CHAPTER 1

Introducing the Rails Framework

Rails is a web application framework for the Ruby programming language. Rails is well thought out and practical. It will help you build powerful web sites quickly, with code that's clean and easy to maintain.

The goal of this book is to give you a thorough and complete understanding of how to build dynamic web applications with Rails. This means more than just showing you how to use the specific features and facilities of the framework, and more than just a working knowledge of the Ruby language. Rails is quite a bit more than just another tool. It represents a way of thinking. To completely understand Rails, it's essential that you know about its underpinnings, its culture and aesthetics, and its philosophy on web development.

If you haven't heard it already, you're sure to notice the phrase "the Rails way" cropping up every now and again. It echoes a familiar phrase that has been floating around the Ruby community for a number of years: "the Ruby way." The Rails way is usually the easiest way—the path of least resistance, if you will. This isn't to say that you can't do things your way, nor is it meant to suggest that the framework is constraining. It simply means that if you choose to go off the beaten path, don't expect Rails to make it easy for you. If you've been around the UNIX circle for any length of time, you might think that this idea bears resemblance to the UNIX mantra: "Do the simplest thing that could possibly work." You would be right. This chapter's aim is to introduce to you the Rails way.

The Rise and Rise of the Web Application

Web applications are increasingly gaining in importance. As our world becomes more and more connected, more and more of what we do is on the web. We check our email on the web, and we do our banking on the web. We take courses, share photos, upload videos, manage projects, and connect with people all over the world from the comfort of our browsers. As our connections get faster, and as broadband adoption grows, web-based software, and similarly networked client/server applications, are poised to displace software distributed by more traditional (read, outdated) means.

As consumers, web-based software affords us greater convenience, allowing us to do more from more places. Web-based software works on every platform that supports a web browser (which is to say, all of them), and there's nothing to install or download. And if Google's stock value is any indication, web applications are really taking off. In fact, the change in the web has been so dramatic in recent years that its current incarnation has been dubbed Web 2.0. All over the world, people are waking up to the new web and the beauty of being web-based. From email and calendars, photos and videos, to bookmarking, banking, and bidding, we are living increasingly inside the browser.

Due to the ease of distribution, the pace of change in the web-based software market is fast. Unlike traditional software, which must be installed on each individual computer, changes in web applications can be delivered quickly and features can be added incrementally. There's no need to spend months or years perfecting the final version or getting in all the features before the launch date. Instead of spending months on research and development, you can go into production early and refine in the wild, even without all the features in place.

Can you imagine having a million CDs pressed and shipped, only to find a bug in your software as the FedEx truck is disappearing into the sunset? That would be an expensive mistake! Software distributed this way takes notoriously long to get out the door because before a company ships a product, it needs to be sure the software is bug-free. Of course, we all know there's no such thing as bug-free software. And web applications are themselves not immune to these unintended features. But with a web application, bug fixes are easy to deploy.

When a fix is pushed to the server hosting the web application, all users get the benefit from the update at the same time, usually without any interruption in service. That's a level of quality assurance you just can't offer with store-bought software. There are no service packs to tirelessly distribute and no critical updates to install. A fix is often only a browser refresh away. And as a side benefit, instead of spending large amounts of money and resources on packaging and distribution, software developers are free to spend more time on quality and innovation.

Web-based software has the following advantages:

- Easier to distribute
- Easier to deploy
- Easier to maintain
- Platform-independent
- Accessible from anywhere

The Web Is Not Perfect

As great a platform as the web is, it's also fraught with constraints. One of the biggest problems is the browser itself. When it comes to browsers, there are several contenders, all of which have a slightly different take on how to display the contents of a web page. While there is a movement toward unification, and the state of standards-compliance among browsers is steadily improving, there is still a lot left to be desired. Even today, it's nearly impossible to achieve 100% cross-browser compatibility. Something that works in Internet Explorer won't necessarily work in Firefox, and vice versa. This lack of uniformity makes it difficult for developers to create truly cross-platform applications, as well as harder for users to work in their browser of choice.

