PROJECT 1

The Store (Shoplet)

CHAPTER 3

Introduction to Rails

This chapter will serve as an introduction to the various parts of the Rails framework. Because Rails is loosely coupled, it makes sense to introduce the parts by themselves, and then describe how to fit them together.

On July 24, 2004 the first public version (0.5) of Rails was released. David Heinemeier Hansson extracted and generalized it from Basecamp, 37signals' first commercial web application. This first release immediately created a huge buzz in the developer community, partly because of the rapid development made possible by the Rails application architecture. Fifteen months later, after much progress and many new features, version 1.0 was released on December 13, 2005. This release solidified everything that made Rails successful and fixed most bugs and problems discovered before going final.

The next major milestone came on March 28, 2006, when David and company released version 1.1 with lots of new, interesting support for various JavaScript and Ajax features, better handling when returning different document formats, polymorphic database associations, and integration tests.

The last year has seen more than 20 books published about Ruby and Rails. Adoption among American companies is seriously getting started, and in the summer of 2006 the first-ever International Rails Conference was held in Chicago. It was a major success, and a few months later a European RailsConf was held in London.

Let's get started with the different parts of Rails.

The Structure of a Rails Application

When creating a new Rails application with the `rails` script, the Rails application generator will create it following certain conventions. The output directory structure will be deep and will have support for many different things. Contrary to what you might expect, this makes development easier within Rails. These constraints free you from lots of decisions that otherwise would have to be made. Of course, many of the Rails helper scripts also depend on this structure being there, so you deviate from it at your own risk.

These directories are automatically created when generating a new Rails application:

- `app`: This is where most of the application code resides. It has subdirectories for models, views, and controllers, and also for helpers.
- `components`: Before the plug-in system was finished, components were the main way to extend a Rails application. They're now mildly deprecated, and I won't cover them at all in this book.

- `config`: The `config` directory contains some pretty important files, including the one for database configuration. It also contains the different configuration parts for separate environments.
- `db`: The `db` directory houses all migration information for the databases.
- `doc`: You can generate an RDoc for your application. If you choose to do that, it will reside in this directory.
- `lib`: If you create code that doesn't fit neatly into the categories of model or controller, you can place it in the `lib` directory. The `lib` directory is on the load path, but you need to `require` the parts you need. Also, when adding new Rake tasks to your build system, you should put the file containing the tasks in the `tasks` subdirectory in the `lib` directory, so Rake will automatically recognize it.
- `log`: When your application is running, all the logs will end up in this directory.
- `public`: Some parts of your web application need to be directly accessible without going through the Rails call stack; for example, images and style sheets. You should put these things in the `public` directory. You should also set this directory as the application root if you're using Apache to deploy your application. When caching is configured in your Rails application, the cached files will end up here too.
- `script`: Rails depends on some helper scripts for different purposes. You can find all of these in this directory. I'll describe these scripts further in the section "Rails Helper Scripts."
- `test`: You can test Rails applications in many different ways. What is common for all methods is that you should place the actual tests in this directory. The Rails Rake file already supports much of the directory structure within, so try to make your testing fit within the existing architecture.
- `tmp`: During runtime, Rails needs to create numerous temporary files. By default, these are placed in this directory. You should never place anything you need to keep here.
- `vendor`: When installing plug-ins, they'll end up in `vendor/plugins`. Also, if you decide to freeze the current Rails version, all the Rails Gems will be copied to this directory.

Of all these directories, the most important ones are the `app`, `config`, `test`, and `db` directories. This is where most of the work on a Rails application takes place. Try to take a few minutes with a newly generated application to walk through all subdirectories and see what they contain.

