10

.NET on Rails

* Know about routing
* Use ActiveRecord
* Use Controllers
* Create views

When I first used the Ruby on Rails framework for building a web application I got the feeling that that was the way web development should be done. The simple separation of concerns made it pretty easy to understand and learn. The learning curve was a lot smaller than the one I went through when I started using ASP.NET Webforms for example.

So far in this book we’ve mostly been looking at IronRuby from the perspective of using .NET technologies. We did use some ruby libraries in the process like RSpec for example. But in this last chapter we’re going to talk about creating a Rails application which is a web development framework built by the Ruby community and one of the big reasons the ruby language gained popularity in recent years.

Many of the concepts we’ll talk about will already be familiar from the ASP.NET MVC chapter, because the Rails framework is also a MVC framework. The concepts we’ll talk about are routing, models, controllers and views. There are a bunch of other books that explain you the rails framework in a lot more depth. We will just look at what’s involved in getting rails to run with IronRuby. Before we look at each of the components separately first we’ll have a discussion about the global Rails framework.

10.1 Rails from a bird’s eye view

Rails is a web development framework in the class of MVC Frameworks. Its first official release was by David Heinemeier Hansson in 2005. Rails is a full-stack MVC web framework that relies heavily on metaprogramming and makes it really easy for the developer to follow good programming standards. It got extracted from the work David Heinemeier did to develop the project management application Basecamp.

Rails advocates 2 principles at its core:

* Don’t Repeat Yourself (DRY)

The goal is that when you define something you only have to define it once and it will be available to you in the right places throughout the application thus avoiding a high degree of code duplication.

* Convention over Configuration (COC)

This means that the developer only has to provide configurations for the items that don’t adhere to the convention. When there is an ActiveRecord model User then the rails framework knows that that model maps to the database table users. However if your model User actually maps to the table logins then you need to provide that in the configuration for that model. Because it’s harder to not follow the convention it’s easier to follow the standards, which leads to easier maintainable applications.

Before we look at the different components presently available in Rails we should really discuss the architecture of a rails application.

10.1.1 Architecture of a Rails application

As we’ve discussed earlier Rails is a web framework that implements the Model-View-Controller pattern. We’ve already had a discussion on the different parts of the MVC pattern in chapter 7. But as a quick refresher we’d like to offer you a short summary.

The Model represents the data / domain model of your application. Almost all operations that concern data manipulation or business logic should be placed inside models. The view is often a visual representation of the data but in fact is your user interface to the application. That means a view can also be xml for other applications to use as the interface to your application. And the last part is the controller, which receives input and in an action it will delegate that input to the appropriate models for processing. The controller will then receive the data back from the models and choose an appropriate view for representing that data.

To see how this pattern is implemented by Rails let’s take a look at the data flow of a rails application.

10.1.2 Information flow through a Rails application

For convenience we’ll only be discussing the flow when a user requests a page from a rails application through a web browser.

1. The user requests a url in the browser
2. The web server receives the request and uses **routing** to map the request to a **controller** and an **action**.
3. A **controller** object gets created. On this object the method that corresponds to the **action** gets executed.
4. The controller action possibly preprocesses some of the **input** and delegates that input to one or more of the **model**s.
5. The controller possibly post processes some of the data it received from the **model**(s) and makes it available in the data for the view.
6. The controller decides which **view** to render.
7. The web server then sends that rendered html to the browser and the user has a new page to view.

I’m sure there is nothing there that rocks your world especially not after having read the chapter on ASP.NET MVC. But this leads us to the different components of the Rails framework.

10.1.3 Components of the Rails framework

The Rails framework consists of a couple of libraries that can be installed and used separately but are generally used together.

* ActiveSupport

This library contains a bunch of extensions to the ruby language. One of the things this library allows you to do is pluralization of nouns etc. This is a core part of the rails framework because this is how it maps model names to database table names for example.

* ActiveRecord

This is the built-in ORM tool for the rails framework and is used when you want to use a database with a ruby on rails application. This is probably the most common library to use when you’re defining your models.

* ActiveResource

This is a library that makes ruby on rails applications RESTful. It provides ORM mapping to REST resources. You can use this to allow different rails applications to talk to each other.

* ActionPack

This library is responsible for handling the request. It contains both the code for the controller (ActionController) and for the html views (ActionView).

* ActionMailer

This is the last library in the standard package. This library contains the code to allow you to send emails with ruby on rails. You have a similar template like when you’re using an html view but in addition you can define some of the headers of the mail in that template. It also contains the code to send emails based on those templates.

