via cd) to `c:\ruby\rubygems`, where you extracted the Rubygems zip file. If you type `dir`, you will see a number of files, including `setup.rb`.
9. Type `ruby setup.rb` at the command line. Information about the installation will scroll up your screen. When the installation is complete, you will have a working `gem` command that you can use from the command line.

With that out of the way, it's time for a breath of fresh air: installing Ruby on Mac OSX. This is the platform that the majority of Rubyists and virtually all of the Merb team develop on.

2.1.3 Installing Ruby on Mac OSX

As it turns out, Mac OSX Leopard comes with Ruby and Rubygems preinstalled with Leopard.

On previous versions of Ma **Installing Ruby binaries on Windows** nt to install Ruby through MacPorts. Some Mac Rubyists install Ru a matter of course, even on Leopard, as it allows them to keep their up to date as well as ensure that any bugs that Apple accidentally in sion of Ruby that comes preinstalled do not hinder their ability to ore developers are mixed; some use Leopard's built-in Ruby and ot ough MacPorts.

1. Visit http://svn.macports.org/re ownloads/MacPorts-1.6.0/
2. You will see a number of files. You are looking for the latest version of MacPorts for your version of Mac OSX (e.g. MacPorts-1.6.0-10.3-Panther.dmg). Download it.
3. Mount the dmg file that you have downloaded. It will appear as a new volume in the Finder.
4. Double-click on the MacPorts package and follow the installation instructions.
5. Now that MacPorts is installed, open Terminal.
6. Enter `sudo port install ruby`.
7. Once Ruby has finished installing, enter `sudo port install rb-rubygems`.

NOTE Many Mac Rubyists install Ruby from source, bypassing ports altogether. We will not cover that technique here, but you can get the files you need from http://www.ruby-lang.org/en/downloads. If you want to track even closer to Ruby's HEAD, you can get Ruby directly from subversion by following the instructions at http://www.ruby-lang.org/en/community/ruby-core/. Keep in mind that if you do opt to track close to Ruby's HEAD, bugs you experience in Merb might actually be bugs in Ruby itself.

Next, let's take a look at how to install Ruby on Linux.

2.1.4 Installing Ruby on Linux

The easiest way to install Ruby on Linux is to use the package manager for your distribution. Unfortunately, people have experienced issues with the packaged Rubygems on Debian and Ubuntu. On those distributions, you should install from source. Follow the instructions in the `README` file of the downloaded code to compile and install Ruby from source.

Now that Ruby's installed, let's tackle another topic that can be tricky: installing rubygems that require the support of your native environment in order to install.

2.1.5 Installing Native Gems

In order to use some features of Merb, you will need to have the ParseTree gem. This will typically install just fine. A precompiled binary exists for Windows that will install automatically, and Unix environments should have no trouble compiling them on demand. This should happen by default when you install Merb, which we will discuss next. In some rare cases, however, ParseTree will fail to build. If you experience these issues, Google is your friend. You will typically be able to find someone else who has experienced exactly the same problem as you. You can also stop by the #merb channel in IRC on irc.freenode.net. People in the IRC room are friendly, patient, and will likely know something about the issue you're having.

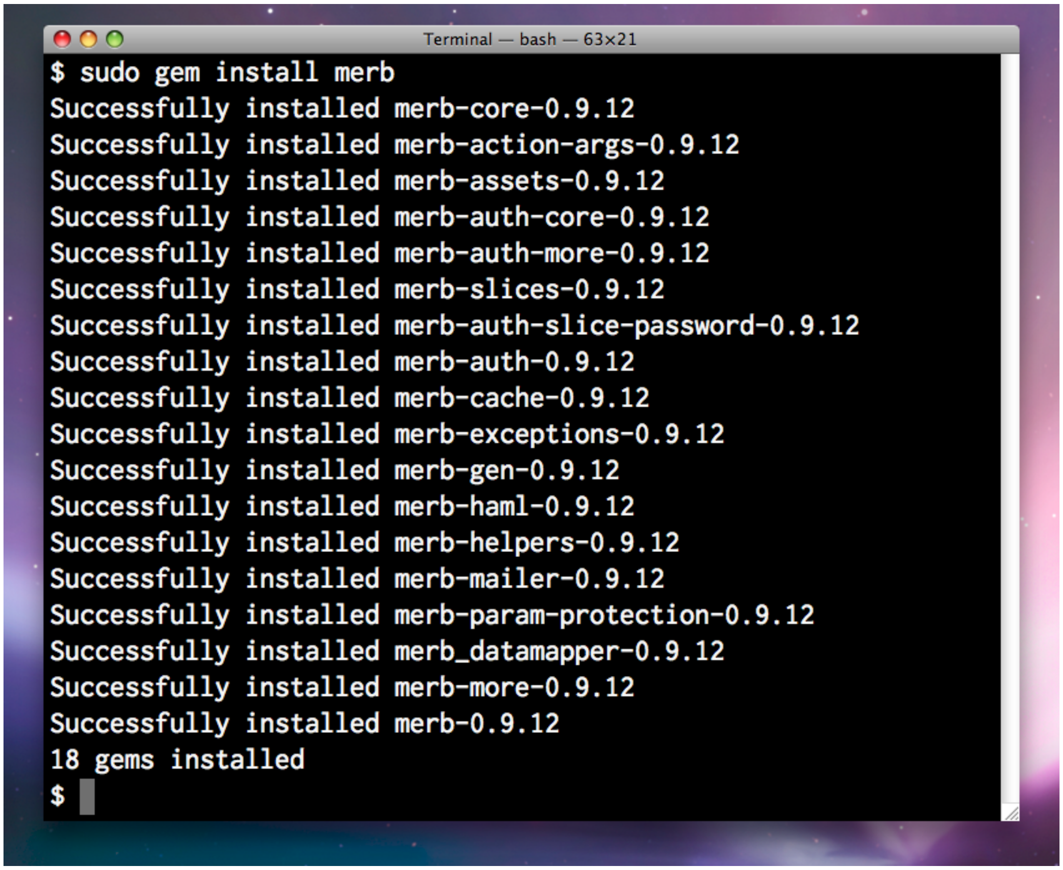
Once your Ruby environment is set up, it's time to install Merb.

2.2 Installing Merb

This should be pretty straight-forward, but let's walk through two ways to get Merb. The first, easier way is to simply install Merb from gems. The second, harder way, is to install Merb from source. This will allow you to get closer to the most recent advances in the framework, but trunk versions of Merb are more unstable than the released gems.

2.2.1 Installing Merb from Gems

The easiest way to install Merb is to go the commad line and type `gem install merb`. If you're on a Unix, you likely need to prefix with `sudo`, as in `sudo gem install merb`. You wll see a large number of gems install. This is normal. As we said earlier, Merb is made up of a number of modules, and `gem install merb` will automatically download and install all of them for you.



```
$ sudo gem install merb
Successfully installed merb-core-0.9.12
Successfully installed merb-action-args-0.9.12
Successfully installed merb-assets-0.9.12
Successfully installed merb-auth-core-0.9.12
Successfully installed merb-auth-more-0.9.12
Successfully installed merb-slices-0.9.12
Successfully installed merb-auth-slice-password-0.9.12
Successfully installed merb-auth-0.9.12
Successfully installed merb-cache-0.9.12
Successfully installed merb-exceptions-0.9.12
Successfully installed merb-gen-0.9.12
Successfully installed merb-haml-0.9.12
Successfully installed merb-helpers-0.9.12
Successfully installed merb-mailer-0.9.12
Successfully installed merb-param-protection-0.9.12
Successfully installed merb_datamapper-0.9.12
Successfully installed merb-more-0.9.12
Successfully installed merb-0.9.12
18 gems installed
$
```

Figure 2.1 Installing Merb

Now that Merb is installed, you can skip down to Section 2.3, Building the Sandwich Shop. Unless, that is, you want to learn more about how to track Merb's development between releases.

2.2.2 Installing Merb from source

For most purposes, installing the gems using the `gem` command is the simplest and easiest way to get Merb. Sometimes, however, you'll want to track the development of the framework more closely. Insiders refer to this as "being on Merb edge." While tracking edge is not for the faint-of-heart, it will allow you to use features that have not yet been released to the general public, with the caveat that they have not yet been completely tested for stability, or packaged up as a release with all the accompanying documentation.

With that said, Merb does have a comprehensive test suite, which helps the team track down bugs that are introduced between builds. Additionally, Merb has a continuous integration server, which automatically runs those tests on every commit, and alerts the team if someone commits a broken build. If you want to track edge, you can run the test suite yourself to give yourself some modicum of assurance that things aren't horribly out of whack.

The easiest way to install Merb from edge is to use the official thor tasks.

Before you install Merb's edge sources, you will need to install git. On Windows, you can download a binary from http://code.google.com/p/msysgit/downloads/list. On OSX, you can use `sudo port install git-core`. On Linux distributions, it should be available for your package manager. Once you have git:

1. Open your terminal
2. Install thor by typing `sudo gem install thor`.
3. The thor tasks are installed in a newly generated app.
4. Run `thor merb:stack:install --edge`

In the future, if you want to update your Merb installation from source, go back into your Merb directory, and perform the following step.

1. Type `thor merb:stack:install`. This will uninstall the Merb gems, update the sources, and reinstall.

To verify that everything is installed correctly, you can run `merb --version`. If you have installed from the git sources, you will see the current version of Merb, which should be one higher than the stable release.

Now we're finally ready to turn to our application, the Merb Sandwich Shop.

2.3 Building the sandwich shop

In this chapter (and the next) we'll be building an application to run a sandwich shop. We want it to have basic information about our shop like directions and hours. We also want to list our menu of sandwiches, and make it easy for customers to learn the ingredients and nutrition information.

NOTE In general, this book will use a new example per chapter, to make it easy to skip around the book without the extra overhead of needing to know what happened in previous chapters. This chapter is an exception. We'll finish the example we start here in Chapter 3.

Merb comes with a set of code generators that will build parts of your application for you. Without further ado, let's generate the application.

2.3.1 Generating code with merb-gen

Merb’s generators can perform a number of functions, including generating your entire application, generating controllers, models, or even plugins. This technique allows you to get up and running quickly without needing to hand-write a bunch of boilerplate every time you want to perform a common task. Plus, the less code you have to write, the fewer opportunities for bugs. Because Merb supports various testing frameworks and various ORMs, Merb’s generators will also customize the generated code based upon your settings.

For now, let’s generate the sandwich shop by running `merb-gen app sandwich-shop`.

Listing 2.1 Merb generator output

```
$ merb-gen app sandwich-shop
Generating with app generator:
     [ADDED]  tasks/merb.thor
     [ADDED]  .gitignore
     [ADDED]  public/.htaccess
     [ADDED]  tasks/doc.thor
     [ADDED]  public/javascripts/jquery.js
     [ADDED]  doc/rdoc/generators/merb_generator.rb
     [ADDED]  doc/rdoc/generators/template/merb/api_grease.js
     [ADDED]  doc/rdoc/generators/template/merb/index.html.erb
     [ADDED]  doc/rdoc/generators/template/merb/merb.css
     [ADDED]  doc/rdoc/generators/template/merb/merb.rb
     [ADDED]  doc/rdoc/generators/template/merb/merb_doc_styles.css
     [ADDED]  doc/rdoc/generators/template/merb/prototype.js
     [ADDED]  public/favicon.ico
     [ADDED]  public/images/merb.jpg
     [ADDED]  public/merb.fcgi
     [ADDED]  public/robots.txt
     [ADDED]  gems
     [ADDED]  Rakefile
     [ADDED]  app/controllers/application.rb
     [ADDED]  app/controllers/exceptions.rb
     [ADDED]  app/helpers/global_helpers.rb
     [ADDED]  app/models/user.rb
     [ADDED]  app/views/exceptions/not_acceptable.html.erb
     [ADDED]  app/views/exceptions/not_found.html.erb
     [ADDED]  autotest/discover.rb
     [ADDED]  autotest/merb.rb
     [ADDED]  autotest/merb_rspec.rb
     [ADDED]  config/database.yml
     [ADDED]  config/dependencies.rb
     [ADDED]  config/environments/development.rb
     [ADDED]  config/environments/production.rb
     [ADDED]  config/environments/rake.rb
     [ADDED]  config/environments/staging.rb
     [ADDED]  config/environments/test.rb
     [ADDED]  config/init.rb
     [ADDED]  config/rack.rb
     [ADDED]  config/router.rb
     [ADDED]  public/javascripts/application.js
     [ADDED]  public/stylesheets/master.css
     [ADDED]  merb/merb-auth/setup.rb
     [ADDED]  merb/merb-auth/strategies.rb
     [ADDED]  merb/session/session.rb
     [ADDED]  spec
     [ADDED]  app/views/layout/application.html.erb
```

Merb has generated created an application skeleton for you, including controllers, helpers, views, configuration, autotest configuration, and basic asset folders. It also creates testing stubs and a `Rakefile` that will provide you with the default Merb rake tasks.

Inside your app, you can use merb-gen to generate code for controllers, models, parts, and resources. You can see all of the available generators from any context by typing `merb-gen`. To find out more about a specific generator, use the `merb-gen` command. For instance, to learn more about the controller generator, type `merb-gen controller`.

Listing 2.2 Merb generator help

```
$ merb-gen controller
Creates a basic Merb controller.

USAGE: controller my_controller
    -S, --[no-]spec                  Generate with RSpec
    -T, --[no-]test                  Generate with Test::Unit
General Options:
    -h, --help                       Show this help message and quit.
    -p, --pretend                    Run but do not make any changes.
    -f, --force                      Overwrite files that already exist.
    -s, --skip                       Skip files that already exist.
    -q, --quiet                      Suppress normal output.
    -t, --backtrace                  Debugging: show backtrace on errors.
    -c, --svn                        Modify files with subversion. (Note:
```

If you attempt to override existing generated code, merb-gen will ask you how to handle the conflict. Now, let’s take a look at all those files you generated.

2.3.2 *What does a Merb app look like?*

Running merb-gen created all of the files you’ll need to get up and running. Figure 2.1 contains a visual representation of the directory structure. Let’s take a look at some of the components. At the root level, you can see a series of directories and the Rakefile. You’ll spend most of your time inside of the `app` directory, which stores your app’s controllers and templates. If you use an ORM, your models will also live under the app directory.

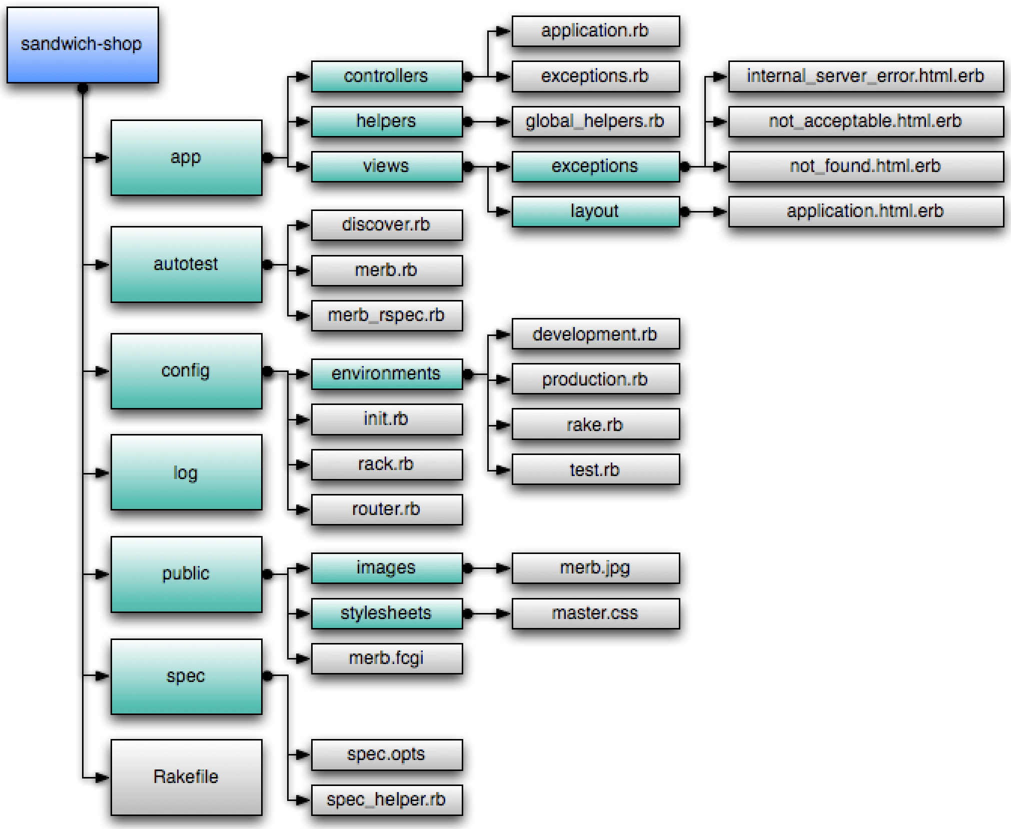


Figure 2.1 The Merb directory structure

Because of Merb's unique exception system, which allows you to treat Ruby exceptions as regular controller actions, Merb comes with a few exception templates under `app/views/exceptions`. It's worth taking a few minutes to look through these templates; these are the default error pages that will appear when Merb encounters an error. When you go to production, you probably want to create your own error templates to meet your specific needs.