Models

In Rails, the models are where the data resides. The model classes correspond more or less directly to a database table (if you use ActiveRecord, that is). You can find all models in the directory `app/models`, where each file corresponds to one model. It can be easy to create a model. In fact, the easiest possible model could look like this, in a file named `order_item.rb`:

```
class OrderItem < ActiveRecord::Base
end
```

In this case, the only thing necessary is to create a class that inherits from `ActiveRecord::Base`. If you have configured a working database, and that database contains a table named `order_items`, then this model will work fine. The next step would be to add some references to other parts of your model:

```
class OrderItem < ActiveRecord::Base
  belongs_to :order
end
```

`belongs_to` is one of the more common references that can be used. In this case, the agreement is that the `order_items` table has a foreign key field called `order_id` that you can use to find the `Order` that this `OrderItem` is part of. Correspondingly, the `Order` model could look like this:

```
class Order < ActiveRecord::Base
  has_many :order_items
  has_one :shipping_address, :class_name => 'Address'
end
```

As you can see, in this way you can use one model object (`Address`) with a different name in a model. This allows you to use a generic model object such as `Address` in several different parts of your application, if need be.

Models also contain validation information. For example, take the `Address` model referenced earlier:

```
class Address < ActiveRecord::Base
  belongs_to :order
  validates_presence_of :country, :zip
  validates_numericality_of :zip
end
```

This simply says that an `Address` needs to contain country and ZIP code information, and a ZIP code needs to be only digits to be a valid ZIP code. There are many kinds of validators, and you can also write your own.

Defining models is one thing, but you also need to know how to use them. Mostly, it's intuitive. For example, say you want to find all orders:

```
Order.find :all
```

Or, say you have an ID for an order and want to get that instance:

```
Order.find 123
```

Or, say you have an order instance called `o` and you want to create a new address that references this order:

```
Address.create :order => o, :country => 'Sweden',
          :zip => '12559'
```

This was a small introduction to models with ActiveRecord. I'll expand on most of this information in later chapters.

Controllers

The controller is the part of your application that first sees a request. The controller decides which models to use, how to use them, and also which view to render. Controller code is usually simple to write when following the Rails conventions. If you have to do something different, things can get more complicated, though.

A Rails controller is a class that can be found in `app/controllers`, and controllers follow a specific naming scheme. Specifically, a controller should be called `SomethingController` and exist in a file called `something_controller.rb`. If you follow this convention, that controller's actions will by default be available under the `/something/action` URL. You can change this by editing the file `config/routes.rb`, but it's a useful convention to stick to. If there is no action named in the URL, the default action name will be `index`. This is a neat thing in Rails: all public methods in a controller are an action. So, you could create a simple controller that displays the view named `index.rhtml` like this:

```
class FooController < ApplicationController
  def index
  end
end
```

If you don't specify a view to render, by default Rails will use one with the same name as the action. However, you can also specify one explicitly:

```
class FooController < ApplicationController
  def index
    render :action => 'bar'
  end
end
```

The `ApplicationController` class is something Rails provides for free. You can find it in the `app/controllers/application.rb` file, and it's useful to change this file to add things to all controllers.

There isn't much to tell about controllers. Except for specifying which view should be rendered, you can also provide data to the view. This is best done by setting instance variables to the values you want the view to have. With some trickery, Rails copies all instance variables from the current controller to the view that should be rendered. So, you can give a view some much-needed data by doing it like this:

```
class FooController < ApplicationController
  def index
    @products = Product.find :all
    @title = "All the Products in the World"
  end
end
```

Another useful thing you can do with controllers is to use filters. You can specify that some code should be executed before, after, or around every action. This allows you to achieve authentication and authorization quite easily, but you can also add encryption, compression, or many other things to your application in this way. Filters will be covered later in the book, when we look at implementing authentication for Rails applications.

Views

The view is the part that generates HTML (or XML, or JavaScript, or anything else really). In Rails, the standard way of doing this templating is called ERb, and the standard file extension is RHTML. There are also a few other ways of doing this, but ERb is the method I'll use in this book.