Now that we know about the different components we’d also like to mention that one of the strong points of rails is that you can override or extend the default behavior of an application through a system of plug-ins. There is a vast amount of plug-ins that have been written for Rails. They go from giving you a very easy way of implementing a state machine to complete view engines or authentication mechanisms. For me, personally, the plug-in system is one of the strongest points of the rails framework.

This takes care of the obligatory background information you may need before you can start using the framework. The rest of this chapter will be dedicated to guide you through creating the necessary pieces of the server side for our twitter sample. The first thing we’re going to need to do is to create a rails application framework.

10.2 The twitter server side

Back in chapter 4 we wrote a client for Twitter with WPF, in that chapter we said that the chapter on rails would have the server side implementation of that API so that you wouldn’t depend on having an Internet connection available to run the samples of that chapter.

In chapter 4 we didn’t implement the entire twitter API but instead just implemented the view for the list of tweets of you and your friends. We also added the possibility to post updates to twitter.

That means we only need to implement 2 models for our sample, the models in question are User and Status. User represents the information about users and also contains the login information. Status represents an update and contains the information to define a tweet.

We will see later how we’re going to create those models but for now it’s sufficient to know which ones we’re going to use and what their primary functions are.

The User model will take care of authenticating a user as well as capture the profile information about that user like its last location. The Status model on the other side captures the content and timestamp of a tweet. Who sent the tweet and to whom the tweet was in reply to as well as saving updates. Now that we have a general idea where we want to get to, lets start the journey!

10.3 Jumpstarting development

Before we can start development we have to make sure we’ve got the rails framework installed on our machine. In this section we’ll first talk about getting rails installed and ready to use with IronRuby. Next we’ll look at generating a rails application skeleton.

10.3.1 Installing Rails

Rails is distributed as a gem and when we use the regular ruby version we can install it by calling executing igem install rails at the command line.

This should download and install rails in our MRI installation. By setting the GEM\_PATH environment variable to the first value you get when you execute the gem env command. In my case that is c:\ruby\lib\ruby\gems\1.8. So in powershell I execute the command $env:gem\_path=”c:\ruby\lib\ruby\gems\1.8” from now on we can use the gems installed for the MRI installation from within IronRuby. And we’re all ready to generate the application skeleton.

10.3.2 Generating the application skeleton

Because rails is all about conventions there is a fixed directory structure that needs to be created on your machine for a new application. It also populates some of those directories with files, like the files for the prototype JavaScript library.

Listing 10.1 shows the series of commands I entered on my mac to generate the application skeleton for our twitter clone.

Listing 10.1 Generating the application skeleton

+ivan@ivan-mbp:~

» cd projects/ironruby\_book/

+ivan@ivan-mbp:~/projects/ironruby\_book

» mkdir twitter\_sample

+ivan@ivan-mbp:~/projects/ironruby\_book

» cd twitter\_sample/

+ivan@ivan-mbp:~/projects/ironruby\_book/twitter\_sample

» rails twitter

create

create app/controllers

create app/helpers

create app/models

create app/views/layouts

create config/environments

create config/initializers

create db

create doc

create lib

create lib/tasks

create log

create public/images

create public/javascripts

create public/stylesheets

create script/performance

create script/process

create test/fixtures

create test/functional

create test/integration

create test/unit

create vendor

create vendor/plugins

create tmp/sessions

create tmp/sockets

create tmp/cache

create tmp/pids

create Rakefile

create README

create app/controllers/application.rb

create app/helpers/application\_helper.rb

create test/test\_helper.rb

create config/database.yml

create config/routes.rb

create config/initializers/inflections.rb

create config/initializers/mime\_types.rb

create config/initializers/new\_rails\_defaults.rb

create config/boot.rb

create config/environment.rb

create config/environments/production.rb

create config/environments/development.rb

create config/environments/test.rb

create script/about

create script/console

create script/dbconsole

create script/destroy

create script/generate

create script/performance/benchmarker

create script/performance/profiler

create script/performance/request

create script/process/reaper

create script/process/spawner

create script/process/inspector

create script/runner

create script/server

create script/plugin

create public/dispatch.rb

create public/dispatch.cgi

create public/dispatch.fcgi

create public/404.html

create public/422.html

create public/500.html

create public/index.html

create public/favicon.ico

create public/robots.txt

create public/images/rails.png

create public/javascripts/prototype.js

create public/javascripts/effects.js

create public/javascripts/dragdrop.js

create public/javascripts/controls.js

create public/javascripts/application.js

create doc/README\_FOR\_APP

create log/server.log

create log/production.log

create log/development.log

create log/test.log

As you can see we’ve now got the skeleton generated and this would probably be a good moment to talk some more about the structure of the skeleton.