Browser issues aside, perhaps the biggest constraint facing web development is its inherent complexity. A typical web application has dozens of moving parts: protocols and ports, the HTML and CSS, the database and the server, the designer and the developer, and a multitude of other players, all conspiring toward complexity.

But despite these problems, the new focus on the web as a platform has meant that the field of web development is evolving rapidly and quickly overcoming obstacles. As it continues to mature, the tools and processes that have long been commonplace in traditional, client-side software development are beginning to make their way into the world of web development.

The Good Web Framework

Among the tools making their way into the world of web development is the framework. A *framework* is a collection of libraries and tools intended to facilitate development. Designed with productivity in mind, a good framework will provide you with a basic but complete infrastructure on top of which to build an application.

Having a good framework is a lot like having a chunk of your application already written for you. Instead of having to start from scratch, you start with the foundation already in place. If there is a community of developers using the same framework, you have a community of support when you need it. You also have greater assurance that the foundation you're building upon is less prone to pesky bugs and vulnerabilities, which can slow down the development process.

A good web framework can be described as follows:

- **Full stack:** Everything you need for building complete applications should be included in the box. Having to install various libraries or configure multiple components is a drag. The different layers should fit together seamlessly.

- **Open source:** A framework should be open source, preferably licensed under a liberal, free-as-in-free license, like BSD or MIT.

- **Cross-platform:** A good framework will be platform-independent. The platform on which you decide to work is one of personal choice. Your framework should remain as neutral as possible.

A good web framework will provide you with the following:

- **A place for everything:** Structure and convention drive a good framework. In other words, unless a framework offers a good structure and a practical set of conventions, it's not a very good framework. The idea is that everything should have a proper place within the system. This eliminates the guesswork and increases productivity.
- **A database abstraction layer:** You shouldn't have to deal with the low-level details of database access, nor should you be constrained to a particular database application. A good framework will take care of most of the database grunt work for you, and it will work with almost any database out there.
- **A culture and aesthetic to help inform programming decisions:** Rather than seeing the structure imposed by a framework as constraining, see it as liberating. A good framework encodes its opinions, gently guiding the developer. Often, a difficult decision has been made for you by virtue of convention. The culture of the framework helps you make fewer menial decisions and helps you to focus on what matters most.

Enter Rails

Rails is a best-of-breed framework for building web applications. It's complete, open source, and cross-platform. It provides a powerful database abstraction layer called Active Record, which works with all popular database systems. It ships with a sensible set of defaults and provides a well-proven, multilayer system for organizing program files and concerns.

Above all, Rails is opinionated software. It has a philosophy on the art of web development that it takes very seriously. Fortunately, this philosophy is centered around beauty and productivity. You'll find that as you learn Rails, it actually makes writing web applications pleasurable.

Originally created by David Heinemeier Hansson, Rails first took shape in the form of a wiki-wiki application called Instiki. The first version of what is now the Rails framework was actually extracted from a real-world, working application: Basecamp, by 37signals. The Rails creators took away all the Basecamp-specific parts, and what remained was Rails.

Because it was extracted from a real application and not built as an ivory tower exercise, Rails is practical and free of needless features. Its goal as a framework is to solve 80% of the problems that occur in web development, assuming that the remaining 20% are the problems that are truly unique to the application's domain. It might be surprising that as much as 80% of the code in an application is infrastructure, but it's not as far-fetched as it sounds. Consider all the work that's involved in application construction, from directory structure and naming conventions, to the database abstraction layer and the maintenance of state.

You'll see that Rails has specific ideas about directory structure, file naming, data structures, method arguments, and, well, nearly everything. When you write a Rails application, you're expected to follow the conventions that have been laid out for you. Instead of focusing on the details of knitting the application together, you get to focus on the 20% that really matters.

Rails Is Ruby

There are a lot of programming languages out there. You've probably heard of many of them. C, C#, Lisp, Java, Smalltalk, PHP, and Python are popular choices. And then there are others you've probably never heard of: Haskel, IO, and maybe even Ruby. Like the others, Ruby is a programming language. You use it to write computer programs, including, but certainly not limited to, web applications.