In the beginning, you will spend a fair bit of time inside the `config` directory. Here, you can set up your application's configuration, including any special Rack handlers that you create to improve the efficiency of simple actions (we'll cover custom Rack handlers later in this book). You will also set up your router in `config/router.rb`. The router matches incoming URLs to controllers, actions, and parameters. You can also specify per-environment configuration in the `config/environments` directory.

Finally, if you want to modify the default directory locations for parts of Merb, you can create a `config/framework.rb`. You might use this feature, for instance, if you want to point Merb's models at an existing Rails application's models folder. But don't worry about that for now. The default directory structure should work just fine.

Let's look at some of the files that were created. First you'll notice that there is an `app` directory. This contains most of the code specific to your application. It's subdivided into `models`, `views`, `controllers`, and `helpers`.

Models contains the business logic for your application, and its connection to a data store, like a database. Views contain HTML or other templates. Controllers are what tie the two together and handle actual requests from web browsers. The goal is to keep as little logic as possible in the views, to make it easy to hand off your templates to a designer and to keep your application as maintainable as possible.

Finally, the `config` directory will hold various configuration details for your application. That's where we will store our database configuration, once we get there. There is also an `init.rb` that contains all of the configuration details for our application itself. Let's take a look at that file.

2.3.3 Configuring the application

The main file that you will use to configure Merb is init.rb. This file is loaded early in Merb's boot cycle, and will allow you to specify dependencies, choose an ORM, test framework, and more. The generated file comes complete with comments describing how to use the file. Let's walk through each line.

NOTE For a detailed look at the entire boot process, see chapter 7.

The first line in a generated init.rb is `require 'config/dependencies.rb'`. This loads the dependencies file, which lists the dependencies for your app. It would be fine to include the dependencies directly into init.rb, but this keeps them separate and maintainable.

Merb's dependencies are just simple plugins, but you'll use the `dependency` keyword so that the Merb framework can perform any necessary magic in the background. For instance, Merb will defer loading dependencies declared this way until the framework is fully loaded, to ensure that you're protected from load-order issues. The `dependency` method takes an optional second parameter, which allows you to declare a specific version requirement. It also takes an optional third parameter, which allows you to specify which file to require, if the name of the gem is not also the name of the file to require. For instance, you might say `dependency "merb-haml", ">= 0.9` if you wanted to include Merb's Haml support, and require at least version 0.9.

In the generated `dependencies.rb`, we declare a variable called `merb_gems_version`, which keeps all of the dependencies locked at the same version. Using the same version of all of Merb modules is a best practice; mixing and matching *might* work, but it also has its share of pitfalls.

NOTE Throughout this book, we make reference to Merb's load order. While knowing how Merb boots up can be useful, as a general rule you will not need to know the specifics in order to use Merb. Merb plugins will typically perform whatever steps necessary to insulate you from the details. In Merb's spirit of being hacker-friendly, however, we will provide the information as appropriate, in case you find it interesting or useful.

After loading in the dependencies, `init.rb` has a few lines where you can specify your ORM, test suite, and template engine. By default, Merb uses DataMapper, Rspec, and ERB. Specifying these options will ensure that the generator generates the proper code for your configuration. Listing 2.3 demonstrates how to select rspec and datamapper.

NOTE Chapter 4 contains much more information about testing your Merb applications.

Listing 2.3 Selecing a test suite and ORM

```
use_orm :datamapper
use_test :rspec
```

NOTE Choosing an ORM will also include the appropriate plugin for the ORM (for instance, choosing DataMapper will require the `merb_datamapper` plugin).

Next, `init.rb` has a small configuration block, which you can use to specify some of the many available configuration options. The default configuration options involve sessions, including which session store to use, and what the session secret key is. You can leave the secret key alone; Merb with automatically generate a safe key for you application.

Next up, you get a couple of ways to specify that dependencies load only after your app has completely loaded. You'll learn more about this in Part III, but there are occasions where you'll want to defer loading of certain files until very late in the boot process. An example of when you might use this feature is if the plugin requires access to your app's models. Listing 2.4 shows how you would load `merb_superplugin` after the app has loaded.

Listing 2.4 Loading dependencies late in the boot process

```
Merb.BootLoader.after_app_loads do
  dependency "merb_superplugin"
end
```

Next up, we'll handle setting up our choice of ORM and test suite. This will usually add rake tasks for the ORM, as well as causing the generators to provide ORM-specific code when appropriate.
With our Merb app all set up, let's put together our first controller.

2.4 Our first controller

In case you're not familiar with the terminology, a simple Merb page is split up into a controller and a view. In order to avoid mixing up a lot of code with your templates, controllers receive requests from the web browser, perform some setup, and then pass control to the templates (which we will call views). Figure 2.1 represents an extreme oversimplification of the roles controllers and views play in your Merb app.

NOTE See Chapter 9 for the inside details of Merb controllers.

First, the client asks your application to say "Hello". This is received by the controller, which stores "Hello" as the thing that should be said. It then passes control to the View, which renders some HTML, including the "Hello" that was set by the controller. Finally, it returns the HTML back to the server.

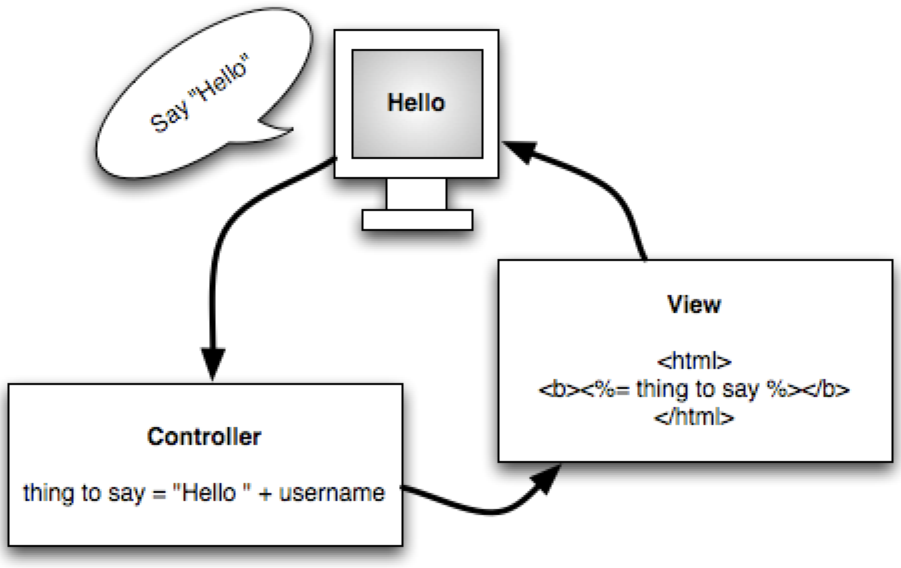

Say "Hello"
Hello
View
<html>
<b><%= thing to say %></b>
</html>
Controller
thing to say = "Hello " + username

Figure 2.1 Controllers and views

Because we have separated the responsibility of determining what should be said from its surrounding template, it is easy to pass off the templates to designers, who don't need to worry about the specifics of how we're going to determine what to say. If we decide to change "Hello" to "Howdy!" later on, we'll just need to modify it in the controller, without having to pick apart our templates to change the logic. Of course, this is an absurdly simple example, and it's easy to imagine storing the very simple controller logic directly in the view. But no real applications are this simple, and splitting up the responsibilities in this way will substantially improve your ability to maintain applications going forward.

Controllers also encapsulate logical units of your application. For instance, you can write helpers that apply to a single controller, and all of the views that you write for a single controller will be stored in a directory for that controller.

Now let's get back to the sandwich shop. We'll start by putting up a front-page for our website to welcome new visitors, tell them our hours, and have links to our menu. Let's start by generating a controller for our front page. This controller will have a handful of static pages, so we'll use the somewhat non-RESTful name "Static". Run `merb-gen controller static` and see what happens.

Listing 2.5 Generating a controller

```
$ merb-gen controller Static
Generating with controller generator:
     [ADDED]  app/controllers/static.rb
     [ADDED]  spec/controllers/static_spec.rb
     [ADDED]  app/views/static/index.html.haml
     [ADDED]  app/helpers/static_helper.rb

Don't forget to add request/controller tests first.
```

If you've used Rails before, this will look mostly familiar. Notice that Merb controllers don't have "Controller" after their name. The controller for pages is called `Pages`, and the controller for static stuff is called `Static`. You have to pay attention so you don't have naming conflicts, but this happens much less often than you'd expect.

You can see that our early choice of Rspec for our testing framework has caused Merb do generate specs for us. If we had chosen Test::Unit, we'd have gotten those instead. Isn't that handy?

Now let's add some code to the `Static` controller and make sure everything is working. Open the file `app/controllers/static.rb` and take a look at the `index()` method. Inside controllers, methods that get called via the web are also called actions, and that's the terminology we'll use from now on. The `index` action is what is typically used for the default action within a controller, and that's how our application is setup by default.

Listing 2.6 Default generated controller code

```
class Static < Application

  def index
    render
  end

end
```

The way Merb deals with response bodies is similar to Ruby: the return value of the action is what is eventually sent to the browser. We don't want to deal with templates yet, so for now let's just return a string. Change `index()` to return `"Welcome to the Merb Sandwich Shop!"` instead of calling `render()`.

Listing 2.7 Modified controller

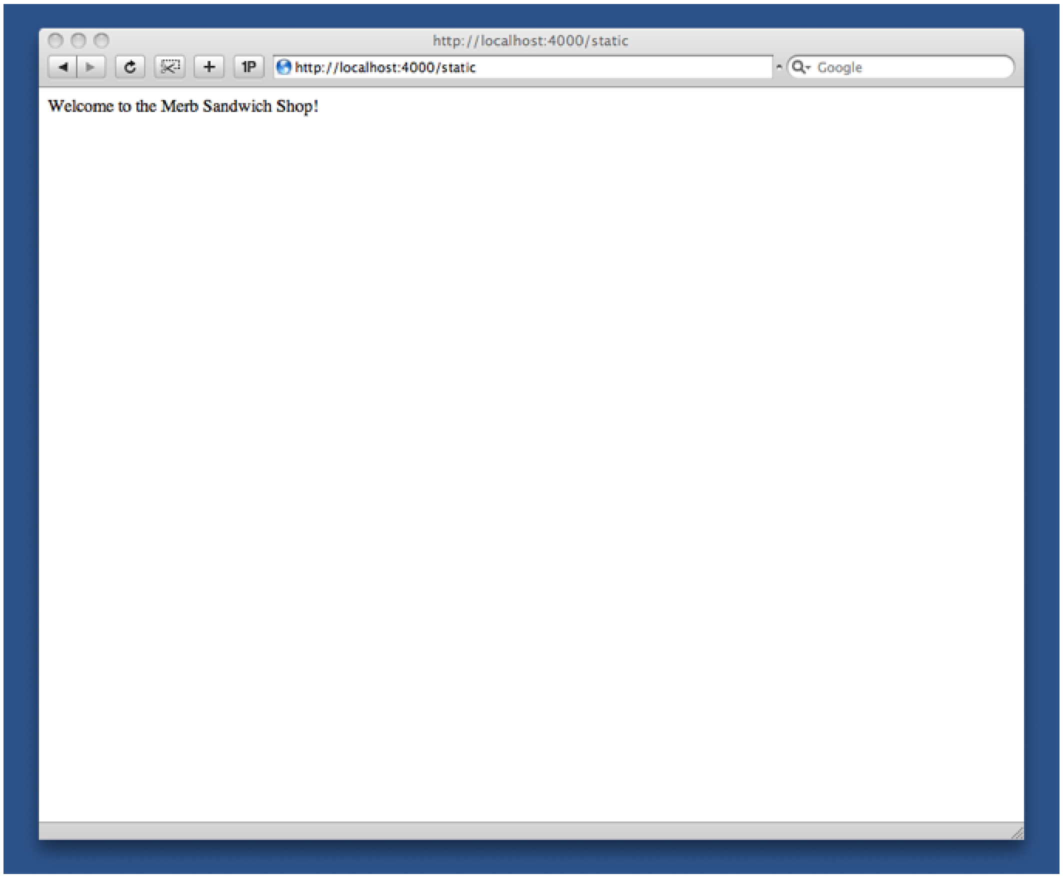
```
class Static < Application

  def index
    "Welcome to the Merb Sandwich Shop!"
  end

end
```

Before we do anything else, let’s run our application and take a look at it. From the application directory, type `merb` and you’ll see the app boot up.

Listing 2.8 Booting the app for the first time

```
$ merb
 ~ Compiling routes...
 ~ Using 'share-nothing' cookie sessions (4kb limit per client)
 ~ Using Mongrel adapter
```

Now let’s point a web browser to http://localhost:4000/static and take a look.



Welcome to the Merb Sandwich Shop!

Figure 2.1 It's alive!

That was pretty cool, right? Certainly not the flashiest web app intro ever, but Merb is more about substance than style.

2.4.1 *Making the URL nicer*

It would be much nicer to use http://localhost:4000 (and later http://merbsandwichshop.com) without having to add "/static" to the URL. The Router handles the mapping of incoming URLs to controllers and actions. By default, Merb loads router configuration from `config/router.rb`, so let's edit that file and add a default route.

Listing 2.9 Front page route

```
Merb::Router.prepare do |r|
  #....

  match('/').
    to(:controller => 'static', :action => 'index').
    name(:front)

  #....
end
```

We've broken the route across three lines to get a better idea of what's going on. The first part of the route definition is a call to `match()` with the URL pattern. In this case, we want to match just the front page, so we use "/". Merb's router is very powerful, so we'll see how to match much more complex routes later on.

Next we specify which controller and action will handle this route using `to()`. Again, we'll see much more powerful uses of this later on. Finally, we call `name()` to set the name of this route. You can use the name of the route later on in your views to generate URLs.

Now if you load http://localhost:4000/ in your browser, you should see the welcome message.

2.5 Making our site look good

While it's exciting that we go the site to load, it sure is boring. Let's make it look a little nicer. Merb supports supports several template languages out of the box, including Erubis and Haml. We'll use Erubis for this project, but you should look into Haml. It takes a little getting used to, but it's worth it.

2.5.1 Rendering templates

Instead of just returning a string, we'll have our index action call `render()`, one of two common ways to access Merb's rendering system (the other is `display`, which we'll see in the next chapter). In the default framework configuration, Merb looks for templates in a subdirectory of `app/views/` named for the controller. Since we're in the Static controller, we need to create our file in `app/views/static/`. The name of our new template should be `index.html.erb`. Merb uses a common naming pattern of `template_name.format.template_language`, as shown in Figure 2.1.