The Rails view templates live in the `app/views/{controllername}/` directory. That means, if our controller is named `FooController`, the index view will be found in `app/views/foo/index.rhtml`. Most of an RHTML file is regular HTML, but parts enclosed in `<% %>` and `<%= %>` blocks are evaluated as Ruby code. As I mentioned in the section on controllers, all instance variables from the controller will be available in the view too. So, say this fragment is in the `index.rhtml` file for the `FooController` specified earlier:

```
<head>
  <title><%= @title %></title>
</head>
```

Then, this would be what was returned to the browser:

```
<head>
  <title>All the Products in the World</title>
</head>
```

That's more or less it for the views. They aren't complicated. In the basic case it's simple to generate the dynamic code you need inside the confines of regular HTML. Rails does give you helpers for many tasks. For example, you should never write your own `<FORM>` tag, because there are many things to keep in mind, and you'll have lots of pieces to change if you switch URL schemes or something like that. For example, say you want to have a button that posts some information to your server. You could use the helper `button_to` like this:

```
Press this button:
<%= button_to "Press Me", :action => 'button_press' %>
```

This generates a small form with a button that when pressed does a `POST` to the URL corresponding to the current controller, but with the action `button_press`. There are many helpers in Rails; I'll introduce them as we go.

Layouts

If you want to have a common style on more than one page of your application it can be a good idea to consider creating a layout. By default, a layout is found in `app/views/layouts/`, and it should have the same name as the controller under execution for it to be loaded automatically. For example, you should call the layout for `FooController` `app/views/layouts/foo.rb`. This file could contain the HTML header and footer, a menu, and everything else that should be shared in many views. You mark the place where Rails should insert the real view by calling `yield` inside an output block, like this:

```
<%= yield %>
```

This outputs the dynamically generated data from the real view at that place. If, for some reason, you would like to have the content duplicated, you shouldn't call `yield` twice. Instead, save the output from the first `yield` and print it two times:

```
<% content = yield %>
ABC:
<%= content %>
CDE:
<%= content %>
```

If you want to have the same layout for more than one controller, you can specify a specific layout inside your controller:

```
class FooController < ApplicationController
  layout "bar"
end
```

For all actions in `FooController`, the file `bar.rhtml` is used as a layout, instead of the default `foo.rhtml`.

Partials

You'll often find yourself needing certain blocks of code over and over again. For example, say you always want to print product details the same way. You could copy and paste this in all the places you need it, but that wouldn't be DRY. So, for these cases you have access to partials. A partial should be in the same directory as the view using it (but you can use partials from other controllers too). You should name it with an underscore (_) before the name to differentiate it from real views. So, you could have a partial that's named `_product.rhtml` and looks like this:

```
<p><b><%= product.name %></b> $<%= product.price %></p>
```

You could use it from a view in this manner:

```
Your choice: <%= render :partial => 'product', :locals =>
                   {:product => @prod} %>
```

We need to explicitly import the product to use by giving the `locals` parameter to the call. This gives us some flexibility, though. For example, say that we had several products in a list called `@products` and we wanted these rendered the same way. We could do that with the same partial:

```
Products:<br/>
<%= render :partial => 'product', :collection => @products %>
```

The partial is called once for each item in the collection, and the current object is set to a local variable with the same name as the partial itself.

Partials are incredibly useful for allowing great code reuse. You should use them as much as possible, especially later, when we start talking about how to use Ajax to re-render only parts of a page.

THE MVC PATTERN

Rails is a pure implementation of the Model-View-Controller (MVC) design pattern. This pattern first originated in the Smalltalk world but is now widespread (and is known under the name Model-2 in the Sun parlance). The pattern has been implemented hundreds of times in several different kinds of applications, but it shines most in those situations where you can cleanly delineate the user interface from the data back end, and the main purpose of the interface is to present massive amounts of data to the user.

The pattern was first described in 1979 by Trygve Reenskaug, then working on Smalltalk at Xerox labs. His implementation of the pattern is described in depth in a well-read paper called "Applications Programming in Smalltalk-80: How to use Model-View-Controller" by Steve Burbeck. Many other systems have been inspired by this original implementation, including parts of Cocoa for Mac OS X, Java Swing, Microsoft Foundation Classes (MFC), and the Qt toolkit. Apart from these, the pattern is heavily used in many web systems.