The app folder will hold our controllers, models, views and helpers and will probably be where you write most of your code. The next folder is the config folder and as you may have guessed that is where you store your configuration files like the connection strings to your database. This folder also contains a file called environment.rb, that file contains the settings and initializers for your application. You can specify environment specific options in the subfolder environments (config/environments). That folder contains the files for each environment you may have, by default those are development, test and production. The config folder also holds the file routes.rb and amazingly that file is where you will configure the routes for your application.

The folder public is actually the root for your website if you look at it from the web servers perspective. And holds all the static files your application may need like the html, JavaScript, CSS and image files. In the log folder you can find the different log files that your application will generate. In the folder script you can find a bunch of scripts that make working with rails easier. It contains scripts to install plug-ins, start the web server, .,,

There is a last folder that deserves some explanation and that folder would be the vendor folder. Rails can be run as a gem but you can also freeze the version of rails you’re using for that application within that application and then that will be the version used by the web server. That process is called freezing rails and those frozen gems are stored in the vendor folder. Furthermore any plug-ins you want to use, are stored in the folder vendor/plugins. The vendor folder was aptly named because that’s where you keep you’re 3rd party vendor libraries. This should give you an idea about what all the folders mean. This may be a good moment to talk a little bit more about the extensibility model of Rails: plug-ins.

10.3.3 Easy extensibility through plug-ins

In any rails application you can override or extend the default behavior through a system of plug-ins. I found them to be very useful and created some to make logic like html helpers etc portable between applications. There was a way we defined forms that involved larger chunks of html and some JavaScript behavior. Since this was the standard way of creating forms for us in all the applications we created a plug-in that we then installed in every application and that was done.

For our application we’re going to use 1 plug-in. This plug-in should make it easier to create an authentication procedure and that plug-in is called restful-authentication. If you’ve never installed a plug-in before you will first need to discover the list of repositories it has to search in. For the plug-in system to work you need to have subversion installed on your machine. It helps to have git too because some plug-ins are hosted in git repositories.

Git

In the rails community many people are using git as their primary version control system. If you haven’t heard of it before it might seem a bit strange at first but it does work really well.

You can get for windows at: <http://code.google.com/p/msysgit/> and for mac users you can install it using macports (<http://www.macports.org/>) with the command sudo port install git-core. On linux you can either compile it from source or through your system’s package manager (<http://git-scm.com/download>).

You will also need an active internet connection to use the plug-in system. The samples that accompany this book have a completed version of the application which contains the plugins you could potentially just copy the contents of the vendor/plugins folder into your application folder and still get the plug-ins to work.

To install the restful authentication plug-in: ir script/plugin install git://github.com/technoweenie/restful-authentication.git

Now we’ve got the necessary plugins installed in our application we only need to make sure we’ve got the RSpec framework installed too. To install those gems execute the command igem install rspec-rails. And the last step in the preparation process involves generating a site key for restful authentication and instrumenting the application to use rspec. To generate the site key you execute the command irake auth:gen:site\_key and to instrument your application for rspec execute ir script/generate rspec.

At this moment we’re done bootstrapping our development and we can move on to actually start development on the application. Our first step in that direction is to create the models we need.

10.4 Building the models

As we’ve seen earlier Rails is an MVC framework and as such most of the business logic is put in the models. To build a model there are a couple of steps we need to follow. First we need to generate a migration and a model file. Next we’ll set up validation and if necessary define the relationships and then we’re read to start adding some methods to it for processing or retrieving data. But lets start at the beginning by generating the migration file.

10.4.1 Migrations for managing database schema changes

Migrations are a part of the ActiveRecord library that is included in the Rails framework. It’s a nice way of keeping your database schema changes in version control. You define the schema using normal ruby code and record any data migrations in an upward and downward direction so that you can always move forward or backwards in the schema. I immediately liked migrations as soon as I got wind of them and since I’ve used them in most of my apps, it’s just a really nice way of managing your database.

In migrations you can define creating/dropping tables, indexes etc. You can add/remove/rename columns and you can use the ActiveRecord models to migrate your data or execute raw sql. As you’ve probably established that this is a pretty powerful tool to have in your arsenal. And as long as you steer clear of database specific features your migrations are portable across database engines. So you can use sqlite or mysql as a development server but deploy to Microsoft SQL Server or Postgres for production.

For our twitter application we need 2 models: 1 for users and 1 for statuses. The user model will be generated through a generator supplied by restful authentication, but we will add the extra columns we need to it. We can just dive in because we’ll be using sqlite as our database engine and rails comes preconfigured for that engine.