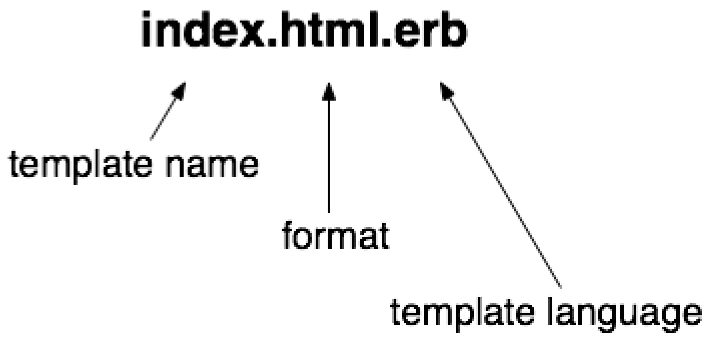


Figure 2.1 Template naming

Calling render by itself will look for the template for this controller and action and wrap it in the application's layout. It then returns the results of rendering as a String, so it's not any different from returning a String directly.

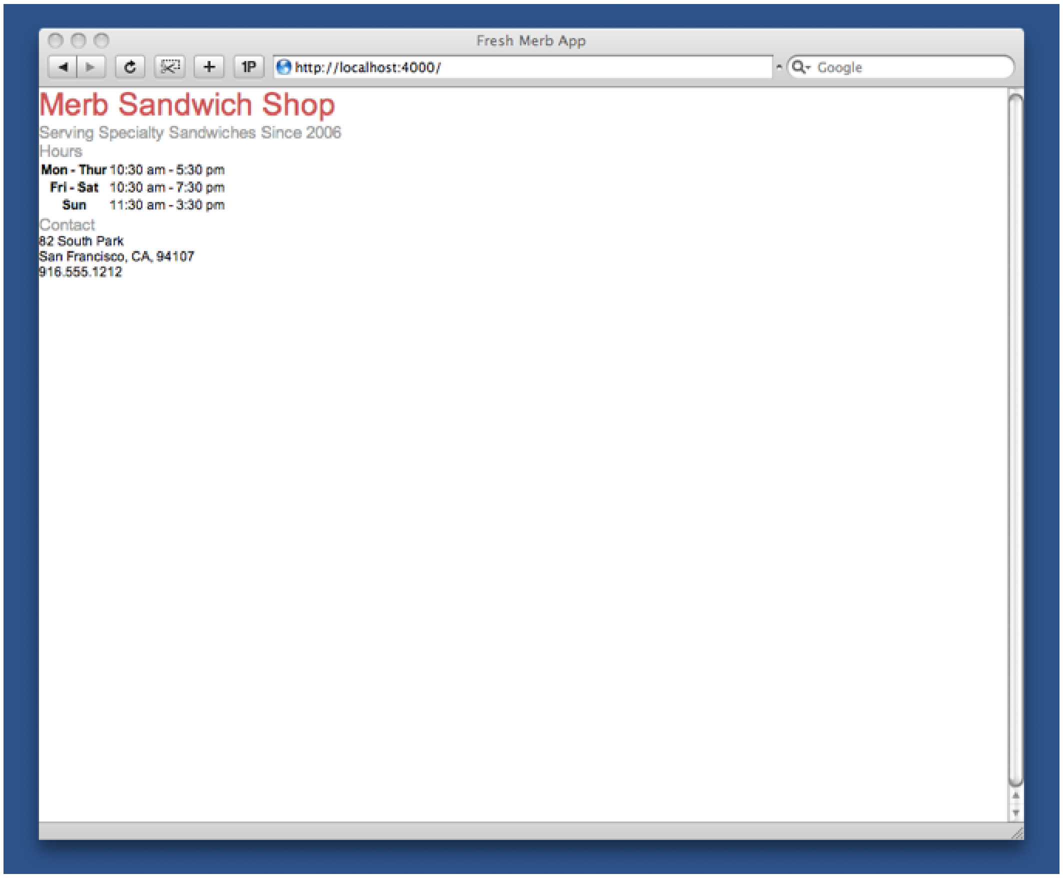
NOTE One of the elegant things about Merb is its use of standard Ruby constructs over magic. This is one such case. By having `render`, `redirect`, and `display` return Strings, we make it very simple to track what is happening in a Merb action. In contrast, Ruby on Rails doesn't require a specific return value for actions, which leads to more complexity in trying to identify how the developer intended to specify what to render.

The generators have already created this file for us, so let's replace the contents with some information about our shop.

Listing 2.10 app/views/static/index.html.erb

```
<h1>Merb Sandwich Shop</h1>
<h2>Serving Specialty Sandwiches Since 2006</h2>

<h2>Hours</h2>
<table>
  <tr>
    <th>Mon - Thur</th>
    <td>10:30 am - 5:30 pm</td>
  </tr>
  <tr>
    <th>Fri - Sat</th>
    <td>10:30 am - 7:30 pm</td>
  </tr>
  <tr>
    <th>Sun</th>
    <td>11:30 am - 3:30 pm</td>
  </tr>
</table>

<h2>Contact</h2>
<div class="vcard">
 <div class="adr">
  <div class="street-address">82 South Park</div>
  <span class="locality">San Francisco</span>,
  <span class="region">CA</span>,
  <span class="postal-code">94107</span>
 </div>
 <div class="tel">916.555.1212</div>
</div>
```

We also need to modify our controller to call `render` with no arguments. Merb will figure out the rest: because the controller is Static, and the action is index, and the format is html (more on alternate formats in Chapter 3), look for any template named `app/views/static/index.html.*` and render it. Let's see if it worked.



Merb Sandwich Shop
Serving Specialty Sandwiches Since 2006
Hours

Mon - Thur	10:30 am - 5:30 pm
Fri - Sat	10:30 am - 7:30 pm
Sun	11:30 am - 3:30 pm

Contact
82 South Park
San Francisco, CA, 94107
916.555.1212

Figure 2.1 Using render

That looks much better! Notice that Merb even styled it with a default set of CSS rules. Thanks, Merb.

NOTE There's lots more information on the render process in chapter 9

The `erb` in the template name stands for Erubis, a pure-Ruby implementation of ERB, or embedded Ruby. For a tutorial on Erubis, visit http://www.kuwata-lab.com/erubis/users-guide.html, but for now the important things to know are that `<%= ruby_code %>` evaluates the ruby code and outputs the result, and `<% ruby_code %>` evaluates the code, but doesn't output anything.

We've seen how to render templates, but most modern apps have a single layout that wraps each page's content. Merb handles this problem elegantly.

2.5.2 Layouts

Most sites have some standard content that goes on every page, like the header and navigation links, and the copyright notice at the bottom. As you would expect, Merb has a way to wrap every page inside a shared layout that contains these common elements. These layout files are stored in `app/views/layouts` and follow the same naming convention as other templates. By default, the main application layout is called `application.content_type.template_language`.

The generators have already created one for us, in fact. That's how we got the page title "Fresh Merb App" and the CSS included, in the previous section. The default layout name is `application.html.erb`. Let's look at it now.

Listing 2.11 application.html.erb

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
      <html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en-us" lang="en
  <head>
    <title>Fresh Merb App</title>
    <meta http-equiv="content-type" content="text/html; charset=utf-8" />
    <link rel="stylesheet" href="/stylesheets/master.css" type="text/css"
       media="screen" charset="utf-8" />
  </head>
  <body>
    <%= catch_content :for_layout %>
  </body>
</html>
```

This looks like a pretty standard HTML file, except for the single ERB tag with `catch_content :for_layout`. Merb uses `throw_content` and `catch_content` to pass chunks of content up from templates and partials to higher templates. The content chunk named `:for_layout` is used by `render`. You won't need to worry about the details now, but if you're interested take a look at Chapter 9: Routers and Rendering.

For now, let's change the title to "Merb Sandwich Shop", and add a minimal header.

Listing 2.12 Modified layout

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
      <html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en-us" lang="en
  <head>
    <title>Merb Sandwich Shop</title>
    <meta http-equiv="content-type" content="text/html; charset=utf-8" />
    <link rel="stylesheet" href="/stylesheets/master.css" type="text/css"
       media="screen" charset="utf-8" />
  </head>
  <body>
    <h1 id="header">Merb Sandwich Shop</h1>
    <%= catch_content :for_layout %>
  </body>
</html>
```

We can use any of the features supported by our template language and the rendering system inside layouts, including partials. Now that we have our website up and running, let's list the various sandwiches we serve. While we could use another static template for this, it will be much easier if we store this information in a database, so we can manage it later.

2.6 Setting up the database

As we mentioned before, Merb is database-agnostic, so you can choose ActiveRecord, Sequel, DataMapper, or any other ORM that has a Merb plugin. Many Merb developers choose DataMapper, and that's what we will use for this application. Before we can create some sandwiches, we need to install DataMapper and setup our database.

2.6.1 Installing DataMapper

The Merb framework is ORM agnostic, which means that you can use Merb with ActiveRecord (which comes bundled with Rails), DataMapper, and Sequel. The Merb core team recommends DataMapper, because it is threadsafe, more architecturally sound, and can be used easily with other storage types like Salesforce, FaceBook, CouchDB or REST-style HTTP. It also comes standard with a number of features that will make your database code more efficient.

The first thing you'll want to do is decide what database you want to use. We'll use Sqlite3, which comes preinstalled on OSX and many flavors of Unix. You can check whether it's installed by typing `sqlite3` on the command line, which will bring you to a prompt marked `sqlite>` if it is installed. If not, install it via your package manager, or if you're on Windows, by downloading the binary from the SQLite website.

Your initial Merb installation will have already installed DataMapper. Now, you'll need to install the database driver that allows DataMapper to connect to sqlite3. You can do that via `gem install do_sqlite3`.

Merb's generator has already set up a `database.yml` file which defaults to SQLite and a database called `sample_development.db`.

Now that we have our database set up, let's set up our sandwich menu!

2.6.2 Creating a resource

As you might imagine, we're going to need a lot of files to set up our sandwich menu. We're going to need to describe our table of sandwiches in our model. But we're also going to need to generate a web page to view those sandwiches. A lot of the code to set this up is boilerplate, so Merb comes with a generator that will generate the required code.

In case you're wondering, a Merb Resource is simply a way to interact with a database model over HTTP. The generated code provides facilities for interacting with the model through basic web forms, as well as over HTTP via a REST-style architecture. You can learn more about REST, which stands for Representational State Transfer, in the original paper that coined the term http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm. For the purposes of Merb, all you need to know is that resources expose your models via HTTP.

We can generate a model, controller, and a set of default views all with one command by using the resource generator. Merb supports the RESTful conventions developed by Rails, and following them can lead to very simple and clean code. Let's build a resource for our Sandwiches.

Listing 2.13 Generating a resource

```
$ merb-gen resource sandwich name:string,description:text,price:big_decima
     [ADDED]  spec/models/sandwich_spec.rb
     [ADDED]  app/models/sandwich.rb
     [ADDED]  spec/requests/sandwiches_spec.rb
     [ADDED]  app/views/sandwiches/show.html.haml
     [ADDED]  app/views/sandwiches/index.html.haml
     [ADDED]  app/views/sandwiches/edit.html.haml
     [ADDED]  app/views/sandwiches/new.html.haml
     [ADDED]  app/controllers/sandwiches.rb
     [ADDED]  app/helpers/sandwiches_helper.rb
resources :sandwiches route added to config/router.rb
```

That one command did a lot of work, so let's step through it.

The first argument, `resource` is the name of the generator to call. The second argument, `sandwich`, is the name of our new resource. It will create a model called `Sandwich` and a controller called `Sandwiches`, in keeping with the RESTful idioms established by Ruby on Rails. The rest of the arguments are fields to store in our new resource, which describe the name of the field, and the type of the field. For instance, the database table will have `name` column, which will be a `string`. Let's take a quick look at the model that Merb generated for us.

Listing 2.14 Sandwich model

```
class Sandwich
  include DataMapper::Resource

  property :id, Serial

  property :name, String
  property :description, Text
  property :price, BigDecimal
end
```

Merb created a stub model, with the properties set up as we described in the generator. To push those definitions into the database, simply run rake `db:automigrate`, which will create the sandwiches table for you to use in your application. If you pop into sqlite3 and take a look at the schema, you'll see that the database is all set up and ready to go.

Listing 2.15 And it works!

```
$ rake db:automigrate
$ sqlite3 sample_development.db
SQLite version 3.4.0
Enter ".help" for instructions
sqlite> .schema
CREATE TABLE "sandwiches" ("id" INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT
  "name" VARCHAR(50), "price" DECIMAL(10,0), "description" TEXT);
CREATE TABLE "sessions" ("session_id" VARCHAR(32) NOT NULL,
  "data" TEXT DEFAULT 'BAh7AA== ', "created_at" DATETIME, PRIMARY KEY("ses
CREATE TABLE "users" ("id" INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,
  "login" VARCHAR(50), "crypted_password" VARCHAR(50), "salt" VARCHAR(50))
```

Merb and DataMapper took care of syncing up your code with your databases. It also created a couple of tables for sessions and users, which come with the merb stack. And it works!

NOTE The automigrate command will wipe all existing data, and rebuild the models from scratch. This is a great way to get up and running, but not a very good way to modify your data in production. For changes to production data, use Classic Migrations, which we will discuss in the next chapter.

Looking back, what's interesting is that Merb automatically handled creating a DataMapper model and rspec test stubs. If you had chosen to use the ActiveRecord ORM and `Test::Unit`, Merb would have created the appropriate files for that configuration. (You can learn how this works, and how to use it for other ORMs in chapter 13.) Your brand new controllers won't do anything unless you tell Merb how to route to them. Let's see how.

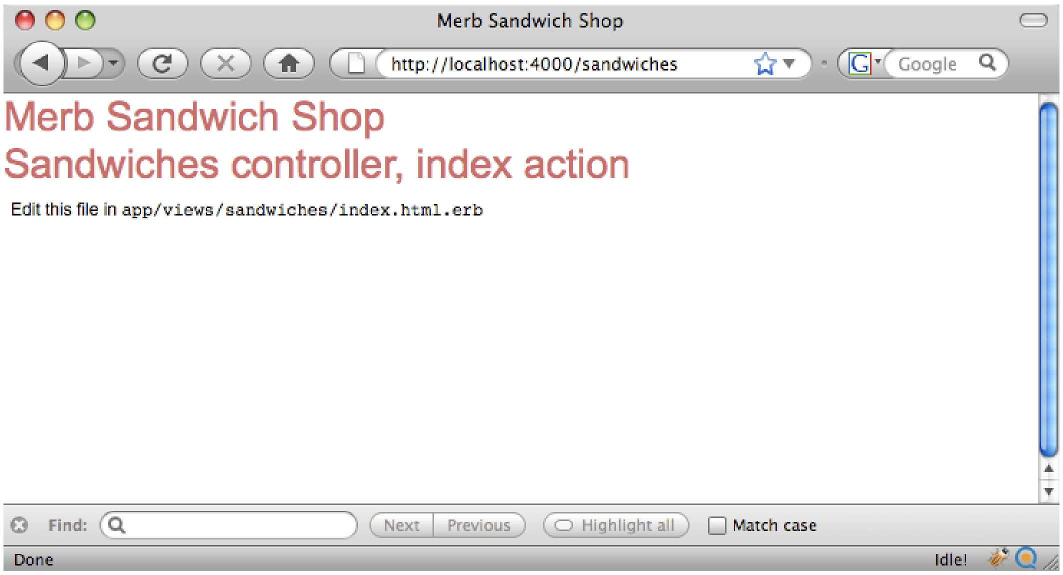
2.6.3 RESTful routes

We only need one more step before we can see this new resource in action: we need to map it to an incoming URL. You might be wondering why we can't just use the default route. Before we go into what that means, let's see it in action. Merb has automatically added the sandwiches route, as in Listing 2.16.

Listing 2.16 Sandwiches route

```
Merb::Router.prepare do |r|
  r.resources :sandwiches

  #....
end
```

Start the application again with `merb` and point your browser to http://localhost:4000/sandwiches. You will see the default page that just provides some basic details about where you are in the app.