Before Rails came along, not many people were writing web applications with Ruby. Other languages like PHP and ASP were the dominant players in the field, and a large part of the web is powered by them. The fact that Rails uses Ruby is significant because Ruby is considerably more powerful that either PHP or ASP in terms of its abilities as a programming language. This is largely another symptom of the web's maturity. Now that it's attracting a larger audience, more powerful languages and tools are falling into the fold.

Ruby is a key part of the success of Rails. Rails actually uses Ruby to create what's called a *domain-specific language*, or a DSL. Here, the domain is that of web development, and when you're working in Rails, it's almost as though you're writing in a language that was specifically designed to construct web applications—a language with its own set of rules and grammar. Rails does this so well that it's sometimes easy to forget that you're actually writing Ruby code. This is a testimony to Ruby's power, and Rails takes full advantage of Ruby's expressiveness to create a truly beautiful environment.

For many developers, Rails is their introduction to Ruby, a language whose following before Rails was admittedly small at best, at least in the west. While Ruby had been steadily coming to the attention of programmers outside Japan, the Rails framework is what brought Ruby to the mainstream.

Invented by Yukihiro Matsumoto in 1994, it's a wonder Ruby remained shrouded in obscurity for as long as it did. As far as programming languages go, Ruby is among the most beautiful. Interpreted and object-oriented, elegant and expressive, Ruby is truly a joy to work with. A large part of Rails's grace owes to Ruby and to the culture and aesthetics that permeate the Ruby community. As you begin to work with the framework, you'll quickly learn that Ruby, like Rails, is rich with idioms and conventions, all of which make for an enjoyable, productive programming environment.

In summary, Ruby can be described as follows:

- An interpreted, object-oriented scripting language
- Elegant, concise syntax
- Powerful metaprogramming features
- Well suited as a host language for creating DSLs

Appendix A of this book includes a complete Ruby primer. If you want to get a feel for what Ruby looks like now, go ahead and skip to that appendix and take a look. Don't worry if Ruby seems a little unconventional at first. You'll find it quite readable, even if you're not a programmer. It's safe to follow along in this book learning it as you go, referencing the appendix when you need clarification. If you're looking for a more in-depth guide, Peter Cooper has written a fabulous book titled *Beginning Ruby: From Novice to Professional* (Apress, 2007). You'll also find the Ruby community more than helpful in your pursuit of the language. Be sure to visit `http://ruby-lang.org` for a wealth of Ruby-related resources.

Rails Encourages Agility

Web applications are not traditionally known for agility. They have a reputation of being difficult to work with and a nightmare to maintain. It is perhaps in response to this diagnosis that Rails came on to the scene, helping to usher in a movement toward agile programming methodologies in web development. Rails advocates and assists in the achievement of the following basic principles when developing software:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

So reads the Agile Manifesto[1], the result of a discussion between 17 prominent figures (including Dave Thomas, Andy Hunt, and Martin Fowler) in the field of what was then called "lightweight methodologies" for software development. Today, the Agile Manifesto[1] is widely regarded as the canonical definition of agile development.

1. `http://agilemanifesto.org`

Rails was designed with agility in mind, and it takes each of the agile principles to heart, almost obsessively. With Rails, you can respond to the needs of customers quickly and easily, and Rails works well during collaborative development. Rails accomplishes this by adhering to its own set of principles, all of which help make agile development possible.

Dave Thomas and Andy Hunt's seminal work on the craft of programming, *The Pragmatic Programmer* (Addison-Wesley, 1999) reads almost like a roadmap for Rails. Rails follows the *don't repeat yourself* (DRY) principle, the concepts of rapid prototyping, and the *you ain't gonna need it* (YAGNI) philosophy. Keeping important data in plain text, using convention over configuration, bridging the gap between customer and programmer, and above all, postponing decisions in anticipation of change are institutionalized in Rails. These are some of the reasons that Rails is such an apt tool for agile development, and it's no wonder that one of the earliest supporters of Rails was Dave Thomas himself.