The basic idea in MVC is that you divide the responsibility into three parts: the model, the view, and the controller. Each part should be more or less self contained, meaning that a change in the view system should not have any impact on the model or controller logic. As a typical benefit, if you code a Java Swing application in a typical MVC architecture, you would only need to change the view logic to port this functionality to a web interface. Of course, this ideal will almost never be purely realized, but it's still useful to design your application with this separation of concerns in mind.

The model is where the domain-specific representation of data resides. This layer is also known as the domain layer. The model objects should be composed of all raw data the application needs, with domain logic added to this. A typical example of model objects can be Product, Order, and Customer. If the application needs persistence of some kind, this will also be a part of the model layer. The view and controller should ideally not need to know anything about databases or other forms of persistence.

The controller is responsible for providing the view with all data that should be available for the current event. The controller reacts to events by evaluating them and then updating the view as a response. Controllers are commonly divided into separate actions, where any one action corresponds to a specific event. For example, in the Java framework Struts, you never write a controller; you just provide different actions for different URLs. The real controller is one big class that dispatches to the different actions and provides different services to the system.

The view renders the model information and the data provided by the controller in a suitable form for the current request. The view displays user interaction elements, and in a typical web application it corresponds neatly to the code that directly generates HTML. The ideal view doesn't contain any logic at all, but only displays the information that is available to it.

The MVC pattern is easy to see in the Rails framework. Most of the components are named in such a way that they correspond directly to the different parts of the pattern. For example, model objects in Rails map neatly to the M in MVC, by handling database communication, validation, and internal logic. Controller classes map to the C in MVC, and each public method in a controller corresponds to a separate action. Finally, there is one separate view—which is an RHTML file—for each distinct action, unless you override the rendering logic. All pieces are there.

When coding an application with Rails it is important to follow this separation of concerns in your own code too. If you don't, many of the benefits you can get from Rails will be void. In real terms, this is simple: never write any code in the view templates that contains business logic in any way. If you find yourself using much code in RHTML, be suspicious. It's not always a sign that something is bad, but it could be. If you need lots of presentation logic, try to abstract it and put it into helper methods. By the same token, try to refactor all logic that concerns the model into the model classes. Make your controllers as clean as possible. Look at the code generated for scaffolds for examples of how to write controller logic that only contains what's

necessary for the controller. Finally, any view information should not be found in the model objects. This is probably one of the few things I object to in the Rails framework. Validation messages are usually placed inside the model objects, together with the rest of the validation rules. However, because this is view information, it's certainly not the right place for it. Keep this in mind, especially when trying to create something that should be used in more than one language.

Much of the power of Rails comes from MVC and the relentless use of it. Don't let that get wasted.

The Other Parts of Rails

Rails is divided into several smaller packages, each more or less self sufficient. ActiveRecord is the component used for the models; ActionPack contains ActionController and ActionView, which are the controller and view parts, respectively. The package Rails (previously called Railties) contains the code that pulls all these parts together. That isn't everything, though—Rails contains several more packages.

ActiveSupport

ActiveSupport's main purpose is to add several helpful extensions to different parts of the Ruby core. It contains core extensions and new utility classes that aren't Rails specific, but that Rails needs to make life easier for the developer. For example, it contains the class `HashWithIndifferentAccess`, which lets you index into a hash with a symbol or a string, and you'll get the "right" value back regardless of which you choose.

ActiveSupport adds many methods to the core classes. The time and date utilities are especially helpful. When ActiveSupport is loaded, you can write things such as `2.days.ago` or `Time.today + 2.hours + 14.minutes`, and get something back that does what you want.

Probably the best thing that ActiveSupport provides is called the `to_proc` hack. It adds a `to_proc` method to the `Symbol` class. The utility of this is that some common iteration scenarios in Ruby will be much more succinct. Thanks to this, instead of writing

```
%w(abc cde efg).map {|v| v.inspect}
```

you can write

```
%w(abc cde efg).map &:inspect
```

This makes code much easier to read. Refer to Appendix A for a small explanation of how this works.

