Extracted from:

Rails for PHP Developers

Chapter 4

Modeling the Domain

Now that we have seen some high-level differences between PHP and Rails in Part I, it's time to put our experiences into action and get hands-on by building a Rails application. In this part, we'll build a Rails app from start to finish; along the way we will see in context how building an application in Rails is different from how we'd go about the task using PHP.

We'll offer an imaginary scenario here of a typical application development situation. Our friend Joe has called us with a plea. Joe is an experienced PHP programmer but has heard enough buzz about Rails to finally pique his interest. He has started a new Rails user group in his area but just doesn't have the time to create a decent website for it. He wants this group to be a success, and he knows that a website with only the date and location of meetings just won't cut it. He has asked for our help in creating an application to help plan and organize the group meetings. Since Joe is a good friend of ours, we'll help him build a killer app for his group.

The application we build will cover many of the features that Rails offers and will help us get a good idea of how to build a typical application using Rails. Creating a user group site will be a great introduction to hands-on coding with Rails because it contains enough objects to exercise the use of various ActiveRecord methods and associations. Giving Joe the ability to manage the application's data will require us to build a simple authentication system. Finally, all of this will need to be nicely wrapped up in a presentable public interface. When the application is finished, you should have a solid understanding of how Rails code is organized and have a good grasp on the practical uses of the various Rails components. Meanwhile, we'll continue to relate these development practices to those typically used in PHP.

Requirement	Feature
1. Describe the group	Display prominent description of group goals.
2. Inform of meeting details	Present information on past and future meetings.
3. Share coder knowledge	Display presentations and encourage members to present.
4. Show off coder projects	Create list of members with breif description of their work.
5. Get people talking	Use a mailing list to get people together outside of meetings.
6. Keep information current	Equip application with an easy-to-use administrative interface.

Figure 4.1: Requirements and features

To follow along as we build the application, you'll probably want to download the code examples for this application. The example source code is available online.[1]

4.1 Defining Requirements

We'll start the same way we might for any application, whether it be PHP or otherwise. We need to figure out the goals we want to accomplish and how our application can help us achieve them. Joe is as opinionated as any client and comes up with a solid list of requirements, as shown in Figure 4.1. We've taken this a little further, and we've assigned a feature to each of his requirements.

Our next step is to create some simple mock-ups of how the application might look. The interface needs to manage meetings, presentations, and users in our application. Joe tells us that he wants the meetings to include a date, a location, and a short description. He also wants to be able to add presentations to each meeting along with the person presenting.

1. http://www.pragprog.com/titles/ndphpr/source_code

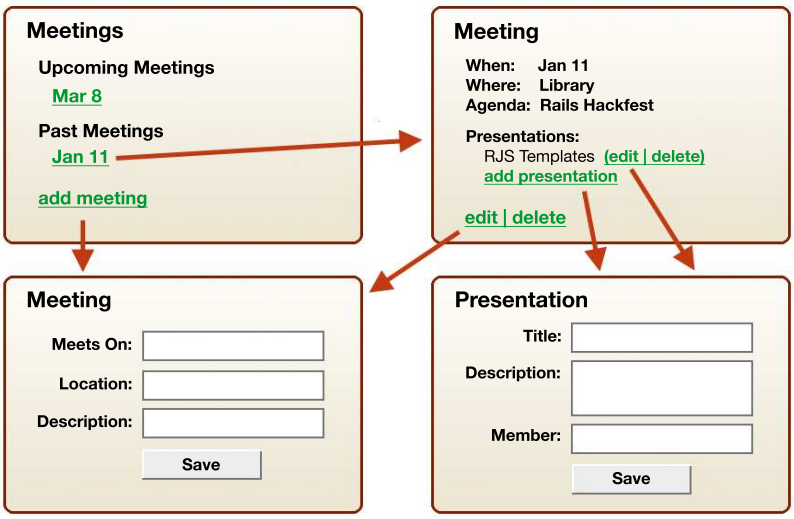

Figure 4.2: Meetings page flow

Let's first concentrate on how we will need to manage these resources, and then we can begin coding them. To map out how we want this to work, we sit down with Joe to create some page flow diagrams. The drawing in Figure 4.2 shows a series of pages representing a typical web application. This includes the display of our meetings along with the ability to add, edit, and delete meetings and their associated presentations.

Our member pages (shown in Figure 4.3, on the next page) are much simpler, consisting of the ability to view and change user profiles. These page flow diagrams should provide us with enough material to start writing code.