In the sections that follow, we're going to take a tour through some of Rails mantras, and in doing so, demonstrate just how well suited Rails is for agile development. While we want to avoid getting too philosophical, some of these points are essential to grasping what makes Rails so important.

Less Software

One of the central tenets of Rails philosophy is the notion of *less software*. What does less software mean? It means using convention over configuration, writing less code, and doing away with things that needlessly add to the complexity of a system. In short, less software means less code, less complexity, and fewer bugs.

Convention Over Configuration

Convention over configuration means that the programmer needs to define only configuration that is unconventional.

Programming is all about making decisions. If you were to write a system from scratch, without the aid of Rails, you would have a lot of decisions to make: how to organize your files, what naming conventions to adopt, and how to handle database access are only a few. If you decided to use a database abstraction layer, you would need to sit down and write it, or at least find an open source implementation that suits your needs. You would need to do all this before you even got down to the business of modeling your domain.

Rails lets you get started right away, by encompassing a set of intelligent decisions about how your program should work, alleviating the amount of low-level decision-making you need to do up front. As a result, you can focus on the problems you're trying to solve and get the job done quicker.

Rails ships with almost no configuration files. If you're used to other frameworks, this fact might surprise you. If you've never used a framework before, you should be surprised. In some cases, configuring a framework is nearly half the work.

Instead of configuration, Rails relies on common structures and naming conventions, all of which employ the often-cited *principle of least surprise* (POLS). Things behave in a predictable, easy-to-decipher way. There are intelligent defaults for nearly every aspect of the framework, relieving you, the developer, from having to explicitly tell the framework how to behave. This isn't to say that you can't tell Rails how to behave. In fact, most behaviors can be customized to your liking and to suit your particular needs. But you'll get the most mileage and productivity out of the defaults, and Rails is all too willing to encourage you to accept the defaults and move on to solving more interesting problems.

While you can manipulate most things in the Rails setup and environment, the more you accept the defaults, the faster you can develop applications and predict how they will work. The speed with which you can develop without having to do any explicit configuration is one of the key reasons why Rails works so well. If you put your files in the right place and name them according to the right conventions, things will *just work*. If you're willing to agree to the defaults, you generally have less code to write.

The reason Rails does this comes back to the idea of less software. Less software means making fewer low-level decisions, which will make your life as a web developer a whole lot easier. And easier is a good thing.

Don't Repeat Yourself

Rails is big on the DRY principle. DRY stands for *don't repeat yourself*, a principle that states information in a system should be expressed in only one place.

For example, consider database configuration parameters. When you connect to a database, you generally need credentials, such as a username, password, and the name of the database you want to work with. It might seem acceptable to include this connection information with each database query, and would surely hold up fine if you were making only one or two connections. But as soon as you need to make more than a few connections, you would end up with a lot of instances of that username and password littered throughout your code. Then if your username and password for the database changed, you would have a lot of finding and replacing to do. It would be a much better idea to keep the connection information in a single file, referencing it as necessary. That way, if the credentials should change, you need to modify only a single file. That's what the DRY principle is all about.

The more duplication there is in a system, the more room there is for bugs to hide. The more places that the same information resides, the more that has to be modified when a change is required, and the harder it becomes to track these changes.

Rails is organized in such a way as to remain as DRY as possible. You generally specify information in a single place, and move on to better things.

Rails Is Opinionated Software

Frameworks encode opinions. It should come as no surprise then that Rails has strong opinions about how your application should be constructed. When you're working on a Rails application, those opinions are imposed on you, whether you're aware of it or not. One of the ways that Rails makes its voice heard is by gently (sometimes, forcefully) nudging you in the right direction. We've already mentioned this form of encouragement when we talked about convention over configuration. You're invited to do the right thing by virtue of the fact that doing the wrong thing is often more difficult.

Ruby is known for making certain programmatic constructs look more natural by way of what's called *syntactic sugar*. Syntactic sugar means that the syntax for something is altered in such a way as to make it appear more natural, even though it behaves the same way. Things that are syntactically correct but otherwise look awkward when typed are often treated to syntactic sugar.