ActionMailer

The ActionMailer package makes it simple to create templates for sending mail, by using much the same view techniques that regular Rails views provide. If you need to send mail with a predetermined format, ActionMailer is probably the best way to do it.

ActionWebService

The new rage in Rails and web applications with regard to web services is called Representational State Transfer (REST). The base idea is that you use regular HTTP calls with human-readable URLs to provide the most common operations. You use the different HTTP methods (such as `GET`, `POST`, `PUT`, `DELETE`, and so on) to provide different operations on the same address, and in the request you provide just the information needed, in a simple XML or YAML format. This makes web services much less painful than they currently are with Simple Object Access Protocol (SOAP) and Web Services Description Language (WSDL). ActionWebService helps you develop clients for such services easily in Rails. Because Rails already has good support for easy URLs, and also makes it easy to do different things depending on which HTTP method is used, Rails has become one of the best ways available with which to implement REST services.

ActiveResource

ActiveResource is not a part of Rails at the time of writing, but will probably be released when you're reading this. The purpose is to take the REST architecture of Rails one step further by providing easy ways to expose web services transparently. This is accomplished by using the same MVC code that regular Rails uses. Because Service Oriented Architecture (SOA) is hailed as the future of application development, this could well provide another sweet spot for Rails applications.

Rails Helper Scripts

Ruby's first major philosophy point is DRY: Don't Repeat Yourself. This shines through the Rails framework in many ways. One of the ways Rails makes your life simpler as a developer is by providing several scripts that help you with different tasks in the development of an application. I'll guide you through the scripts that can be found in the `script` directory of a newly generated Rails application. Some of these you won't use often; some will be totally indispensable. I won't describe Rake tasks used for different tasks here, though, because they have a slightly different purpose.

about

If you create a new Rails application, start a WEBrick instance, and point your browser to `http://localhost:3000/`, you'll see a page describing your application. The `about` script provides exactly the same information. It prints the versions of Ruby, RubyGems, and Rails (and all dependent packages). It also prints which environment is used, the database, and which migration version number the application is currently at. As such, it can be useful debug information.

breakpointer

The `breakpoint` library in Ruby allows you to call a method at an arbitrary point of your code. If you run this program locally, outside of Rails, this will open up an Interactive Ruby (IRB) instance that lets you debug the application at the point of the breakpoint. But if you use such a breakpoint in a Rails application, something else happens. Because most Rails applications

don't have a natural place to dump you into an IRB, the application won't do that. Instead, Rails will provide a breakpoint server. The `breakpointer` script lets you attach to such a breakpoint server (it works remotely too), and when the application hits a breakpoint you get an IRB session into it. Quite neat, and very useful.

console

This script lets you start up an IRB console with the current Rails application loaded. You have access to your model objects, all Rails helpers, and much more. You can use this script to test how you think some method in ActiveRecord will work, or almost anything else you can think of. When I create a new Rails application, I usually make sure always to have a console started, because it's so easy to test things out. You can also use it to check on the data in the database. Because you can execute arbitrary SQL through a model object, this means you can do mostly anything you want.

destroy

The `destroy` script is the negative counterpart of the `generate` script, described next. It's generic and uses a generator class to remove something that has been created by the `generate` script. It won't remove things you have changed, only unchanged files and code. However, it can be handy to have, because the `generate` script creates many files in different places.

generate

The `generate` script is one of the most important scripts for DRY in Rails. The purpose is to allow you to autogenerate as much code as possible. As such, it doesn't do anything you couldn't do by hand, but remembering everything you should add when creating a new model quickly becomes tedious. The `generate` script is also generic, which means you can install new generators that can generate your own custom code. The most-used generators are probably the model, the controller, and the scaffold generators.