At this point, we have a fair idea about how the application will look. If we were building this application in PHP, it would be tempting to simply make a PHP file for each one of these pages. First, we'd spend a bit of time working out the directory structure of our little application, figure out how to connect the PHP files, and probably gather up our favorite libraries from PEAR and other repositories to do tasks such as form handling. If we had chosen some PHP framework, we'd have fewer decisions, but we'd also have to start with the huge decision of which of the dozens of PHP frameworks to choose.

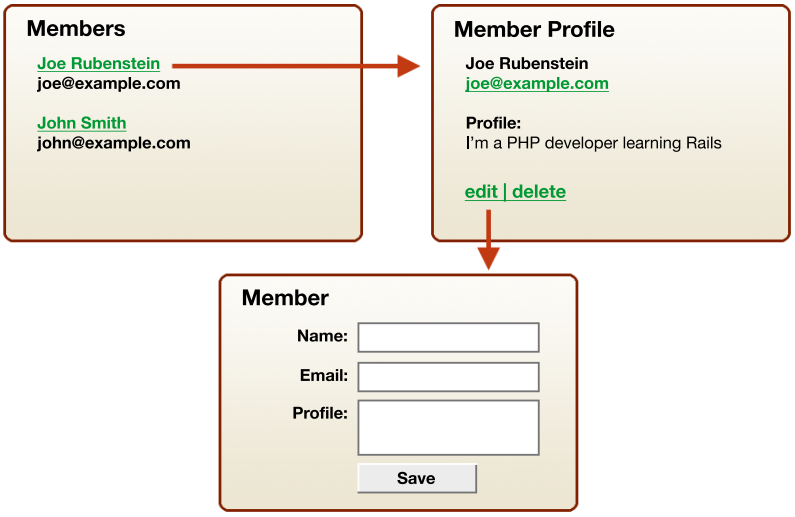

Figure 4.3: Users page flow

We could then dip into the home page and start fleshing it out in real PHP code and have a nonfunctioning mock-up of the home page to show Joe a couple of hours later. Joe would probably be impressed we threw it together so quickly. Once that was out of the way, we could start building the other pages and some code to deal with the database.

This isn't PHP, though; it's Rails. One of the big wins of adopting Rails is that it frees your mind of almost all the up-front decisions such as where to put things or what libraries to use. For our application, we'll do it "the Rails way" and follow whatever methods and tools that Rails has given us to use. By making this conscious decision to worry less about the innards of our application, we can simply concentrate on solving Joe's problems and trust that Rails will have the facilities available to let us do that efficiently.

We called this chapter *Modeling the Domain* instead of *Building the Website* because a Rails application has a different focus and workflow than a usual PHP website. Where plain PHP lets us start any place we'd like and build whatever we'd like, Rails has a strongly defined workflow for us to follow. That workflow starts by making us examine our problem domain—Joe's user group meetings—and modeling the data and interactions around that.

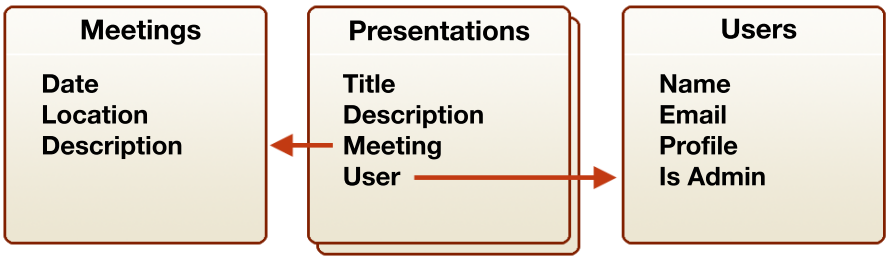

Figure 4.4: Modeling the application data

Through our interviews with Joe and creating these page flow diagrams, we should be able to identify the basic data that our application is dealing with and determine what our domain model will look like. Looking through our diagrams, let's create a list of the data we'll need to represent in our application. It seems right now that we have three sets of data to represent the resources in our application. If we take a look at Figure 4.4, we see that each meeting needs an association with one or more presentations, and each presentation will be associated with a user. Before we actually model this data into Ruby classes in our application, it's important to learn a little more about Rails' opinion of databases.

4.2 Using the Database

Rails rejects the idea of putting business logic in the database in the form of in-database constraints, referential integrity, or stored procedures. While the database is seen as a way to store relational data, all business logic for that data belongs in the domain model of our application.

If you've primarily worked with MySQL in the past, this is a pretty standard approach. Although MySQL supports many these features, they are not terribly commonplace in PHP applications that use MySQL.

If you're accustomed to using things such as referential integrity and stored procedures in your databases, this approach may seem ignorant or controversial at the very least. There have been many discussions

about this topic in the past, and Rails' rejection of these concepts is not likely to change. Rails focuses on using the database as an "application database" and not as an "integration database." It expects your application to be the single point of interaction with the database.