Rails has popularized the term *syntactic vinegar*. Syntactic vinegar is the exact opposite of syntactic sugar: awkward programmatic constructs are discouraged by making their syntax look sour. When you write a snippet of code that looks bad, chances are it is bad. Rails is good at making the right thing obvious by virtue of its beauty, and the wrong thing equally obvious by virtue of ugliness.

You can see Rails's opinion in the things it does automatically, the ways it encourages you to do the right thing, and the conventions it asks you to accept. You'll find that Rails has an opinion on nearly everything about web application construction: how you should name your database tables, how you should name your fields, which database and server software to use, how to scale your application, what you're going to need, and what is a vestige of web development's past. If you subscribe to its world view, you'll probably get along with Rails quite well.

Like a programming language, a framework needs to be something you're comfortable with—something that reflects your personal style and mode of working. It's often said in the Rails community that if you're getting pushback from Rails, it's probably because you haven't experienced enough pain from doing web development the old-school way. This isn't meant to deter developers, rather it's meant to say that in order to truly appreciate Rails, you might need a history lesson in the technologies from whose ashes Rails has risen; sometimes until you've experienced the hurt, you can't appreciate the cure.

Rails Is Open Source

The Rails culture is steeped in open source tradition. The Rails source code is, of course, open. And it's significant that Rails is licensed under the MIT license, arguably one of the most "free" software licenses in existence.

Rails also advocates the use of open source tools, and encourages the collaborative spirit of open source. The code that makes up Rails is 100% free and can be downloaded, modified, and redistributed by anyone at anytime. Moreover, anyone is free to submit patches for bugs or features, and hundreds of people from all over the world have contributed to the project over the past two years.

You'll probably notice that a lot of Rails developers use Macs. The Mac is clearly the preferred platform of the core Rails team, and you'll find in general that most Rails developers are using UNIX variants (of which Mac OS X is one). The UNIX operating system is hailed by hackers and used almost exclusively among the hacker elite. There are several reasons for this, not least of which is the fact that UNIX is a well-tested and proven operating system, forged in an open source ecosystem, with contributions from some of the smartest programmers on the planet. Having been born in the 1970s, at this late stage, the UNIX operating system has evolved into lean and powerful example of open source craftsmanship. The beauty, simplicity, and singularity of purpose of UNIX is not lost on the creators of Rails.

Although there is perhaps a marked bias towards UNIX variants when it comes to Rails developers, make no mistake, Rails is truly cross-platform. It doesn't matter which operating system you choose, you'll be able to use Rails on it. Rails doesn't require any special editor or IDE to write code. Any text editor will do just fine, as long as it can save files in plain text. The Rails package even includes a built-in, stand-alone web server called WEBrick, so you don't need to worry about installing and configuring a web server for your platform. When you want to run your Rails application in development mode, simply start up the built-in server and open your web browser. Why should it be any more difficult than that?

The next chapter will take you step by step through the relatively painless procedure of installing Rails and getting it running on your system. But before we go there, and before you start writing your first application, we would like to talk a bit about how the Rails framework is architected. This is important because, as you'll see in a minute, it has a lot to do with how you organize your files and where you put them. Rails is actually a subset of a category of frameworks that are named for the way in which they divide the concerns of program design: the Model-View-Controller (MVC) pattern. Not surprisingly, the MVC pattern is the topic of our next section.

The MVC Pattern

Rails employs a time-honored and well-established architectural pattern that advocates a division of application logic and labor into three distinct categories: the model, view, and controller. In the MVC pattern, the model represents the data, the view represents the user interface, and the controller directs all the action. The real power lies in the combination of the MVC layers, which is something that Rails handles for you. Place your code in the right place and follow the naming conventions, and everything should fall into place.

Each part of the MVC—the model, view, and controller—is a separate entity, capable of being engineered and tested in isolation. A change to a model need not affect the views; likewise, a change to a view should have no effect on the model. This means that changes in an MVC application tend to be localized and low impact, easing the pain of maintenance considerably, while increasing the level of reusability among components.