The model generator creates a new empty model file, a new migration file for this model, and several test files where you can add your own test code. The controller creates a simple controller file and corresponding tests. Scaffolding creates a new controller and views that give access to CRUD operations for an existing model. It's useful as a base for further code, especially for administrative interfaces. You'll use this approach with the Shoplet application in the next chapter.

plugin

Rails' plug-in mechanism is very powerful (and one of the main methods the core Rails team uses to test new features). This script is the portal to all plug-in goodness. You use it to install new plug-ins and remove old ones. You can also use the script to extract parts of an existing code base into a plug-in. The most common usage is to install a new plug-in, possibly through the `-x` flag (that only works if you store your Rails application in a Subversion repository—which you should), which links directly to the version control repository for the plug-in you want to install. Very handy.

runner

Sometimes you need to run a bit of code within the context of a Rails application without having to start an IRB console. In those cases, the runner allows you to execute an arbitrary piece of Ruby code, written on the command line. One of the more regular uses of this mechanism is to add a `crontab` entry to clean old sessions from the ActiveRecord store by running a command with `runner`.

server

You most commonly use the `server` script to start up your application and try it out. It starts up a standalone Rails instance listening on port 3000. The default is to start WEBrick, but if you have Mongrel or lighttpd installed, the script will use one of them instead. In production, the `server` script isn't usually used, because there are better ways to deploy a Rails application. However, the `server` script works fine for development, especially when you need to restart your application often.

RAILS VS. OTHER WEB FRAMEWORKS

There are hundreds, if not thousands of web frameworks in different languages. Some people consider it a good task to learn a programming language better so you can implement your own framework. In that respect it would be hard to compare Rails to other frameworks. However, I'll still try to describe the most popular frameworks in several different languages and also contrast them against Rails. Some would say that such a comparison will always be subjective. I won't argue with that; these are my personal thoughts about the closest competitors to Rails.

Struts (Java)

Struts is arguably the number one web framework used with Java. It doesn't usually sit alone, though. The most common situation is probably Struts combined with Tiles and Hibernate. That's the scenario I'll consider here, because the different parts neatly match against the letters in MVC. This is also the first major difference between Struts and Rails. Struts doesn't provide MVC functionality at all; it just gives you the controller part. Tiles is easy to get going with Struts, but it's not part of the framework and also needs separate configuration. Configuration is also the part of Struts that can be quite painful. You gain lots of flexibility, but at the same time you need to keep numerous XML files up to date.

Rails configuration is basically guesswork unless you want to provide something substantial to it. Only one configuration item is needed to get Rails working, and that is the database parameters.

The model is the big difference between Struts and Rails, though. The only part of the model that is available within Struts is validation, and this also requires some configuration to get going. Using Hibernate gives you much flexibility, and you can do some things with Hibernate that are hard to achieve with ActiveRecord in Rails. This flexibility comes at the cost of another configuration file. You also need to create model classes (or autogenerate them once and work from there), which contain getters and setters for attributes. It is in the model that the cost of a statically typed, compiled language gets in the way of rapidly getting something working.

In summary, Struts gives you much flexibility, but the cost of getting started is much higher. Maintenance also suffers, because there is just much more code and configuration.

Django (Python)

Django is, on many levels, very similar to Rails. The big difference between the two frameworks is first and foremost one of philosophy, and this difference hails more from the respective languages than from any explicit design choice the creators have made in the software.

There is also a difference in feel. Django seems more aimed at web publishing applications, while Rails is well suited for generic web application development. Django ships with some things out of the box that Rails doesn't contain. For example, a built-in authentication system comes with Django, but not with Rails.

Another difference is how Django places all information about models in one place. Python code defines the database tables, and this is generated from the same place that defines the attributes on a model object. In Rails you define migrations for your models in Ruby, and then the attributes in the database are available to your model object without you having to specify them. These are two slightly different approaches, but in the end which one you choose doesn't matter much.

In summary, the big difference between the frameworks is in style and feeling, rather than functionality. They are both perfectly satisfactory for most web tasks. However, if you want Ajax, Rails will probably give you a big head start. Also, Rails runs on the JVM, but Django does not.