Referential Integrity

Referential integrity in the form of database-defined foreign keys is a hot topic. There are still many developers in the Ruby community who think this is an oversight and that these constraints should have better native support in Rails. One of the great aspects of Rails is the plug-in environment that allows us to disagree with the Rails core by simply installing a plug-in to add the features we want. Although using database foreign keys is unconventional and discouraged in a typical Rails application, there is a plug-in to help make them less painful to use; it's available on the Red Hill Consulting website.[2]

You can find more information about installing Rails plug-ins within your application in Section 13.13, *Rails Plug-Ins*, on page 398.

Using a Single Primary Key

Another intensely debated opinion in Rails is the rejection of composite keys in favor of all tables using a single primary key named id. The core team believes the cost of supporting composite keys outweighs the benefits. The cost in this case is the immense and ugly increase in the complexity of the Rails code. The ripple effect of supporting composite keys would have too many implications in the simplicity and beauty of Rails code.

Another reason is that there is usually not a tangible benefit to using composite keys over a single unique key. This is even truer when we're using a simplified Rails syntax for performing much of our database interactions. Like support for foreign key constraints, there is a Rails plug-in to add composite key support if your application requires them. The composite keys plug-in was written by Dr. Nic Williams; you can find it in RubyForge.[3]

Stored Procedures

Stored procedures are another database feature that is not recommended in Rails applications. Rails is attached to the idea of having a

2. http://www.redhillonrails.org/
3. http://compositekeys.rubyforge.org/

single layer of domain logic and complexity and having that logic written in Ruby. The typical need for stored procedures is in heavy "integration database–style" environments where multiple applications and people need to interact with a single database. Rails favors using web services to talk to the integration database through the Rails application itself.

Avoiding stored procedures generally makes it easier to keep revision history on your domain logic and makes application code easier to unit test. We realize that not all organizations have a choice of avoiding stored procedures, especially in an Oracle or SQL Server environment. There is a page that further details working with stored procedures on the Rails wiki.[4]

Model and Database Naming Conventions

Coming from PHP, we know that different developers have vastly different PHP coding styles. Some developers like to use CamelCase names like getFoo(), while others prefer underscore names like get_foo(). PHP itself is a big mix of different styles, so it provides little guidance on how our code should look. Ruby, on the other hand, provides a solid foundation of standards. Features of the Ruby language even help enforce these standards. As a result, most Ruby code looks quite similar. This is great for us because it keeps code readable, and mixing code from different sources doesn't end up looking like a hodgepodge of different coding styles.

In every way, Rails is an extension of Ruby. Although Ruby provides guidance for how to name our classes and methods, Rails takes this further and even gives us conventions for naming database tables and columns. In PHP, there are no such rules, and many developers like it this way. This may require a little shift in thinking.

As we said earlier, Rails is largely about removing the burden of decision about how to structure all the little details of our applications. This allows us to focus more on our application itself and less on its gritty implementation details. By following the Rails conventions for naming things in the database, Rails will implicitly connect the database tables to their corresponding model objects without us needing to do any configuration to map them together. These conventions also help keep Rails applications easily readable.

4. http://wiki.rubyonrails.org/rails/pages/StoredProcedures

Model Naming Conventions	
Table:	`software_projects`
Class:	`SoftwareProject`
File:	`app/model/software_project.rb`
Test File:	`test/unit/software_project_test.rb`

Figure 4.5: Model naming conventions

While Ruby's well-defined conventions keep classes and methods in check, Rails' conventions keep application structure in check. This is a big help for application maintenance as well. If the next developer who maintains the application understands Rails and our application is built with all the Rails standards, then that developer will come in already having some understanding of the application.

By taking a look at Figure 4.5, we can see that database tables are expected to be named using a plural form of whatever we are storing, formatted with underscores. Each database table in our application will have an associated model, which is named using the singular form of the table name formatted using CamelCase. Finally, the filename will be based on the name of the model but in an underscore format.

This might seem like quite a few rules to follow when creating files and classes, but Rails does most of the work for you. When you run the generate script, Rails will automatically create the correct files and filenames according to conventions.

4.3 Creating the Application

Before we create our models, we need to set up a new Rails application. This means creating a new Rails project along with the MySQL database needed for development. We'll name this application user_group, and once again use MySQL for our database.

Joe Asks...

What's This "rake" Command?

We briefly mentioned Rake in Section 1.2, *The Components of Rails*, on page 21, and we'll use this tool often as part of our development process. Rake is "Ruby Make," a great system for gluing together all of your Ruby command-line tasks. Rails uses Rake extensively and even supports making your own special automation tasks!