Merb Sandwich Shop

Sandwiches controller, index action

Edit this file in app/views/sandwiches/index.html.erb

Figure 2.1 Index template

In the next few chapters, we'll learn how to fill this, and the page at http://localhost:4000/sandwiches/new so you can create your menu. If you want to be adventurous, or know a bit about Rails, you might want to see how far you can get setting up this administrative portion of the sandwich shop. After chapter 4, when you're more familiar with Merb, we'll remind you to come back and finish the Sandwich Shop application.

As an exercise, try seeing what other URLs are available, and think about what they might be used for. In the coming chapters, we'll show you how to use all the RESTful URLs that Merb comes with to build a solid, maintainable application. Before we move on to the next chapter, let's take a brief detour to discuss how Merb handles the issue of backwards compatibility.

2.7 A note on backward compatibility

A core principle of Merb is the differentiation between our public and private API. The public API is the documentation at api.merbivore.com. This includes all methods that you will need to know about to build a Merb application. Of course, there are many more methods inside of Merb itself that are used to help Merb do its work. However, those pieces of functionality are considered private and should not be modified or overridden. As a result, new releases of Merb focus heavily on maintaining backward-compatibility with the previous release's public API, but may heavily refactor the internals of the framework.

Because the Merb core team so strongly objects to using the private API in a Merb application, they consider it a bug if you find that you need to in order to achieve something in your application. Feel free to file a ticket on the Merb bug tracker with information about your experience and the team will try to expose new public functionality that meets your needs or explain how to use existing public functionality.

The upside of this is that you only need to know about a small portion of the code in the framework to get your work done. Since Merb is already a fairly small framework, you'll probably be able to keep much of the public API in your head while working on applications, improving your ability to make use of all of Merb's features. You can think of the public API as a contract between the Merb team and you as a developer: "We will not make changes to these pieces of functionality without a major announcement and an entry in the Public API changelog". This will make it very easy to upgrade from one release of Merb to another, as you'll have a small, readable list of changes between the last release and the current release that affect you as an app developer.

2.8 Summary

We just blazed through installing Merb, getting up and running with some simple code generation, and putting together our first pages. Don't worry if it seemed a bit daunting. In the coming chapters, we'll take a more in-depth look at many of the pieces we skimmed over in this chapter, helping you learn about the power behind what we have achieved so quickly here.

If you're looking for a challenge before moving on, try creating your own simple Merb app for something you're passionate about. See if you can replicate the steps we performed above to create a simple static application for your action figure collection (or, who knows, beanie baby collection?). Next up, we'll talk about using Merb with a database, and fill out those resource stubs that we saw in this chapter.

3

Getting Started With a Database

Since you'll be using a database with your application, you'll need a way to get data from the database into your application. As a Ruby programmer, you'd like to avoid writing raw SQL commands and manually paging through result sets. Instead, you'd prefer to write Ruby, and have a Ruby program automatically handle database access. There are several Ruby libraries capable of interacting with the database for you, each with a unique set of pros and cons.

These programs are called Object Relational Mappers (ORMs), and they provide a Ruby interface in front of your database access. For instance, you will be able to get back a list of all menu items as an Array, iterate over the Array, access the properties over individual items, and pretty much any normal Ruby operation. In essence, the purpose of an ORM is to hide the fact that you're working with a database, and to make code accessing your database seamlessly integrate with your other Ruby code.

NOTE One downside of ORMs, in general, is that they hide away differences between Ruby and database access that may cause you to access the database very inefficiently. For instance, looping over an Array of database objects and accessing an associated object may produce a database hit for each object looped over. This is only true in less sophisticated ORMs, like ActiveRecord, but the general caution applies even to very sophisticated ones.

There are a number of ORMs available for Ruby. Perhaps the most well-known is ActiveRecord, included in Ruby on Rails. Merb has very good support for ActiveRecord (commonly abbreviated as AR), but works well with other options, too. Merb also supports Sequel, an ORM that runs "closer to the metal" with SQL than other Ruby ORMs. Many of you will be interested in Merb because of its association with DataMapper (DM), a very sophisticated ORM that Merb also works exceptionally well with. How do you decide which is the right ORM to choose? If you're using the `merb` package, you've already gotten DataMapper, but let's look at all of them more closely.

3.1 How do I know which ORM to use?

The three ORMs natively supported by Merb have benefits and drawbacks. Let's take a look at what using the various libraries will mean to you.

3.1.1 ActiveRecord

To start, you could use ActiveRecord. Because of its inclusion with Ruby on Rails, it's ubiquitous in the Ruby ecosystem. Almost any Rails programmer will already know how to use it, and have a toolkit of extensions to improve its functionality. It's also extremely simple to use. However, there are a number of extensions for ActiveRecord that only currently exist for Ruby on Rails. Additionally, because ActiveRecord is such a simple ORM, there are a number of features commonly found in ORMs that are simply not present in ActiveRecord.

For instance, ActiveRecord makes a number of assumptions that make it very difficult to use multiple databases at a time. It does not use database-specific optimizations, like prepared statements. It also only supports SQL databases, like PostgreSQL or MySQL, so the convenience of its API cannot easily be translated to working with web-service-based APIs, like Salesforce or Facebook.

ActiveRecord was written as part of Ruby on Rails, and extracted out Basecamp by 37Signals. As a result, it is extremely capable for simple database-driven web applications. On the other hand, a lot of the assumptions of those sorts of apps are baked into ActiveRecord (for instance, it's non-trivial to get AR to connect to multiple databases conveniently). Sometimes, you'll need something less opinionated.

3.1.2 Sequel

The second Merb-supported option is the Sequel ORM. Sequel is a wrapper on top of SQL databases, allowing you to succinctly express SQL queries in Ruby. For instance, you could perform the following query:

```
DB[:zoos].join(:animals, :zoo_id => 1).first
```

Which would compile into:

```
SELECT * FROM zoos INNER JOIN animals ON animals.zoo_id = id;
```

One nice feature of Sequel is that it will delay actually making a query until you ask for objects in the return set, so you can build up a query in Ruby, and have it execute efficiently one time on the database.

Sequel is an excellent choice if you plan to spend a lot of time thinking in terms of SQL queries, as it provides a thin Ruby wrapper around those queries. It abstracts away the different syntax of the various different database servers, like MySQL, Postgres, SQLite3, and Oracle, so you can write Ruby code and be sure that the SQL generated will work correctly with your database.

Sequel also has experimental support for more traditional ORM concepts like models, which map database tables to Ruby classes. However, model support is not core to how Sequel works, as it focuses more heavily on the SQL wrapper. In addition, the models can only be used to back relational databases that support SQL. If you want a mapper that doesn’t make any assumptions about the back-end data store, you’ll need something even more sophisticated.

3.1.3 DataMapper

The third, most popular ORM used with Merb, is called DataMapper. Unlike ActiveRecord and Sequel, DataMapper abstracts away the idea of a database entirely, allowing you to use it with many different kinds of data, including Salesforce, IMAP, or even raw files. It also provides a powerful multiple-data-source paradigm, allowing you to reuse the same class across multiple data stores. Like Sequel, DataMapper also defers queries as late as possible, and only retrieves data when it is needed. Probably the most powerful concept is that DataMapper provides a true identity map: two queries that return the same piece of data will return the exact same object (this is illustrated in Figure 3.1).

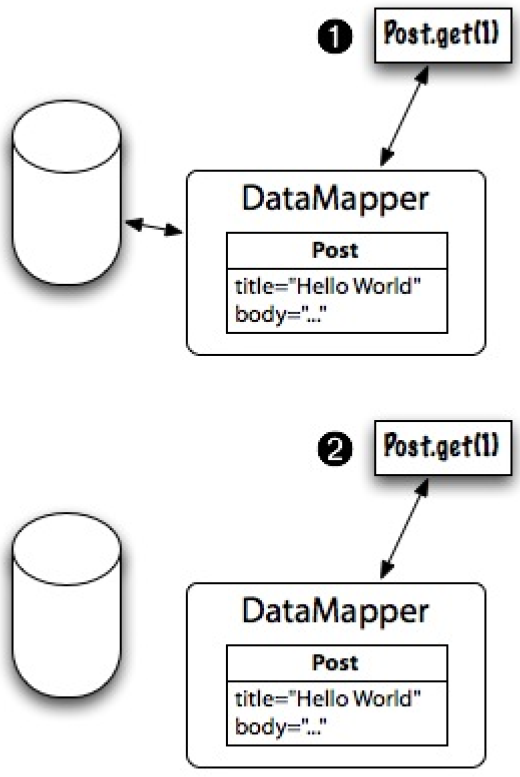


Figure 3.1 DataMapper's identity map ensures that database rows are always represented as a single object

These features make it ideal for applications that utilize multiple data sources, including data from web services that can be represented via a data-store paradigm, which serves to unify the interface that Merb developers will use for interacting with various forms of data. As a result, most members of the Merb core team, and most early adopters of Merb, have opted to use DataMapper to work with data in both databases and via web services.

NOTE A lot of the ideas DataMapper uses are common in ORMs in other languages, like .NET and Java. As a result, it is significantly more powerful than either ActiveRecord or Sequel, but learning about all the features, and when to use them, is admittedly more difficult than fully understand ActiveRecord.

Because of the strong, early enthusiasm for the Merb/DataMapper combination, you will find more community support and existing code written around it. If you're starting a new application today with Merb, the Merb core team strongly encourages using DataMapper.

Now that you know what ORMs are available, you're probably wondering how different using them with Merb will be.

3.2 Using Merb With ORMs

Because Merb is not tightly coupled to any of the above ORMs, it does not currently attempt to abstract away things like database access into a single Merb API. In other words, you will need to know how the ORM you're using works, and switching from ActiveRecord to DataMapper will require modifying the code that you use to access data.

Accessing data in ActiveRecord is shown in Listing 3.1.

Listing 3.1 Getting access to a record with ActiveRecord in Merb

```
class Records
  def show(id)
    @record = Record.find(id)
    display @record
  end
end
```

DataMapper has a subtly different API, shown in Listing 3.2.

Listing 3.2 Getting access to a record with DataMapper in Merb

```
class Records
  def show(id)
    @record = Record.get!(id)
    display @record
  end
end
```

This may seem like a lot of code that doesn't mean anything, but that's OK. The important thing to notice is that ActiveRecord based applications will use 'find()' while DataMapper based applications will use 'get!()'. There are other small differences in the API of the various ORMs that you'll have to be aware of, especially if you want to run the code samples in this book, but with something other than DataMapper.

However, Merb's generators do attempt to provide a single mechanism for creating new models and resources. As we saw in Chapter 2, new resources are generated with the following syntax:

```
 $ merb-gen resource sandwich name:string,description:text,price:big_decim
```

Even though ActiveRecord, DataMapper, and Sequel provide very different ways to create the sandwiches table in the database, creating a new resource in Merb is accomplished the same way across all ORMs.

NOTE Merb's maintainers also plan to provide an abstraction layer that plugins can use to implement data-store functionality without having to explicitly provide support for ActiveRecord, DataMapper, and Sequel. Instead, the ORM adapters would be responsible for exposing common functionality to Merb applications. This could, for instance, be used to implement data-store based authentication and authorization.

For the most part, Merb has abstracted away the differences between using various ORMs. For instance, generating a model in ActiveRecord uses the same syntax as generating a model in DataMapper or Sequel.

However, since ActiveRecord uses migrations to manage the database schema, and DataMapper does not, what you'll need to do after generating your model will differ. In ActiveRecord, you would run 'rake db:migrate', while in DataMapper, you'll run 'rake db:automigrate'. How exactly ActiveRecord works is out of the scope of this book. However, there are a number of books that can walk you through beginner, intermediate, and advanced use of ActiveRecord.

There is no current book that provides this information about DataMapper. Since many developers will be interested in using DataMapper with Merb, and because we've chosen to use DataMapper as the primary ORM throughout the rest of this book, we've included a general introduction to DataMapper, as well as more advanced usage hints throughout the book.

If you installed `merb`, you already have DataMapper installed. But what if you want to use ActiveRecord or Sequel? You shouldn't be surprised to learn that the installation process is not much more difficult.

3.3 Installing an ORM

Installing an ORM is usually a painless process, which entails installing a Ruby gem or two onto your system, and adding some configuration information to your Merb app. If you're using `merb` and want to follow along as we finish the sandwich shop, feel free to skip this section.

3.3.1 Installing ActiveRecord

The first step in installing ActiveRecord is to install the ActiveRecord gem:

```
$ gem install active_record
```

You also need the `merb_activerecord` gem, which will give Merb the information it needs to interact with ActiveRecord, including providing the typical rake tasks for migrations and support for testing ActiveRecord models.

```
$ gem install merb_activerecord
```

NOTE Installing ActiveRecord will also install ActiveResource, Rails' library of extensions to the Ruby programming languages. This is a fairly large dependency that can account for around 10MB of RAM, and makes a number ofinvasive changes to the way Ruby handles loading classes. That's one of the reasons many Merb developers have stayed away from using ActiveRecord as the ORM with Merb.

Once ActiveRecord is installed, you'll want to go into config/init.rb in your application, and tell Merb to use it. If you run `rake -T db`, you should see a list of rake tasks for ActiveRecord, which should be identical (or at least similar) to the output in Listing 3.3.

Listing 3.3 Telling Merb about your use of ActiveRecord will add useful rake tasks to your app

```
# config/init.rb
use_orm :activerecord

# output
$ rake -T db
rake db:abort_if_pending_migrations
rake db:charset
rake db:collation
rake db:create
rake db:create:all
rake db:drop
rake db:drop:all
rake db:fixtures:load
rake db:migrate
rake db:migrate:redo
rake db:migrate:reset
rake db:reset
rake db:rollback
rake db:schema:dump
rake db:schema:load
rake db:sessions:clear
rake db:structure:dump
rake db:test:clone
rake db:test:clone_structure
rake db:test:prepare
rake db:test:purge
rake db:version
```

As you can see, the Rails database tasks are now available to your Merb app. From the command line, type `merb` to start the server. This will generate a sample `database.yml` file in `config/database.yml.sample` which you should rename to `database.yml` and provide your connection details. For more information on how to configure your database with ActiveRecord, pick up any of the numerous Ruby on Rails or ActiveRecord books.

3.3.2 Installing Sequel

3.3.3 Installing DataMapper

If you installed `merb`, you already have DataMapper installed. If you installed `merb-core`, without the rest of the stack, enter the following commands the commands in Listing 3.4.

Listing 3.4 Installing DataMapper

```
gem install datamapper
gem install merb_datamapper
```

You will also need the database driver for whichever database you want to use. DataMapper uses the DataObjects drivers to connect to the database, so you will need to install `do_mysql`, `do_sqlite3`, or `do_postgres`. Drivers for Oracle and MS SQL are in the works. Feel free to visit datamapper.org for more details.

What is DataObjects?

Ruby has a number of extremely inconsistent database drivers used to connect to the various database servers. Each of these drivers has its own mechanism for creating a connection, passing queries to the database, receiving back results, and looking at the results.

The DataObjects project rewrote the drivers from the ground up, in C, to provide a single interface for accessing databases, regardless of the underlying database semantics.

Unlike ActiveRecord and Sequel, therefore, DataMapper does not need to contain shim code for each database driver, making it more efficient. Figure 3.1 shows how DataMapper interacts with the database through DataObjects.