Seaside (Smalltalk)

It's hard to compare Seaside with Rails, because they embody completely different design and architectural decisions. Seaside is a web framework based on continuations. Now, the thing that makes Seaside such an interesting framework is that it inverts the usual way of writing a web application. Instead of having small separate parts of functionality that trigger at different points in the life cycle of a web interaction, and maintaining state by saving away small nuggets of information indexed by data in client cookies, Seaside lets you write your web application more or less in the same style as you would write a regular command-line application. You start up, initiating everything, then loop and let the customers specify products they want to order. Then, when that part is finished you go ahead by asking which delivery address they want the order sent to, and so on.

In reality, Seaside works the same way as all other web applications, under the covers. However, this part of the system is hidden from the person using the framework. Most of the complexity of maintaining state is managed by saving continuations at strategic points of the execution that can be resumed at any time, or discarded.

Seaside doesn't contain explicit support for a database or model layer. The developer has to add this. In the same manner, Seaside doesn't contain a separate templating system for easily creating HTML views. Instead, you write Smalltalk code that generates HTML. In some situations this flexibility can be good, but in other cases—especially when creating larger applications—this makes it hard to let a designer generate the HTML design for you and then integrate it into the application.

So, Seaside is good at several things that almost no other framework handles well, but other detriments make Rails a more widely useful full-stack solution for web applications.

PHP

There are certainly many frameworks for PHP that I could compare Rails against, but Ruby on Rails sits more or less in the exact same spot that PHP has traditionally occupied. The big difference is that Rails scales better and provides more benefits. With PHP, you first need to choose a framework that gives you some kind of

Object-Relational mapping. For small sites, you could use SQL directly, but that would be far from MVC, and as I said earlier, not scalable. Zend (the company behind PHP) has announced the Zend Framework, aimed at competing with Rails and providing some of the features that are missing (or hard to use) in a regular PHP installation.

PHP is good for a few reasons, though. It's widely available. It's easy to learn and many people know it already. It's designed specifically to be a web language. However, this is also a weakness, because it isn't a general-purpose language like Ruby. Sometimes PHP doesn't feel object-oriented enough, and it can be hard to abstract functionality.

The big problem compared to Rails is that there is no difference between the view and the other places. There's never an overarching controller. Everything is file based. This makes a PHP project simple to start but hard to continue working with.

So, in summary, if you want one or two pages that fetch data from a database, you can probably do it faster with PHP than Rails, but that's about it.

Testing

One of the strengths of Rails is that it makes testing easy and natural. You can run your tests often, and the testing is an integral part of the application. I won't talk specifically about how to write the tests here, but instead will describe the different kinds of tests available to you, and where they live.

First, to start testing, just run `rake` in your application directory. That runs all unit tests you've written. That's the first and best way to test.

As you can see, if you check in the `test` directory of a new application, Rails comes with several versions of tests. The directories found here are `fixtures`, `functional`, `integration`, `mocks`, and `unit`. However, only the `functional`, `integration`, and `unit` directories contain tests. In the `fixtures` directory, you'll define data fixtures for your models, so that the tests you write can have a consistent and dependable basis for testing. In the same way, the `mocks` directory contains fake implementations of real parts of your application. The canonical example is the credit card payment service. You don't want to talk with a payment gateway each time you run tests, so instead you create a mock of this service code, which generates implementations of the interesting methods that return predetermined values.

In Rails, unit tests have the explicit responsibility to test the model. There should be one file in the `unit` directory for each model you've created, and if you've used the generators to create them (as you should), these files will have been created automatically.

A typical unit test looks much like a regular `Test::Unit` test case, except that Rails unit tests can specify a fixture to use. You can use more than one fixture in the same test case, but the way Rails generates your test files, you'll get one separate fixture for each test case file.

Functional tests—in contrast to unit tests—should test controllers. They're usually written at a higher level than unit tests, because the functionality of models is a part of the functioning of the controllers. The best way to see how to write simple functional tests is to generate a scaffold controller and read the code for the generated functional test. As you can see in Listing 3-1, the language is high level, and looks more like a DSL for testing than regular Ruby code.