There is also a somewhat similar system for PHP called Phing, but it has had limited adoption by PHP developers. By contrast, almost all Rails developers use and love Rake.

```
derek> cd work
work> rails -d mysql user_group
```

The development database for this project will be named, by Rails conventions, user_group_development. We'll use a Ruby tool called Rake to create this database for our application. Navigate to your application's root directory to run db:create.

```
work> cd user_group
user_group> rake db:create
(in /Users/derek/work/user_group)
```

If we want to change the username and password used to connect to this database, we need to edit our config/database.yml configuration, as discussed in Section 1.6, *Configuring the Database*, on page 29. Other than that, we should now have a new Rails application ready to go. Let's start WEBrick to get the application running on localhost.

```
user_group> ruby script/server
```

At this point in a typical PHP application, we would most likely create a relational database schema using a tool such as phpMyAdmin or even straight SQL create statements. Although we created our table with plain old SQL for our newsletter application in Chapter 1, *Getting Started with Rails*, on page 20, we'll take a different approach this time.

Rails *migrations* are a higher-level way of creating and modifying database tables using Ruby code instead of SQL. In this application, we'll create and modify all of our tables using migrations. A migration file will be created automatically for each model we generate.

4.4 Generating the First Model

We'll start constructing our application by creating a model to represent a user in our application. Our conventions state that for a table named users, we'll create a model named User. Let's use script/generate to create this.

```
user_group> ruby script/generate model User
exists  app/models/
exists  test/unit/
exists  test/fixtures/
create  app/models/user.rb
create  test/unit/user_test.rb
create  test/fixtures/users.yml
create  db/migrate
create  db/migrate/001_create_users.rb
```

We can now see all our naming conventions fall into place. The generate script has already created our model and test file. It has even created the migration file we'll be using to create the database table. If we open the model file app/model/user.rb, we can see that the class has correctly been named User. Likewise, opening the migration file db/migrate/001_create_users.rb shows us that we'll execute create_table :users, which is the plural underscore version of our model name.

You might be wondering how Rails determines the plural version of a word. Rails includes an inflection component to convert words to their plural or singular forms. To see this in action, we'll use another utility script that comes with Rails. This script starts an IRB session but also loads our Rails environment and code. This lets us interactively play with our application through the command line.

```
user_group> ruby script/console
Loading development environment
>> 'user'.pluralize
=> "users"
>> 'users'.singularize
=> "user"
```

We can see that the pluralize and singularize methods are added to all strings and that our user string is converting as expected. Most of the time, Rails' default inflection engine will handle our models as expected. Rails will successfully convert most irregular words as well but doesn't catch absolutely everything. Let's try something a little less expected.

```
>> 'bacon'.pluralize
=> "bacons"
```

Joe Asks...

Is Pluralization Worth the Hassle?

There have been many heated discussions over the pluralization conventions in Rails. The reason pluralization was added to Rails was to make the language more natural when referring to data and classes. A database table contains plural users, while a User class represents a single user. This follows in line with the principle of least surprise. It is possible to turn off pluralization by adding the following to your configuration block in config/environment.rb.

```
config.active_record.pluralize_table_names = false
```

This option is most useful for legacy database schemas that can't be changed to use Rails conventions. We highly suggest you stick with the conventional approach for all new projects.

If we were building a meaty application that needed a bacon table, we would want to refer to our bacon in plural as simply bacon. Rails seems to be adding a trailing *s* where it isn't warranted. We can fix this by adding custom inflection rules to an initializer that runs as Rails starts. Open the file config/initializes/inflections.rb, and at the bottom we'll see some sample code on how to modify inflections. Below the sample code, we'll add bacon as an uncountable word similar to fish and sheep, since the word remains the same in both singular and plural form.

```
Inflector.inflections do |inflect|
  inflect.uncountable %w( fish sheep bacon )
end
```

Now if we exit and reload our interactive console to reinitialize the Rails environment, the pluralization of *bacon* will behave as expected.

```
>> exit

user_group> ruby script/console
Loading development environment
>> 'bacon'.pluralize
=> "bacon"
```

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

Rails for PHP Developers Home Page
http://pragprog.com/titles/ndphpr
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
http://pragprog.com/updates
Be notified when updates and new books become available.

Join the Community
http://pragprog.com/community
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
http://pragprog.com/news
Check out the latest pragmatic developments in the news.

Buy the Book

If you liked this PDF, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: pragprog.com/titles/ndphpr.

Contact Us

Phone Orders:	1-800-699-PROG (+1 919 847 3884)
Online Orders:	www.pragprog.com/catalog
Customer Service:	orders@pragprog.com
Non-English Versions:	translations@pragprog.com
Pragmatic Teaching:	academic@pragprog.com
Author Proposals:	proposals@pragprog.com