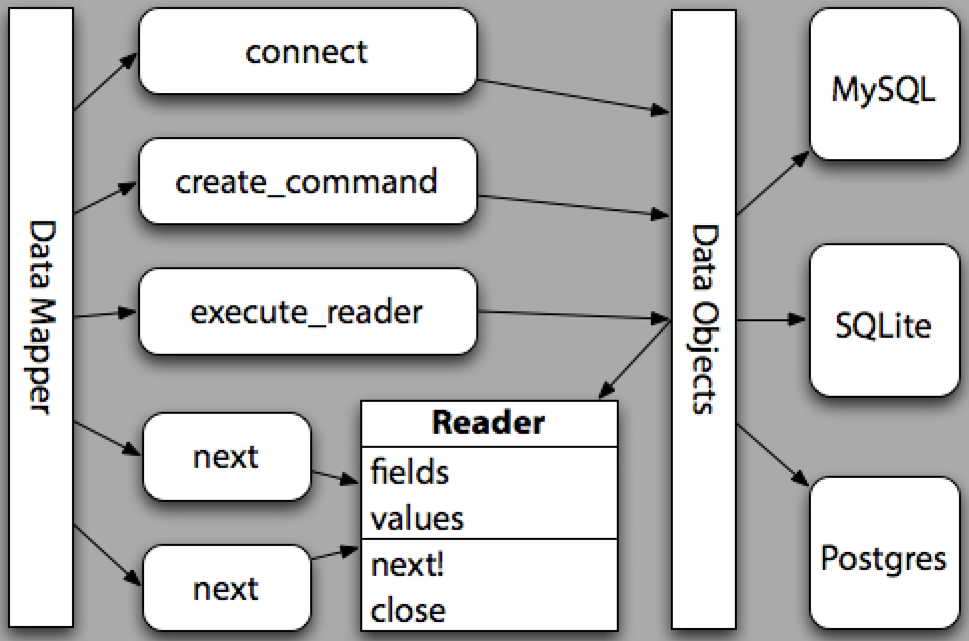


Figure 3.1 Data Objects simplifies the interaction with the database

DataMapper uses a decidedly different strategy for getting your database to look like your Ruby code. Unlike ActiveRecord, DataMapper models provide a mapping for each property in your database that you wish to map. For instance, a simple user model might look like Listing 3.5.

Listing 3.5 A simple User model using the DataMapper ORM

```
class User
  include DataMapper::Resource

  property :id, Serial
  property :name, String
  property :password, String
end
```

Because you have provided the schema for the table in your model, DataMapper can set up the database to match this schema without requiring special migration files. To set up your database for the first time, run `rake db:automigrate`. This will create a new table called `users` with an integer, autoincrementing column called `id` and two `varchar(50)` columns called `name` and `password`.

While automigration is great for getting a new database up and running, it doesn't work as well for upgrading a live schema, in production. For production schemas, DataMapper provides two additional strategies: autoupgrade and classic migrations.

3.3.4 Automatically migrating less dangerously

Automigrations work by completely trashing any existing table and rebuilding it from scratch. This works great for development, where you don't have any mission-critical data in your database. For production, though, alarm bells are probably going off in your head. We're going to need a different strategy.

The first alternate strategy is called autoupgrade, which will attempt to modify the existing schema to match your models, but will never destroy data. For instance, if you add a new column, it will add it to the database without deleting your data. Much better. Unfortunately, autoupgrade has its drawbacks. For one, it can't handle column renaming, or, at the moment, changes to the properties of a column (for instance, adding a default to a column is not yet supported).

For robust control over live, production data, we're going to need something different.

3.3.5 Classic migrations

For modifying live data, you're probably going to want simple, repeatable, testable, and controllable migrations. DataMapper provides support for migrations, similar to ActiveRecord migrations.

3.3.6 When to use the available strategies

Because Rails is so popular, it has popularized a number of common idioms for working with migrations. Many projects have a series of migrations that new developers run, starting from migration number 1, to get their databases up and running. But projects also commonly use migrations to add bootstrapping data, or perform data manipulations that are heavily reliant on the state of the Ruby codebase when those manipulations occur.

As a result, it can become very difficult to keep migrations working for new developers who need to start from migration 1.

Because of these problems, most DataMapper developers do not use migrations as an all-purpose database bludgeon. For getting a new system in sync with the current database, whether it's a development or production database, use automigrations. Since there's no data yet, you won't risk losing any. This makes it very simple to get new developers up and running, as they'll just need to run rake db:automigrate.

For bootstrap data, use a single Ruby script that creates the data, and keep it up to date against the current codebase. Since its purpose is always to bootstrap new machines, you won't have to worry about messing with past migrations.

Now, migrations serve purely to painlessly and cleanly update a running schema in production. Once a migration is run, it becomes a historical relic, never to be used again, because any new boxes will leapfrog over all of the previous migrations by running automigrations.

As for autoupgrade, it is mostly a curiosity at the moment. Virtually nobody wants to run it in production, for fear of unpredictable results. However, the DataMapper team is planning to allow the results of autoupgrade to be serialized into a classic migration, which would allow all the convenience of automigrate, and all of the stability and control of classic migrations.

With all that meandering, you're probably wondering when we were going to get around to finishing the sandwich shop. Your mission, should you choose to accept it, is to follow us into the next section, where we'll put all this knowledge into use.

3.4 Integrating DataMapper with the sandwich shop

When we left off, we had generated a static version of the sandwich shop, and started playing with connections to the database. Let's think about what our database should look like. For each sandwich, we'll probably want to store the name of the sandwich, a description, and a price. We might also want another database table where we could store a record every time someone purchases a sandwich.

Because sandwich prices might change, we'll probably want to store both a link to the sandwich purchased, as well as the price at time of purchase in that record. In other words, we'll want something like Figure 3.1.

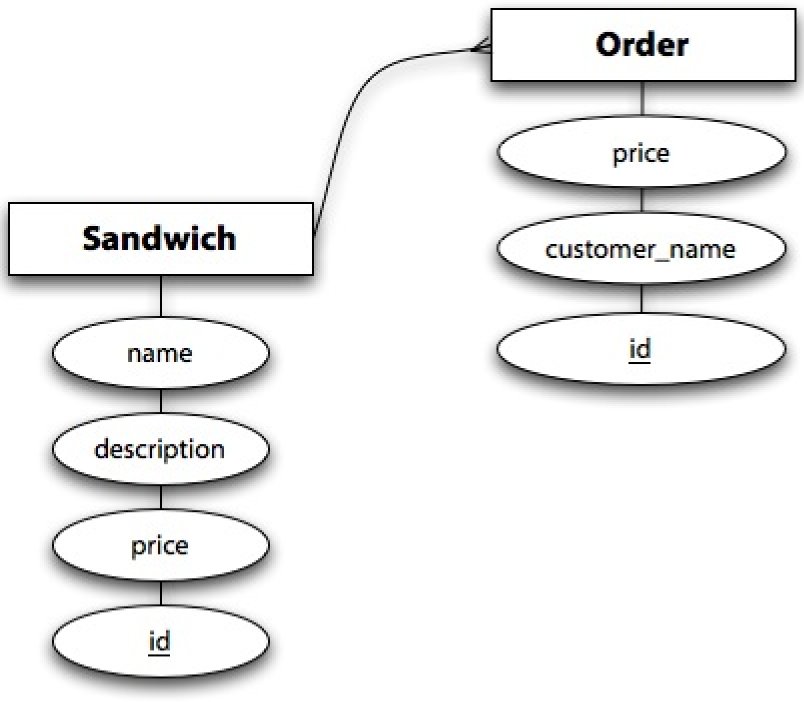


Figure 3.1 In the database, we'll store the sandwiches, as well as orders for those sandwiches.

Converting that to DataMapper models is easy. Listing 3.6 shows what your DataMapper classes would look like.

Listing 3.6 The sandwiches and orders, as DataMapper models

```
# app/models/sandwich.rb
class Sandwich
  include DataMapper::Resource

  property :id, Serial
  property :name, String
  property :description, Text
  property :price, BigDecimal

  has n, :orders
end

# app/models/order.rb
class Order
  include DataMapper::Resource

  property :id, Serial
  property :customer_name, String
  property :price, BigDecimal

  belongs_to :sandwich
end
```

In order to create those models, we could use the model generator:

```
merb-gen model order customer_name:string,price:big_decimal
```

Note that we already generated the `sandwich` model in chapter 2 when we generated the `sandwich` resource. This will create the models in Figure 3.1, but we'll need to add in the `has n` and `belongs_to` associations. Open `sandwich.rb` and `order.rb` and add those in.

Before we continue, let's set up DataMapper to use SQLite. First, go to `config/environments/development.rb`. At the end of the file, add the contents of Listing 3.7

Listing 3.7 Setting up SQLite with DataMapper

```
use_orm :datamapper do
  DataMapper.setup(:default, "sqlite3:config/dev.db")
end
```

Now, do the same in `config/environments/test.rb`, replacing `dev.db` with `test.db`.

Now that Merb knows where to find out database, let's automigrate! Run `rake db:automigrate`. Voila! Your `config/dev.db` will now contain the schema. Pretty cool, no? To convince yourself that it actually worked, feel free to drop into the Sqlite console (Listing 3.8) and take a look.

Listing 3.8 The SQLite console shows that automigrating worked!

```
$ sqlite3 config/dev.db
sqlite> .schema
CREATE TABLE "orders" ("id" INTEGER NOT NULL PRIMARY KEY \CC\
AUTOINCREMENT, "customer_name" VARCHAR(50), "price" \CC\
DECIMAL(10,0), "sandwich_id" INTEGER);
CREATE TABLE "sandwiches" ("id" INTEGER NOT NULL PRIMARY \CC\
KEY AUTOINCREMENT, "name" VARCHAR(50), \CC\
"description" TEXT, "price" DECIMAL(10,0));
```

Since we're making a web app and not a command-line app, we'll probably want some way to interact with our new store through the web. Let's take a look at how controllers look in Merb, and how to use Ruby to route incoming requests to controllers and actions.

3.4.1 Creating a Controller

Once again, Merb provides a generator that will create some stub controllers and views. While the controllers are very useful, and probably won't need to be changed all that much, the generated views are simply stubs that you should replace very early on.

In Chapter 2, we generated these files by running `merb-gen resource sandwich name:string,description:text,price:big_decimal`. If we wanted to generate the controller files for a model that already existed, we could run `merb-gen resource_controller sandwich`.

Generating a resource generated a bunch of controller and view files. But what do they do? The name of each file, and its purpose, is shown in Table 3.1.

Table 3.1 The purpose of each of the generated files.

File	Purpose
`app/controllers/sandwiches.rb`	The sandwiches controller
`app/views/sandwiches/index.html.erb`	The view that will show a list of sandwiches
`app/views/sandwiches/show.html.erb`	The view that will show a single sandwich
`app/views/sandwiches/edit.html.erb`	The view that will allow editing a sandwich
`app/views/sandwiches/index.html.erb`	The view that will allow you to create a new sandwich
`app/helpers/sandwiches_helper.rb`	Where we'll put helper methods relating to our sandwiches controller
`spec/requests/sandwiches_spec.rb`	The place we'll spec our sandwiches controller

Another thing that the generator did for us is to create a route for the resource. Like the router in Ruby on Rals, Merb's router allows you to specify, in Ruby, where in our application to send inbound requests. In the case of resource controllers, which provide a front-end for a DataMapper model, Merb provides a one-line shortcut, shown in Listing 3.9.

Listing 3.9 Add a route for the `sandwiches` resource

```
Merb::Router.prepare do
  resource :sandwiches             # 1
  default_routes
end
```

1. Our sandwiches route

That is pretty simple, but what the heck does it mean? That resource route unpacks into the routes shown in Listing 3.10.

Listing 3.10 What the single resource route looks like to Merb

```
Merb::Router.prepare do
  with(:controller => "sandwiches").match("/sandwiches") do
    match("(.:format)").name(:sandwiches)                        # 1
    match("(.:format)", :method => :post).
      to(:action => "create").name(:create_sandwich)             # 2
    match("/new(.:format)").
      to(:action => "new").name(:new_sandwiches)                 # 3
    match("/:id(.:format)").
      to(:action => "edit").name(:edit_sandwich)                 # 4
    match("/:id(.:format)", :method => :put).
      to(:action => "update").name(:update_sandwich)             # 5
    match("/:id(.:format)", :method => :delete).
      to(:action => "destroy").name(:destroy_sandwich)           # 6
  end
end
```

1. GET /sandwiches routes to Sandwiches#index
2. POST /sandwiches routes to Sandwiches#create
3. GET /sandwiches/new routes to Sandwiches#new
4. GET /sandwiches/1 routes to Sandwiches#edit with params[:id] = 1
5. PUT /sandwiches/1 routes to Sandwiches#update with params[:id] = 1
6. DELETE /sandwiches/1 routes to Sandwiches#delete with params[:id] = 1

If your head's spinning, you're not alone. Merb's router is very powerful, and resource routes merely hide away a common idiom so you don't have to write all of the above routes every time you want to use a resource.

3.4.2 Writing your own routes

We'll go into more detail about how to use Merb's powerful router in a later chapter, but for now, let's take a look at the basic syntax of a Merb route.
The first part of the route is the *matcher*, which is matched against incoming URLs. This can be either a string or regular expression, but we'll typically use the String form. A matcher can contain one or more segments, separated by `/`s or `.`s. Figure 3.1 shows how Merb matches a normal, string-based route.

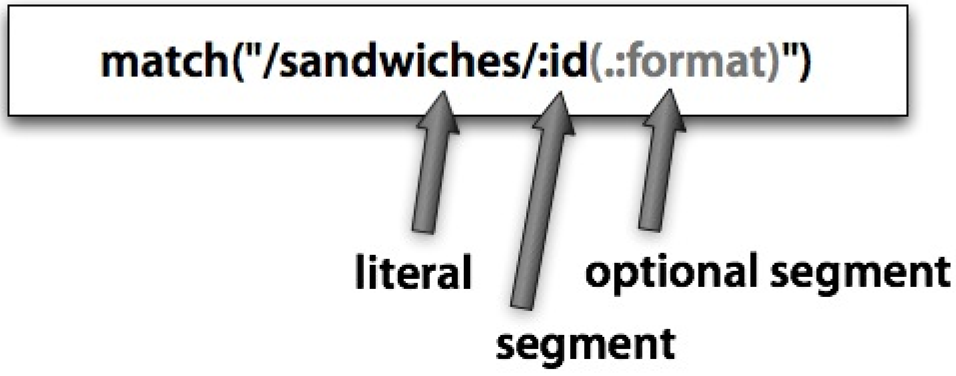


Figure 3.1 Matching parts of a typical route.

Each segment can be literal (such as `sandwiches` above) or parameter. Parameter segments begin with `:`s and are added to the list of parameters that come in through the query string or POST body.

For example, match("/:action/:id") will match "/foo/bar", and the parameters available in your controller will be {:action => "foo", :id => "bar"}.

Second, segments can be required or optional. Optional segments are surrounded by parentheses, and can even be nested. For instance, you can say `match("/:action(/:id(.:format))")`, which makes `/:id.:format` optional, and even if `/:id` is provided, `.:format` would still be optional.

In addition, the matcher can contain rules to match any part of the request object. In the example above, we used :method => :post. Internally, Merb looks at request.method and checks whether it is, in fact, :post. This could even be used to match things like SSL, by doing :ssl? => true. Figure 3.1 shows how Merb matches information from the inbound request object. For more information on what is contained in the Merb::Request object, consult the Merb documentation.

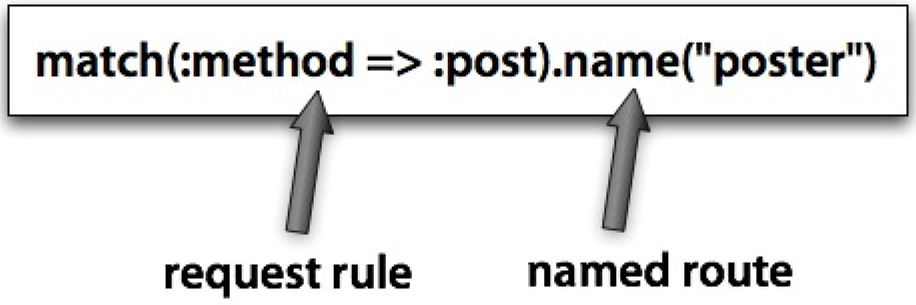


Figure 3.1 Merb has support for matching any part of the route.

Finally, routes can be named. This allows you to easily generate the URL from a controller or view. For instance, to generate a URL for editing a sandwich called @sandwich, you can call `url(:edit_sandwich, @sandwich)`. In general, if you are writing your own routes, you will want to name them. This is because Merb's router is so flexible that it would be cost-prohibitive to generate a URL by scanning through all of your routes and finding the least ambiguous match.
Phew! That was a lot of information, but it helped us to understand what the resource shortcut is doing. Don't worry if you're still a bit lost on some of the details; we'll do a full treatment on the router in a future chapter. For now, it's enough to recognize that we have provided our users with a way to get to every action in the `Sandwiches` controller.