Listing 3-1. *Parts of Functional Test Code for a User Controller*

```
class UsersControllerTest < Test::Unit::TestCase
  fixtures :users

  def setup
    @controller = UsersController.new
    @request    = ActionController::TestRequest.new
    @response   = ActionController::TestResponse.new
  end

  def test_index
    get :index
    assert_response :success
    assert_template 'list'
  end

  def test_list
    get :list

    assert_response :success
    assert_template 'list'

    assert_not_nil assigns(:users)
  end

  def test_show
    get :show, :id => 1

    assert_response :success
    assert_template 'show'

    assert_not_nil assigns(:user)
    assert assigns(:user).valid?
  end

  def test_new
    get :new

    assert_response :success
    assert_template 'new'

    assert_not_nil assigns(:user)
  end
end
```

Integration tests, finally, are even higher than functional tests. The purpose is to test the flow through an application. As such, this kind of testing uses operations that resemble the actions that a user will execute while using the application, rather than going behind the covers, on the bare metal, like functional and unit tests. In the integration test, you send in requests, check the response, follow redirects, fill in form information, and so on.

All in all, these three versions of testing let you have good control over your application's behavior, and it also makes it easy to test against regression. As such, testing should always be written concurrently while writing the implementation code. Alas, due to space constraints, this book won't contain much testing information. This will be part of the downloadable complete code, and I'll also show you how to test your first application with unit and functional tests. However, the rest of the projects won't have tests written for them in this book.

Plug-Ins

Rails sports an easy-to-use plug-in system. It allows you to extend and replace functionality in any part of the core system, or add completely new capabilities. There's some debate going on as to whether there should be a separate plug-in mechanism from RubyGems. That's because you can do almost everything with RubyGems that you can do with plug-ins, and Gems also gives you some extra features. My own opinion is that it can be useful to associate plug-ins with a specific Rails application. If you install a Gem, it will be available to all Ruby applications in the system. It's also easier to develop a plug-in without some of the overhead that a Gem requires.

The way to work with plug-ins is through the `script/plugin` script. With it, you can list all available plug-ins, and install, update, and remove plug-ins. To get more information about how the script works, just run it without parameters.

Some plug-ins are more useful than others. These are my picks of a few that have proven handy from time to time.

Acts As Taggable

This simple plug-in adds the class method `acts_as_taggable` to ActiveRecord. If you mark a model class with this method, each instance of that model can be associated with tags. The plug-in also gives you ways to search on tags.

CAS Filter

If your organization uses Central Authentication Server (CAS), which is a common protocol for centralized authentication and single sign-on, this filter allows you to add authentication easily to your application. You have the choice to protect only parts of the application, or everything.

Globalize Plug-In

This plug-in adds transparent translation of both models and views. Because Rails currently doesn't handle internationalization that well, something like this plug-in is much needed if you want to do development outside the United States. Globalize provides localization of numbers, dates, and currencies, and helps with translation.

Rails Engines

This plug-in is a little bit different, in that it doesn't add anything by itself. Instead, it adds more specialized plug-in functionality. In the parlance of engines, you can install a Login Engine, or a User Engine, or something else, and these engines provide a full chunk of more-or-less finished functionality that you can use or modify. The idea is that these engines won't affect the existing code at all, and they are aimed at being fully featured, vertical, MVC solutions.

Summary

As you can see, Rails is a fairly complex beast, comprised of several interacting parts. As such, this chapter has given you a cursory introduction to all the important parts. However, there's still much to learn. Instead of taking the components on separately, in the next chapter we'll start building an application, using most of the parts of a typical Rails application. In this way it will be easier to see how all the parts fit together.

The creators of Rails often say that Rails is "opinionated software." This is manifest in many ways, but the most important way is that you should follow specific paths when creating an application. You can deviate from them, but the framework benefits you most while following the rails. The purpose of the next few chapters is foremost to show you how a Rails application should be developed. The only differences here are that the application will be backed by a database accessed through JDBC, rather than through a native database driver, and that we'll run it with JRuby instead of Ruby.