Contrast this to the situation that occurs in a highly coupled application that mixes data access, business logic, and presentation code (PHP, we're looking at you). Some folks call this spaghetti code because of its striking resemblance to a tangled mess. In such systems, duplication is common, and even small changes can produce large ripple effects. MVC was designed to help solve this problem.

MVC isn't the only design pattern for web applications, but it does happen to be the one that Rails has chosen to implement. And it turns out that it works great for web development. By separating concerns into different layers, changes to one of them don't have an impact on the others, resulting in faster development cycles and easier maintenance.

The MVC Cycle

Although MVC comes in different flavors, control flow generally works as follows (see Figure 1-1):

1. The user interacts with the interface and triggers an event (for example, submits a registration form).
2. The controller receives the input from the interface (for example, the submitted form data).
3. The controller accesses the model, often updating it in some way (for example, by creating a new user with the form data).
4. The controller invokes a view that renders an updated interface (for example, a "welcome" screen).
5. The interface waits for further interaction from the user, and the cycle repeats.

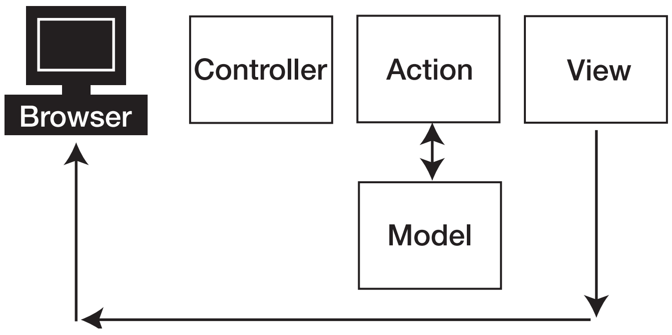

Figure 1-1. *The MVC cycle*

If the whole MVC concept sounds a little involved at first, don't worry. While entire books could be written on this pattern, and people will argue over its purest implementation for all time, you'll find that it's really quite easy to grasp, especially the way Rails does MVC.

Next, we'll take a quick tour through each letter in the MVC, and then describe how Rails handles it.

The Layers of MVC

The three layers of the MVC pattern work together as follows:

- **Model:** The information the application works with.
- **View:** The visual representation of the user interface.
- **Controller:** The director of interaction between the model and the view.

Models

In Rails, the model layer represents the database. While we call the entire layer the model, Rails applications are usually made up of several individual models, each of which (usually) maps to a database table. For example, a model called `User` would map to a table called `users`. The `User` model assumes responsibility for all access to the `users` table in the database, including creating, reading, updating, and deleting rows. So, if you wanted to work with the table and, say, search for someone by name, you would do so through the model, like this:

```
User.find_by_name('Linus')
```

This snippet, while very basic, would search the `users` table for the first row with the value `Linus` in the `name` column and return the results. To achieve this, Rails uses its built-in database abstraction layer, Active Record. Active Record is a powerful library, so needless to say, this is only a small portion of what you can do with it.

Chapters 4 and 5 will give you an in-depth understanding of Active Record and what you can expect from it. For the time being, the important thing to remember is that models represent data. All rules for data access, associations, validations, calculations, and routines that should be executed before and after save, update, or destroy operations are neatly encapsulated in the model. Your application's "world" is populated with Active Record objects: single ones, lists of them, new ones, and old ones. And Active Record lets you use Ruby language constructs to manipulate all of them, meaning you get to stick to one language for your entire application.

Controllers

We're going to rearrange the MVC acronym a bit here and put the *C* before the *V*. As you'll see in a minute, in Rails, controllers are responsible for rendering views, so it makes sense that we should introduce them first.

Controllers are the conductors of an MVC application. In Rails, controllers accept requests from the outside world, perform the necessary processing, and then pass control to the view layer to display the results. It's the controller's job to do the fielding of web requests, like processing server variables and form data, asking the model for information, and sending information back to the model to be saved in the database. While it may be a gross oversimplification, controllers generally perform a request from the user to create, read, update, or delete a model object. You'll see these words a lot in the context of Rails, most often abbreviated as CRUD. In response to a request, the controller typically performs a CRUD operation on the model, sets up variables to be used in the view, and then proceeds to render or redirect to another action once processing is complete.