NOTE In writing this book, we could have opted to preserve the whiz-bang magic act for a bit longer. However, in Merb's spirit of less magic and more explicitness, we chose to pull the curtain back and show you what's going on as early as possible.

Finally, we'll need to modify the templates that Merb generated for us to allow us to show and edit our sandwiches. Let's start with the index.

3.4.3 Listing the Sandwiches

By default, Merb creates a very simple template, shown in Listing 3.11.

Listing 3.11 The generated `index` template

```
<h1>Sandwiches controller, index action</h1>

<p>Edit this file in <tt>app/views/sandwiches/index.html.erb</tt></p>
<p>For more information and examples of CRUD views read
  <a href="http://wiki.merbivore.com/howto/crud_view_example_with_merb_usi
    this wiki page
  </a>
</p>
```

1. Generated header
2. Generated text (to delete)

As you can see, this generates template is not designed to be used as is. First, let’s replace the text in the `<h1>` (#1) with `Sandwiches to Buy`. Next, delete the second line. Finally, let’s add a table to list our sandwiches.

We’ll probably want to include the name of the sandwich in the index, as well as the price. Add `<th>Name</th>` and `<th>Price</th>` to `<thead>` of the table..

Next, the template iterates over all of the available sandwiches. At the moment, all it does it provide links to show, edit, or delete the sandwich. We’re going to want to include the name and price as well. Before looking at Listing 3.12, which shows the final template, try to do this yourself.

Listing 3.12 The final sandwiches list

```
<h1>Sandwiches to Buy</h1>

<table>
  <thead>
    <tr>
      <td>Name</td>
      <td>Price</td>
    </tr>
  </thead>

<tbody>
<% @sandwiches.each do |sandwich| %>
  <tr>
    <td><%= sandwich.name %></td>
    <td><%= "$%.2f" % sandwich.price %></td>
    <td><%= link_to 'Show', resource(sandwich) %></td>               # 1
    <td><%= link_to 'Edit', resource(sandwich, :edit) %></td>        # 2
    <td><%= delete_button sandwich %></td>
  </tr>
<% end %>
</tbody>
</table>

<%= link_to 'New', resource(:sandwiches, :new) %>
```

NOTE In showing the price, we used Ruby's built-in formatting tools, which are very similar to printf or sprintf in other languages. In this case, we're simply specifying that we want the price to appear as a decimal (`f` stands for floating point), with two decimal places. This will guarantee that 2, 2.0, and 2.00 will all display as "$2.00". Of course, it would be easy to write a helper to do the formatting, but we wanted to show how easy it was to do with pure-Ruby.

At #1, and a few other times in the completed index, we use a new helper called `resource`. This helper allows you to generate an URL for a given resource, based upon how you set up the resource in your router. In this case, `resource(sandwich)` will generate `/sandwiches/<id>`. At #2, `resource(sandwich, :edit)` will generate `/sandwiches/<id>/edit`, which the router will send to the edit action in the Sandwiches controller.

You can think of the `resource` helper as the counterpart of the `resources` router method. While the router method describes how inbound URLs should be dispatched, the helper creates URLs for a given object.

Okay, so we have a list of sandwiches. What if we want to actually look at one of them?

3.4.4 Displaying a Sandwich

On the index page, Merb provided a link to `Show` the sandwich. That link pointed to url(:sandwich, sandwich). As we discussed earlier when we covered the router, that URL will point to the Sandwiches controller, and the `show` action. Before we look at the template, let's see what the `show` action looks like in Listing 3.13.

Listing 3.13 The `show` action, as generated by Merb

```
def show(id)
  @sandwich = Sandwich.get(id)                          # 1
  raise NotFound unless @sandwich                       # 2
  display @sandwich                                     # 3
end
```

The action is pretty straight forward. First, we get the sandwich, based on the `id` that was passed in. If no sandwich was found, we raise a NotFound error, which will be returned to the client as a 404 error. Finally, if a sandwich was found, we display it.

Using the same code for XML, JSON, and HTML

When writing web services that can also be accessed via the web, a common idiom is to render a template for HTML but to send back a JSON or XML representation when a web service is expecting that.

Merb encapsulates that idiom in the `display` method. First, `display` looks for a template for the current content-type. Typically, a template would be available for the HTML content-type, but plenty of people use templates for XML as well (using the Haml or Builder template engines). If no template is found, Merb will attempt to call `to_xml` or `to_json` on the object passed into `display`.

The transforming method is registered when the mime-type is registered, so this will typically work out of the box. For instance, during bootup, Merb registers the JSON mime type by:

```
Merb.add_mime_type(:json, :to_json, \CC\
%w[application/json text/x-json], :charset => "utf-8")
```

If you want to create your own mime type that works with display, use `Merb.add_mime_type` in your `init.rb` to add a mime of your own.

In the case of HTML, which we're dealing with at the moment, `display @sandwich` is identical to `render`, since DataMapper objects do not respond to `to_html`.

Assuming a valid sandwich was passed in, we're going to be whisked away to the `show` template, which is displayed in Listing 3.14.

Listing 3.14 The template for the `show` action, as generated by Merb

```
<h1>Sandwiches controller, edit action</h1>

<p>Edit this file in <tt>app/views/sandwiches/edit.html.erb</tt></p>
<p>For more information and examples of CRUD views read
  <a href="http://wiki.merbivore.com/howto/crud_view_example_with_merb_usi
    this wiki page
  </a>
</p>
```

Once again, Merb has provided us with a very simple template. The first thing to do is to replace the contents of the `<h1>` with something like `Sandwich: <%= @sandwich.name %>`. This will display the sandwich's name as the header. We also want to delete the `<p>` on the second line, and add some information to the template. Again, try to solve this problem before peeking at Listing 3.15.

Listing 3.15 The final template for the `show` action

```
<h1>Sandwich: <%= @sandwich.name %></h1>

<p><%= @sandwich.description %></p>
<p>Price: <%= "$%.2f" % @sandwich.price %></p>

<%= link_to 'Edit',  resource(@sandwich, :edit) %> |
<%= link_to 'Back', resource(:sandwiches) %>
```

If you were making a real application, you'd probably have a picture, and some other fancy details. Perhaps you'd even include how popular each sandwich was on its `show` page. If you're looking for a challenge, try extending this page to include those details after we add the order page later.

We're making good progress now. We can see all the sandwiches, but we have no way to add any. The index and show pages are going to be lonely with no sandwiches on them. Let's remedy the situation.

3.4.5 Creating a New Sandwich

The index page provides a link to `url(:new_sandwich)`, which points to our Sandwiches controller and the `new` action. Because web services can POST directly to create a new sandwich, this action is only required for HTML.

Listing 3.16 The `new` action, as generated by Merb

```
def new
  only_provides :html                     # 1
  @sandwich = Sandwich.new                # 2
  display @sandwich                       # 3
end
```

1. Don't provide JSON or XML
2. Make a new sandwich
3. Display it

The first line of the method tells Merb to override the default mime type support for the class and provide only HTML. The second line creates a new sandwich, and the third line displays it. Because this method provides only HTML, `display @sandwich` is functionally identical to `render`. We use `display` to make it simpler to add support for other mime types (like :iphone) if the need should arise in the future.

The template for the `new` action is very simple.

Listing 3.17 The template for the `new` action, as generated by Merb

```
<h1>Sandwiches controller, new action</h1>

<p>Edit this file in <tt>app/views/sandwiches/new.html.erb</tt></p>
<p>
  For more information and examples of CRUD views read
  <a href="http://wiki.merbivore.com/howto/crud_view_example_with_merb_usi
    this wiki page
  </a>
</p>
```

As we did before, the first step is to change the `<h1>` to something more descriptive, and remove the placeholder `<p>`. We'll title this page `Make a new Sandwich`. We'll want to replace the placeholder with an actual form, using Merb's form helpers.

NOTE If you're not using the full stack, you can install the merb helpers via `gem install merb-helpers`. You will also need to add `dependency 'merb-helpers'` to your init.rb.

Listing 3.18 shows the final template.

Listing 3.18 The working template for the `new` action

```
<h1>Make a new Sandwich</h1>

<%= form_for(@sandwich, :action => url(:sandwiches) ) do %>
  <p><%= text_field :name %></p>
  <p><%= text_area :description %></p>
  <p><%= text_field :price %></p>
  <p><%= submit "Create" %></p>
<% end =%>

<%= link_to 'Back', url(:sandwiches) %>
```

NOTE If you're familiar with Rails, you'll notice a slight difference in the format of the form helper. Instead of including the begin and end of the helpers in `<%` and `%>`, Merb requires that the start block be surrounded by `<%= %>`, like any other text-emitting helper, and that the end block by surrounded by `<% =%>`, to denote the end of a unit of code. This makes writing block-accepting helpers easier and more efficient, and also makes it more clear that the entire block of code is placing a string into the output.

With what we have so far, we can open our sandwich shop for business. But what if we decide to change the price of our Pumpernickel Supreme, or add swiss cheese to our Turkey Club?

3.4.6 Editing a Sandwich

Looking at the `show` template, we can see that Merb has provided a link to edit the sandwich, which points to the `Sandwiches` controller and the `edit` action. Let's take a look at that action.

Listing 3.19

```
def edit(id)
  only_provides :html
  @sandwich = Sandwich.get(id)
  raise NotFound unless @sandwich
  display @sandwich
end
```

Like the `new` form, this action only provides HTML, because web services can PUT directly to the resource without first needing to go through the edit page.

Next, we grab the sandwich matching the ID passed in. If no sandwich was found, we raise a NotFound error, which is sent back to the browser as a 404. Otherwise, we render the template. Let's take a look at the default template.

Listing 3.20

```
<h1>Sandwiches controller, edit action</h1>

<p>Edit this file in <tt>app/views/sandwiches/edit.html.erb</tt></p>
<p>
  For more information and examples of CRUD views read
  <a href="http://wiki.merbivore.com/howto/crud_view_example_with_merb_usi
    this wiki page
  </a>
</p>
```

We could do the same thing as we did in the last section and add the form inline. However, it would look identical to the form we created in the previous section. This is a perfect opportunity to use a partial, which allows us to extract out common bits of template code into a single template that is referenced from each template that uses it.

Partials are stored in the same directory as the templates themselves, with a leading underscore. For instance, we would call the form partial `_form.html.erb`. Extracting out the common parts from the `new` template, the form partial would look like Listing 3.21.

Listing 3.21 The form to be used in the other templates

```
<p><%= text_field :name %></p>
<p><%= text_area :description %></p>
<p><%= text_field :price %></p>
<p><%= submit "Create" %></p>
```

The `new` template would become Listing 3.22.

Listing 3.22 The `new` template using the form partial

```
<h1>Make a new Sandwich</h1>

<%= form_for(@sandwich, :action => url(:sandwiches) ) do %>
  <%= partial :form %>
<% end =%>

<%= link_to 'Back', url(:sandwiches) %>
```

As you can imagine, updating the `edit` form is now trivial. The results are shown in Listing 3.23.

Listing 3.23 The `edit` template using the form partial

```
<h1>Editing <%= @sandwich.name %></h1>

<%= form_for(@sandwich, :action => resource(@sandwich)) do %>
  <%= partial :form %>
<% end =%>

<%= link_to 'Show', resource(@sandwich) %> |
<%= link_to 'Back', url(:sandwiches) %>
```

We now have a tasty administrative interface, where you can go in and create sandwiches to your heart's content. But what about purchasing sandwiches. After all, what good is a listing of the yummy goods unless people can actually order things.

As an exercise, try creating a front-end to the sandwich store. It should allow users to buy any quantity of sandwiches, and allow them to check out and buy them when they're done. Don't worry about authentication or actually charging credit cards. Assume that your customers will come in to your shop and pay in cash.

You will probably want to set up a simple shopping cart, which you should manage using sessions, which are extremely easy to use. In your controller and views, you will have access to a `session` object, which effectively serves as a Hash of values. Anything that you put into the session hash will be available on future pages when accessed by the same user. In other words, each user has their own private bag that you can throw data into.

One solution to creating the shopping cart is available with the code that can be downloaded from this book's website at http://manning.com/ivey.

Once you're ready to launch Sandwich Shop (beta, of course), follow me into the next section, where we'll discuss how to get the best performance out of DataMapper.

3.5 Squeezing performance out

DataMapper is designed to be a very efficient ORM, and uses various tricks to avoid making large numbers of requests to the database. Specifically, DataMapper can be used concurrently using threads, and will cache certain kinds of queries. But you can't efficiently leverage those features without understanding them. This section will give you a somewhat detailed view on some of the ways DataMapper can help with performance, and how to be sure you're making use of those features.

3.5.1 *Leveraging thread safety*

Both Merb and DataMapper are `threadsafe`. If you don't know what that means, don't worry. The authors of this book still have trouble wrapping their heads around the concepts sometimes. Essentially, thread-safe code can be run in parallel, while single-threaded code can only be run one request at a time.

To help illustrate, consider the very simple, trivial code in Listing 3.24:

Listing 3.24 A simple name printer

```
class NamePrinter
  attr_accessor :name

  def print_name
    puts self.name
  end
end

NAME_PRINTER = NamePrinter.new
```

Now imagine that you had some code using the `NAME_PRINTER` that set the name, and then printed it. It might look something like Listing 3.25.

Listing 3.25 Using the NamePrinter class in another method

```
def print_name(name)
  NAME_PRINTER.name = name
  NAME_PRINTER.print_name
end
```

Of course, you would never write code like this; it simply illustrates a point about thread-safety. If you attempted to call the `print_name` method simultaneously, what would likely happen is that the name printer's name value would get set to some value, but then another invocation of `print_name` would print, causing conflicts.
The reason this happens is that the `print_name` method is using a "shared resource", so its internal state can be changed by another simultaneously running method (in another thread) without our knowledge. There are two ways to solve this problem. The first way is to tell Ruby not to allow two threads to run the `print_name` method simultaneously. Ruby provides a facility called a "mutex", which handles this. You can see a solution using a mutex in Listing 3.26.

Listing 3.26 A solution to the concurrency problem using a `Mutex`

```
MUTEX = Mutex.new
def print_name(name)
  MUTEX.sychronize do
    NAME_PRINTER.name = name
    NAME_PRINTER.print_name
  end
end
```

This will prevent a second thread from changing the name of the `NAME_PRINTER` before the first thread has had a chance to print it. This does solve the problem, but it significantly reduces the effectiveness of threading, because instead of executing in parallel, threads are forced to wait on other threads to finish running.

In the context of Merb, this means that if a second request comes into Merb before the first one has completed, it would need to wait for the first request to finish before being allowed to run. The other solution is not to use any shared state. A solution involving no shared state might look something like Listing 3.27.

Listing 3.27 A solution to the concurrency problem by avoiding shared state

```
def print_name(name)
  name_printer = NamePrinter.new
  name_printer.name = name
  name_printer.print_name
end
```

Because each invocation of `print_name` has its own name printer, other threads cannot change the name before this thread has had a chance to complete. The two invocations can run in parallel. Both Merb and DataMapper use no shared state, allowing multiple requests to be run in parallel. However, this means that you need to be careful, in your own code, not to rely on state in a single request that could potentially be changed by another request.

In general, this means not using class variables or global variables to pass information from object to object. Since each request has its own copy of the controller, using instance variables in a controller is perfectly safe.

NOTE Ruby on Rails and ActiveRecord, however, heavily use a mutex-based solution to prevent so-called race conditions, which can occur with shared-state. By default, ActiveRecord is not threadsafe, storing information about the current scope of certain finders in global variables. ActiveRecord does have a threadsafe mode, but it requires the allocation of a new database connection for each thread, which is prohibitively expensive.

If you use Merb with DataMapper, Merb will not use a mutex around the invocation of your actions. However, if you choose to write code with shared state (which is strongly recommended against), you can turn on the mutex in the configuration block in `config/init.rb` by adding `c[:use_mutex] = true`.

3.5.2 The nefarious *n* + 1 problem

Assume for a moment that our sandwich shop is chock full of sandwiches, and we've had thousands of orders. You decide that you'd like to know how popular each sandwich is, so you write an action and a view to display a list of the sandwiches ordered. Later, you might decide to aggregate them, and show you the most popular ones, but to give you quick and dirty information, you're happy with just seeing a list of all of the orders.

Your code would probably have something like `Order.all.map {|o| [o.sandwich.name, o.price] }`, which would return back a list of the name and price of each order. In naïve ORMs, that will typically cause Ruby to first pull a list of all of the orders. As it loops through the orders, those ORMs will ask the database for the associated sandwich as it comes to it, pulling out from the association and the price from the order. That can be extremely inefficient, potentially leading to thousands of database queries each time you want to see the list of ordered products.

If you're using ActiveRecord, the solution is to do something like `Order.find(:all, :include => :sandwich)`. This will pull down all of the associated sandwiches at once, on the first query. This works fine if everything's together, but frequently you'll create the list of orders in your controller or a helper, and iterate over them in a view. You'll need to keep track of all of the consumers of your order list, and make sure to include the necessary associations.

DataMapper solves this problem with a technique called Strategic Eager Loading. When you first call `Order.all`, DataMapper pull down only the list of orders from the database, but it remembers that they were all collected together. Later, if you ask for the sandwich of any of the orders, DataMapper will retrieve the sandwiches for *all* of the items in the loaded set. This optimization alone makes using DataMapper much less prone to accidentally making thousands of queries for a single page.

3.5.3 The Identity Map

Another common issue with ORMs is that objects are pulled out of the database, but if the ORM retrieves the same row in the database, it creates a new object. This can lead to the very perplexing `Order.first != Order.first`, unless the ORM overrides == to tell Ruby that two different objects are in fact the same one.

This can get worse if one of the two objects is updated and saved, while the other object retains its original state. DataMapper solves this problem by using an Identity

Map, which stores a cache of all of the loaded objects, keyed by their primary key. Before creating a new object, DataMapper first checks to see whether a copy of it exists in memory. If so, it simply returns the in-memory copy instead of creating a new one.

In some cases, this can result in fewer queries to the database. For instance, DataMapper will not repeat the `Order.get(1)` query; once it has Order #1 in memory, it doesn't try to retrieve it again. However, the bigger win is that two queries that produce overlapping groups of objects will actually share some of the same Ruby objects.

DataMapper keeps an identity map while inside what it calls a "repository scope". By default, Merb's DataMapper plugin will wrap a repository scope around invoking an action, so you be sure that each object in the database is represented by a single Ruby object for the duration of the action.

3.5.4 Improving performance with cached views

Another potential optimization involves when DataMapper actually invokes a query. If you write `Sandwich.all`, DataMapper creates a new Query object that represents the query that will eventually be sent to the database, but does not yet send it. When you iterate over the Array, or ask for an member of it, DataMapper will kick off the query behind the scenes. This process is illustrated in Figure 3.1

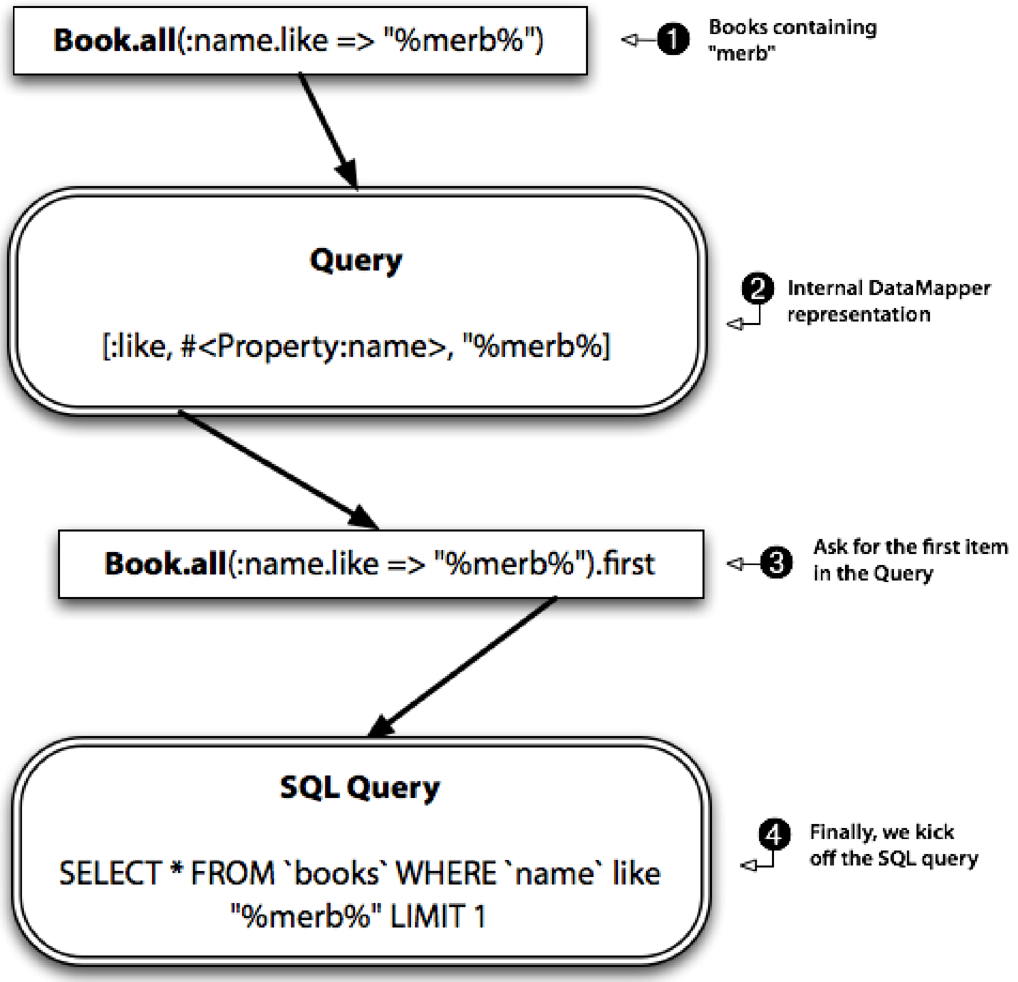

Book.all(:name.like => "%merb%")
1 Books containing "merb"
Query
[:like, #<Property:name>, "%merb%]
2 Internal DataMapper representation
Book.all(:name.like => "%merb%").first
3 Ask for the first item in the Query
SQL Query
SELECT * FROM `books` WHERE `name` like "%merb%" LIMIT 1
4 Finally, we kick off the SQL query

Figure 3.1 DataMapper loads the query only when you ask for it

This means that it's relatively inexpensive to create a new Query in a controller, because it will likely not be kicked off until you iterate over the objects in the view. If you cache the view, the query will never be called at all, resulting in a big win over ORMs that kick off the query as soon as you write it.

Also, because the Query object is simply waiting for instructions, you can modify an existing query before running it. Consider the code in Listing 3.28.

Listing 3.28 The Query is a first-class object, and can be modified before it's kicked off

```
orders = Order.all                                       # 1
orders.merge(:price.lt => 1)                             # 2
orders.each {|order| puts order}                         # 3
```

1. Start making the query
2. Modify the query, by adding a new condition
3. Kick off the SQL query with #each

Because the query is not actually kicked off until #3, it can be modified via `merge` in #2. As a result, it’s possible to create a large query in a helper or the controller, and then narrow it down in the view based on other conditions. This winnowing would be extremely cheap because it’s not sorting through objects already returned from the database, but merely modifying the query that will eventually be sent.

3.6 Summary

In this chapter, we have taken the sandwich shop from a mere facade to a fully functioning, data-driven website. Along the way, we have learned about what ORMs Merb supports, why you should use DataMapper, and how to effectively use the optimizations it provides.

Keep in mind that while Merb provides some out-of-the-box templates, they are only provided to help you get off the ground. You should probably fully replace them, style them, and integrate them into the experience you are attempting to provide with your site.

We also walked through the basics of the Merb router, showing you how the resource routes work under the covers. Don’t worry if you don’t fully understand it yet, we’ll be going into much more detail in a further chapter. Feel free to keep experimenting with some of the techniques we exposed you to in this chapter. It would probably make sense to try and apply the things you learned here to a simple project of your own before we delve into Merb controllers and views in more detail in the next chapter.

4 Automated Testing

Automatically testing your applications can be an extremely powerful way to ensure that changing and refactoring your codebase does not break the way your application works.

Testing Merb applications can be divided up into two parts. First, you'll want to test that requests for a particular URL actually return what you expect them to. You might also want to test your ORM layer to be sure it continues to work effectively.

The recommended way to test Merb applications follows a few rules:

1. Your tests should fail if you break the interface with your users
2. Your tests should not fail if the interface has not broken
3. Only write tests that confirm things that you actually care about

For instance, it is a somewhat popular practice to confirm, via tests, that certain helper methods are called when you go to a certain controller. However, this prevents you from moving the helper method inline, breaking it up, or combining it with another helper, without breaking a test.

This would require modifying existing tests, which would mean that you have no tests that confirm that the refactor didn't break anything. Additionally, if you modify the helper to do something different, your tests will still pass, even if user expectations have been broken.

Instead, Merb developers tend to test that the HTML output included what the helper is outputting. As a result, changes to what the helper outputs will break tests, but refactoring the helper, moving it inline, or breaking it up, will not cause any tests to fail.

What to test?

When testing a request, you will test the response when hitting your application with a particular URL. In your app, there are two kinds of URLs. Certain URLs are either used as part of a public API or are cached in Google. For instance, a blog or newspaper site has public URLs (or permalinks) that should not change over time.

Those URLs should be tested directly: given a particular explicit URL, you expect a specific response.

In other cases, the URL will always be generated by the application, and is not expected to remain permanently. For instance, the URL used in forms generated by your application will never be used directly by your users. In that case, Merb provides the same `#url` helper in your tests that is available in your views.

In that case, you'll still be requesting a particular URL, and testing its response, but you're using the same helper in your app and in your tests.

The bottom line is that you want to be testing only what you care about, and in the case of URLs, you need to decide whether or not you care about the explicit URL (as in the case of web services and pages cached on search engines), or whether you only care about internal consistency (in which case you can simply use the `#url` helper).

Let's take a look at one example of a test failure caused by refactoring code, something we want to avoid. Imagine a simple controller an action, shown in Listing 4.1, that simple renders the text `Hello`.

Listing 4.1 A simple controller that renders the text 'Hello'

```
class SimpleController < Application

  def say_hello
    print_hello
  end

  protected
  def print_hello
    "Hello"
  end

end
```

We'd also have a route, shown in Listing 4.2, that routes `/hello` to this controller an action.

Listing 4.2 Routing /hello to SimpleController#say_hello

```
Merb::Router.prepare do
  match("/hello").to(:controller => :simple_controller,
    :action => :say_hello).name(:say_hello)
end
```

The traditional Ruby way to test this, used frequently in Ruby on Rails tests, is to dispatch directly to the controller and action, shown in Listing 4.3.

Listing 4.3 Typical, incorrect way to test this simple case

```
it "calls print_hello" do
  controller = dispatch_to(SimpleController, :say_hello) do
    self.should_receive(:say_hello).and_return("Hello")
  end
end
```

This test confirms that when `SimpleController#say_hello` is dispatched to, `print_hello` method is called. This works great, especially when `print_hello` is a helper that does complex logic that requires some setup. However, it's trivial to show a case that doesn't break the functionality of the application at all, but seriously breaks this test in Listing 4.4.

Listing 4.4 It's easy to break the test without breaking functionality, which violates rule 2

```
class HelloController < Application

  def return_hello
    "Hello"
  end

end

Merb::Router.prepare do
  match("/hello").to(:controller => :hello_controller,
    :action => :return_hello).name(:say_hello)
end
```

After this refactoring, going to `/hello`, or even using `url(:say_hello)` inside the application still works correctly, but our test fails. This means that we no longer have a test that demonstrated that this refactoring maintained that functionality, but instead have a test that doesn't help us at all. Instead, Listing 4.5 shows the right way to spec this controller so that the specification survives this relatively simple refactor.

Listing 4.5 If you test the request and response, your tests will survive refactoring

```
it "returns Hello" do
  request(url(:say_hello)).body.should == "Hello"
end
```

This test will pass in both the original version of the code, and the refactored version. That's because we tested what we care about, and not the internal details of how we get there.

The three rules above are just a shorthand for the logic presented here. Thankfully, Merb's default testing harness provides you with a lot of tools to ensure that you're testing inbound requests and outbound responses, without tying your tests down to the internal implementation of your application.

In this chapter, we'll build a very simple CMS, and show how you'd build it test-first. We'll show how to use the Merb testing harness to write tests that will survive refactoring, and test what you actually care about. In future chapters, every example will be test-first, so this is a great time to get your bearings.

4.1 Start testing with a new Merb app

Let's start by generating a new Merb application, following the instructions in Listing 4.6

Listing 4.6 Generating the blank application

```
$ merb-gen app cms
$ merb-gen controller login
```

This should have generated a new controller, as well as a new file under `spec/request` called `login_spec`. At this point, we will just use the default routes, which will point the URL `/login` to the `Login` controller and the `index` action.

NOTE Merb comes with an authentication module which handles what we're going to work on here and more. As a result, your first step should be to comment out the lines referring to `merb-auth` in `dependencies.rb`. In a real application, you would just use `merb-auth`, and not roll your own login and password system.

If we load up the Merb application, by using the `merb` command, and hit `http://localhost:4000/login`, you'll see the text `You're in index of the Login controller.` Let's start by adding some specs that confirm that the `/login` URL actually works, as seen in Listing 4.7

Listing 4.7 Spec for /login

```
describe "/login", "index action" do
  it "returns successfully" do
    request("/login").should be_successful
  end
end
```

Running this spec should demonstrate that the `/login` URL returns something (specifically, a 200 status code), but that isn't exactly what we want it to do. Let's add a new spec that checks to make sure a form exists, with a login and password input in Listing 4.8.

Listing 4.8 Check to make sure that /login returns a real page

```
describe "/login", "index action" do
  before(:each) { @response = request("/login") }                     #1

  it "returns successfully" do
    @response.should be_successful
  end

  it "has a login form" do
    @response.should have_xpath("//form")
  end

  it "has a username box" do
    @response.should have_xpath(
      "//form//label/following-sibling::input[@type='text']")
  end

  it "has a password box" do
    @response.should have_xpath(
      "//form//label/following-sibling::input[@type='password']")
  end
end
```

Note that we've moved out the request to `/login` into a `before` block#1, because we're going to test multiple aspects of the response separately. This technique is sometimes referred to as "one assertion per test", and provides for significantly better isolation of test failures than making numerous assertions in a single test.

Using xpath is a simple and fast way to check the contents of the returned body, but we can make it even easier, using CSS selectors.