Controllers typically manage a single area of an application. For example, in a recipe application, you would probably have a controller just for managing recipes. Inside the recipes controller, you would define what are called *actions*. Actions describe what a controller can do. If you wanted to be able to create, read, update, and delete recipes, you would create appropriately named actions in the recipes controller. A simple recipes controller might look something like this:

```
class RecipesController < ApplicationController
  def index
    # logic to list all recipes
  end
  def show
    # logic to show a particular recipe
  end

  def create
    # logic to create a new recipe
  end

  def update
    # logic to update a particular recipe
  end

  def destroy
    # logic to delete a particular recipe
  end
end
```

Of course, if you wanted this controller to do anything, you would need to put some instructions inside each action. When a request comes into your controller, it uses a URL parameter to identify the action to execute, and when it's done, it sends a response to the browser. The response is what we'll look at next.

Views

The view layer in the MVC is the one that makes up the visible part of the application. In Rails, views are the templates that (most of the time) contain HTML markup to be rendered in a browser. It's important to note that views are meant to be free of all but the simplest programming logic. The idea is that any direct interaction with the model layer should be delegated to the controller layer so as to keep the view clean and decoupled from the application's business logic.

Generally, views have the responsibility of formatting and presenting model objects for output on the screen, as well as providing the forms and input boxes that accept model data, such as a login box with a username and password, or a registration form. Rails also provides the convenience of a comprehensive set of helpers that make connecting models and views easier, such as being able to prepopulate a form with information from the database, or the ability to display error messages if a record fails any validation rules, such as required fields.

You're sure to hear it eventually if you hang out in Rails circles, but a lot of folks consider the interface to *be* the software. We agree with them. The idea is that since the interface is all the user sees, it's the most important part. Whatever the software might be doing behind the scenes, the only parts that an end user can relate to are the parts they see and interact with. The MVC pattern helps by keeping programming logic out of the view. With this strategy in place, programmers get to deal with code, and designers get to deal with HTML. Having a clean environment in which to design the HTML means better interfaces and better software.

The Libraries That Make Up Rails

Rails is a collection of libraries, each with a specialized task. Assembled together, these individual libraries make up the Rails framework. Of the several libraries that compose Rails, three map directly to the MVC pattern:

- **ActiveRecord:** A library that handles database abstraction and interaction.
- **ActionView:** A templating system that generates the HTML documents that the visitor gets back as the result of a request to a Rails application.
- **ActionController:** A library for manipulating both application flow and the data coming from the database on its way to being displayed in a view.

These libraries can be used independently of Rails and of each other. Together, they make up the Rails Model-View-Controller development stack. Since Rails is a “full-stack” framework, all the components are integrated, so you do not need to set up bridges between them manually.

Rails Is No Silver Bullet

There is no question that Rails offers web developers a lot of benefits. In fact, after having used Rails, it’s hard to imagine going back to web development without it. Fortunately, it looks like Rails will be around for a long time, so there’s no need to worry. But it brings us to an important point.

As much as we’ve touted the benefits of Rails, it’s important that you realize that there are no silver bullets in software design. No matter how good Rails gets, it will never be all things to all people, and it will never solve all problems. Most important, Rails will never replace the role of the developer. Its purpose is merely to assist developers in getting their job done. Impressive as it is, Rails is merely a tool, which when used well can yield amazing results. It is our hope that as you continue to read this book and learn how to use Rails, you’ll be able to leverage its strength to deliver creative and high-quality web-based software.

Summary

This chapter provided an introductory overview of the Rails landscape, from the growing importance of web applications to the history, philosophy, evolution, and architecture of the framework. You learned about the features of Rails that make it ideally suited for agile development, including the concepts of less software, convention over configuration, and DRY. Finally, you learned the basics of the MVC pattern, and received a primer on how Rails does MVC.

With all this information under your belt, it's safe to say you're ready to get up and running with Rails. The next chapter will walk you through Rails installation, so you can try it for yourself and see what all the fuss is about. You'll be up and running with Rails in no time.