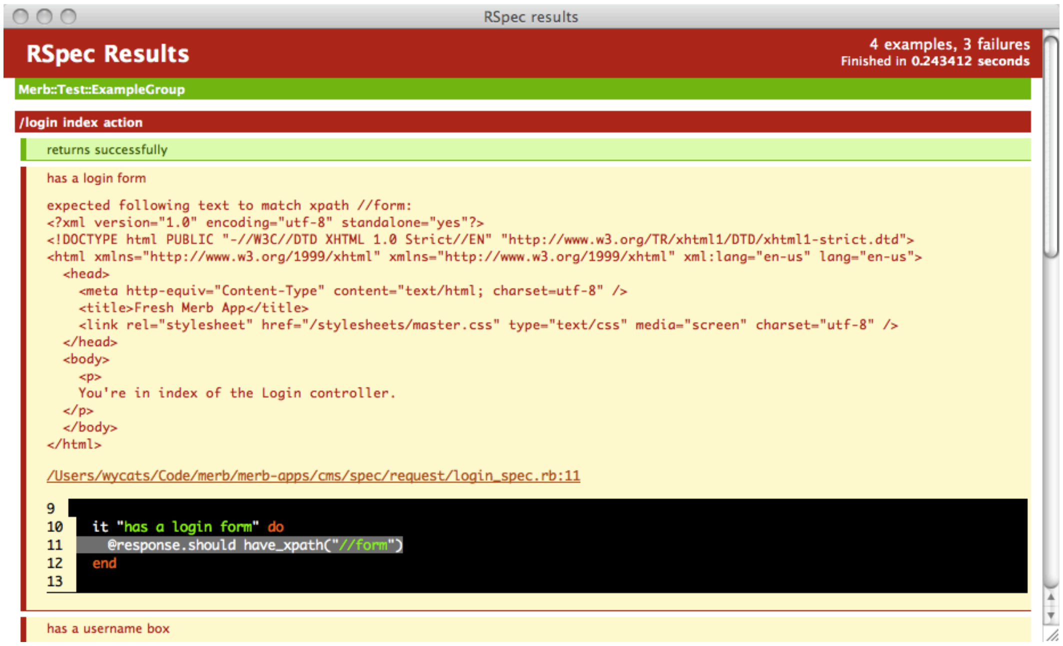
4.2 Even easier testing using `have_selector`

In each content test, we use the `have_xpath` helper to confirm that specific content is available. But you're probably more familiar with CSS-style selectors. That's why we make the the `have_selector` helper available, which uses CSS-style selectors to make assertions. For instance, you could have rewritten your tests like Listing 4.9

Listing 4.9 Using CSS selectors when speccing out return values

```
describe "/login", "index action" do
  before(:each) { @response = request("/login") }

  it "returns successfully" do
    @response.should be_successful
  end

  it "has a login form" do
    @response.should have_selector("form")
  end

  it "has a username box" do
    @response.should have_selector(
      "form label + input[type=text]")
  end

  it "has a password box" do
    @response.should have_selector(
      "form label + input[type=password]")
  end
end
```

If we run those specs, we will get three failures. You can see one of the failures in Figure 4.1



```
expected following text to match xpath //form:
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xmlns="http://www.w3.org/1999/xhtml" xml:lang="en-us" lang="en-us">
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <title>Fresh Merb App</title>
    <link rel="stylesheet" href="/stylesheets/master.css" type="text/css" media="screen" charset="utf-8" />
  </head>
  <body>
    <p>
    You're in index of the Login controller.
  </p>
  </body>
</html>

/Users/wycats/Code/merb/merb-apps/cms/spec/request/login_spec.rb:11

  it "has a login form" do
    @response.should have_xpath("//form")
  end
```

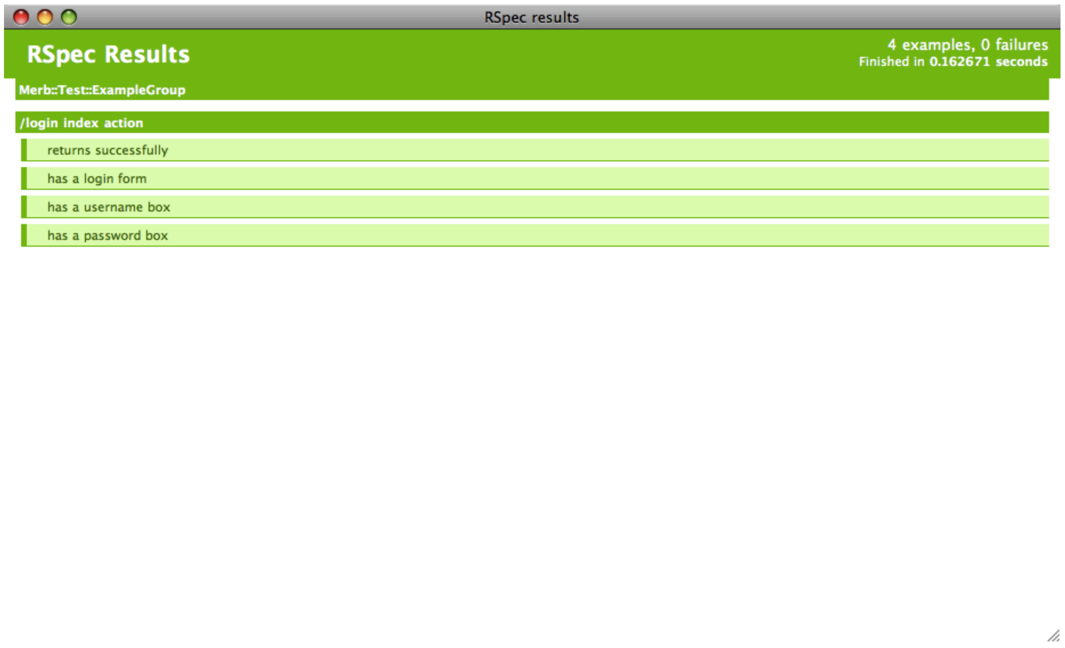
Figure 4.1 Before we add the login form, the tests fail as expected

To fix these failures, we'll just add in the form to the view. You can see the view that will cause the tests to pass in Listing 4.10

Listing 4.10 Creating the login form, so the specs will pass

```
<%= form :action => "/login" do %>
  <%= text_field :username, :label => "User" %>
  <%= password_field :password, :label => "Password" %>
  <%= submit "Login" %>
<% end =%>
```

If we now run the specs, they should be all green.


RSpec results

RSpec Results

4 examples, 0 failures
Finished in 0.162671 seconds

Merb::Test::ExampleGroup

/login index action

returns successfully

has a login form

has a username box

has a password box

Figure 4.1 Now that we have the login form available, all specs now pass

You'd probably want to add additional specs to handle the form's action, and other specifics. Feel free to take this opportunity to flesh out your specs and make sure your new specs pass.

Now that we've set up the username and password box, let's set up the required user model.

NOTE We'll use a very simple authentication system here, storing username and password in clear text in the database. For more secure authentication, Merb comes bundled with an authentication plugin, which we will cover in a future chapter.

4.3 Creating the User model

Next, let's set up a User model. We'll use the `model` generator, using the commands in Listing 4.11

Listing 4.11 Generating the User model

```
merb-gen model user username:string,password:string
rake db:automigrate
```

At this point, we don't need to add any logic to the model, so we don't need to write any tests. Next, we'll add support for the login and password.

4.4 Writing tests for logging in

Before adding support for logging in, let's write the specs for that functionality. We're going to want to test that if the correct username and password is provided, the user is logged in, but that when the incorrect information is provided, that we redirect back to the login.

Let's first add a test for when the information is correct. In that case, we will redirect to `/home`. You can see the results in Listing 4.12.

Listing 4.12 When the username is correct, confirm that going to the login form redirects us to /home

```
describe "/login", "index action" do
  ... snip ...

  describe "POSTing" do
    describe "when the login is correct" do
      it "redirects to /home" do
        response = request("/login", :method => "POST",
          :params => {:username => 'myUser', :password => 'myPass'})

        response.should redirect_to("/home")
      end
    end
  end
end
```

In this case, we have hardcoded in a user name and password. When testing, it is acceptable to start with simple cases, and then extend the tests with more realistic examples after more code has been written. Let's write the code that will implement this test.

4.5 Writing the code for login

But first, we will need to add a new route to handle POST to `/login`. We'll start by doing that in Listing 4.13.

Listing 4.13 POSTing to /login should dispatch to the Login controller and :post action

```
Merb::Router.prepare do
  match("/login", :method => :post).
    to(:controller => :login, :action => :post).name(:post_login)

  default_routes
end
```

This will point the `POST` to `/login` to the `Login` controller and `post` action. The next step, of course, is to actually create the controller that will make our tests pass.

Listing 4.14 Now, let's write the code that will redirect to /home if the username and password is correct

```
class Login < Application

  def index
    render
  end

  def post(username, password)
    if username == "myUser" && password == "myPass"
      redirect("/home")
    end
  end

end
```

Now that we have specs for what to do when the password is correct, let's write additional specs for when the password is wrong.

4.6 *Speccing out incorrect passwords*

If the password is incorrect, we'll want to redirect back to `/login`. Take a look at Listing 4.15.

Listing 4.15 When the password is incorrect

```
describe "/login", "index action" do
  ... snip ...

  describe "POSTing" do
    ... snip ..

    describe "when the login is incorrect" do
      it "redirects to /login" do
        response = request("/login", :method => "POST",
          :params => {:username => 'badUser', :password => 'badPass'})

        response.should redirect_to("/login")
      end
    end
  end
end
```

If we run these specs, Merb will tell us that it expected a redirect but got a 200. That's because we didn't return anything from the action, which Merb returned to the client as an empty string. We need to add in the functionality to redirect to `login` in the controller.

4.7 Updating the login controller to support incorrect logins

The updated controller, which passes the tests is in Listing 4.16.

Listing 4.16 Passing login controller

```
def post(username, password)
  if username == "myUser" && password == "myPass"
    redirect("/home")
  else
    redirect("/login")
  end
end
```

If you run the specs now, they will all pass.

Now that we have some basic authentication working, let's set up a way to add new users to the database.

4.8 Starting simple user management support

First, we'll make a page that lists all the current users. We'll need a new controller; let's call it Users. To generate the controller, run the command in Listing 4.17.

Listing 4.17 Generating the Users controller

```
$ merb-gen resource_controller users
[ADDED]  spec/requests/users_spec.rb
[ADDED]  app/controllers/users.rb
[ADDED]  app/views/users/index.html.erb
[ADDED]  app/views/users/show.html.erb
[ADDED]  app/views/users/edit.html.erb
[ADDED]  app/views/users/new.html.erb
[ADDED]  app/helpers/users_helper.rb
```

We now have a controller to manage our users, as well as a new request spec. Because `Users` will be a resource, Merb's generator adds a `resources` route to our router as shown in Listing 4.18.

Listing 4.18 Add a :users resource to the router

```
Merb::Router.prepare do
  resources :users                                                  #1
  match("/login", :method => :post).
    to(:controller => :login, :action => :post).name(:post_login)

  default_routes
end
```

The new line, #1, adds all of the routes required to route incoming requests to the users model. We previously discussed what routes are included by the `resources` in Chapter 3.

4.9 Speccing out a list of users

Let's start by testing the index page. When we generated a resource controller, Merb automatically generated a series of simple tests. For instance, one generated test confirms that `/users` returns successfully.

NOTE All of Merb's generated tests involve direct uses of the web service component of a generated resource. This is because Merb does not try to guess what your views will look like; every app has very different views. However, Merb does provide some "pending" specs, which should be used for informational purposes only, and should be modified for the actual content of your views.

Let's add another simple test. For now, we'll just make it a simple list of the available users. The first test might look something like Listing 4.19.

Listing 4.19 First list

```
describe "resource(:users)" do

  describe "GET" do
    it "displays a list" do
      request(resource(:users)).should have_selector("ul")
    end
  end

  describe "POST" do
    before(:each) do
      request(resource(:users), :method => :post,
        :params => {:user => {:username => "user", :password => "pass"}})
    end

    it "creates a new user" do
      request(resource(:users)).should(
        have_selector("ul li:contains(user)"))
    end
  end

end
```

This is a simple set of tests that simply test two things. First, the main index page for users displays a list. Second, once we've created a new user, the username appears in that list. Let's get cracking. Making the first test pass is pretty straight forward.

4.10 Showing a simple list

We can make the test pass very simply, as shown in Listing 4.20.

Listing 4.20 Making the first spec pass (index.html)

```
<h1>Users</h1>

<ul>

</ul>
```

Since the main test was pretty simple (it just looks for a `<ul>`), this will pass. Since we're going to be adding some users soon, Merb's generated `index` action grabs the list of all users in the controller in Listing 4.21

Listing 4.21 The controller index

```
class Users < Application

  def index
    @users = User.all
    render
  end

end
```

If we run our specs now, the first spec should pass. All we're doing here is storing off the list of all users in an instance variable, so we can get it later in the view.

The second spec `POST`s to `resource(:users)`, which will get routed to the `create` action of the `Users` controller. The spec says that the action will take a `Hash` containing a user, which will have a username or password. It is easy to write the `create` action, which will accept that `Hash`. The results are in Listing 4.22.

Listing 4.22 Creating a new user

```
class Users < Application
  ... snip ...

  def create(user)
    @user = User.create(user)
    if @user
      redirect(resource(@user))
    end
  end
end
```

The `create` action takes a user as a parameter. In this case, it will come in from the web as a `Hash` and our tests set that `Hash` as `{:username => "user", :password => "pass"}`. We then take the inbound Hash and try to create the user. If the user is correctly created, we redirect to the page for the user.

NOTE Merb has already generated a working `create` method, which is more feature-rich than the simple one here, so we don't need to actually put this code into place. However, it would successfully pass our simple test.

Next, in Listing 4.23, we'll go back to the index page, and have it show the list of users.

Listing 4.23 Show the list of users

```
<h1>Users</h1>

<ul>
  <% @users.each do |user| -%>
    <li><%= user.username %></li>
  <% end -%>
</ul>
```

If we run the specs now, they should pass. However, we still have significant pieces of missing functionality. For one, we have no actual way to create a new user in the first place. In the case of a Merb resource, you get a form for creating a new resource by going to `resource(:users, :new)`.

4.11 *Speccing out the finished form*

In Listing 4.24, let's create the test for a new form for editing a user.

Listing 4.24 Testing the new user form

```
describe "resource(:users, :new)" do

  before(:each) do
    @response = request(resource(:users, :new))
  end

  it "has a form" do
    @response.should have_xpath("//form[@action='#{resource(:users)}']")
  end

  it "contains a username field" do
    @response.should have_xpath("//form//input[@type='text'][@name='user[u
  end

  it "contains a password field" do
    @response.should have_xpath("//form//input[@type='password'][@name='us
  end

  it "contains a submit button" do
    @response.should have_xpath("//form//input[@type='submit']")
  end

end
```

Here, we create a simple test that confirms that we have a form, POSTing to `resource(:users)`, that has the requisite `username` and `password` fields, and also has a submit button. In the previous test, we checked to be sure that when POSTing with that information, a new user was created and visible on the `index` page.

4.12 Finishing up the new user form

Now, in Listing 4.25, let's make this spec pass. It's pretty straight-forward at this point.

Listing 4.25 Make the new user form

```
<h1>New User</h1>

<%= form_for(@user, :action => resource(:users)) do %>
  <%= text_field :username %>
  <%= password_field :password %>
  <%= submit "Create" %>
<% end =%>
```

As you can see, it's quite possible to create simple test-first tests that can test through the interface your user is testing, but still prove a whole lot about your application. At this point, try hitting `/users/new`, create a new user, and then go to `/users` to see your application working.

The fact that everything worked, without ever having to go play with the application as you were building it, should be reason enough to favor this approach to testing. But there are still a few things left to do. We'll want to handle invalid data passed in to the `new` page.

As an exercise, try and complete this simple login manager. The goal is to have a working login, as well as a working way to create new users and modify existing users. For a challenge, try adding a requirement that usernames be at least six characters, and display feedback to the user when they fail to correctly create their user. To complete this task, you'll want to use Merb's `error_messages_for` helper.

The complete working code can be found in the downloadable code provided at this book's page on Manning's website. When you're working on adding the features, pay special attention to writing your tests first, and testing just the requests and responses.

4.13 Summary

When we write automated regression tests, our mission is to create tests that can be run again and again, even as we refactor our application. This means that our test suite needs to fully cover the application, but from the perspective of the application's features, not its implementation.

Our testing strategy follows three basic rules. First, when a bug is introduced with a code change, at least one test should fail. Second, if no bug is introduced with a code change, no tests should fail. Finally, your tests should fully cover only the functionality that you actually care about, and no more.

In this chapter, we showed how to apply those rules to the real-life creation of a login form and user management. By testing requests to our application and the responses to those requests, we provide evidence that the user experience doesn't change when we modify our application.

We used Merb's built-in `request` helper to perform those requests, which emulates a browser down to support for cookies and sessions, and automatically sending the default headers and methods.

Now that we understand how Merb works and how to use test-driven development to develop a Merb app, we'll next take a look at how to deploy a Merb application to a live production server.