To generate the necessary controllers, model and migration for the authentication routine we have to execute script/generate authenticated user sessions --rspec in the application folder.

At this point we have something we can run already. The restful authentication generator also created the specs (tests) for all the items it generated. Why don’t we try to run what we’ve got so far. The first step in this process is migrating the database to do that execute the command irake db:migrate. This will create the users database table for us and we’re ready to start running the specs by executing irake spec. Listing 10.2 shows the output of those commands.

10.2 Running your first migration and specs

+ivan@ivan-mbp:~/projects/ironruby\_book/twitter\_sample/twitter

» irake db:migrate

(in /Users/ivan/projects/ironruby\_book/twitter\_sample/twitter)

db/development.sqlite3 already exists

== 20081102210757 CreateUsers: migrating ======================================

-- create\_table("users", {:force=>true})

-> 0.0046s

-- add\_index(:users, :login, {:unique=>true})

-> 0.0024s

== 20081102210757 CreateUsers: migrated (0.0076s) =============================

+ivan@ivan-mbp:~/projects/ironruby\_book/twitter\_sample/twitter

» irake spec

(in /Users/ivan/projects/ironruby\_book/twitter\_sample/twitter)

..................................................................

Finished in 0.539931 seconds

66 examples, 0 failures

.........................................................................................................................................................

Finished in 0.982146 seconds

153 examples, 0 failures

The restful authentication generator generated more than 200 tests and luckily they all pass. All you have to do now is make sure they continue to pass and add new tests for the behavior we’re going to add to the user model. At this point it might be a good idea to read through some of the specs that have been generated so you can get a feel for what’s going to be in store for you in the remainder of this chapter.

Now that you’ve had a taste of what the flow of developing rails will feel like it might be time for some real action. The next model we need to create is the status model. This time there will be some actual coding involved. The first step in the process is to generate the model. Normally you would generate a model using the command ir script/generate model <<modelname>>. Because we’re using RSpec we need to generate a model that is accompanied by a spec file instead of a test file, so in our case the command to execute is ir script/generate rspec\_model status. Listing 10.3 shows the output of that command.

10.3 Generating the Status model

» ir script/generate rspec\_model status

exists app/models/

exists spec/models/

exists spec/fixtures/

create app/models/status.rb

create spec/fixtures/statuses.yml

create spec/models/status\_spec.rb

exists db/migrate

create db/migrate/20081103183911\_create\_statuses.rb

As you can see in the listing above this generated a model file called status.rb. It also generated a fixture and a spec file as well as a migration file. We haven’t talked about fixtures earlier but it’s a way of specifying data for your tests. At some point you will probably need some test data for your tests. It would probably also be preferable if you were in complete control of that test data so that you can also test all the edge-cases or reproduce bugs in a repeatable fashion later on in the lifetime of your application. Fixtures allow you to specify test data using YAML.

The second step is to define all the columns we need on the status model. We know which columns we need by looking at the model classes we created for WPF in chapter 4 or by looking at the xml for one of the urls on the twitter website. Listing 10.4 shows which columns to add to the create\_statuses migration. The columns we have to add are bolded in the listing.

Listing 10.4 Adding the columns to the statuses table

class CreateStatuses < ActiveRecord::Migration

def self.up 1

create\_table :statuses do |t| 3

t.boolean :favourited, :default => false

t.integer :in\_reply\_to\_status\_id

t.integer :in\_reply\_to\_user\_id

t.string :source, :limit => 50

t.string :source\_url, :limit => 500

t.string :text, :limit => 140, :null => false 4

t.boolean :truncated, :default => false

t.timestamps

end

end

def self.down 2

drop\_table :statuses

end

end

1. The upward migration
2. The downward migration
3. Creating a table
4. A required column

As you can see a migration class contains 2 class methods one that is to be executed in when the migrations are upgraded with name up #1. The other method gets executed when you downgrade the database schema and is called down #2. The downward migration in this case only contains the code to drop the statuses table in the database. The upward migration is far more interesting in this case.

The upward migration defines the statement to create the statuses table. The create\_table method is passed a block with the table definition as argument #3. We then go on defining all the columns we’re going to need. The column that defines the text column is a required column, otherwise a status update wouldn’t make much sense.

Before we move on to implementing the code necessary for the model we’re going to do something that goes against the iterative style of development you would normally use when doing ruby and generate another migration to add the extra columns to the user model. We do it this way because that works better in the structure of this chapter.

The user model does have more columns than the default ones that get generated by the restful authentication generator. To add those we’re going to create another migration called add\_columns\_to\_user and we do that by executing the command ir script/generate migration add\_columns\_to\_user. That will generate a migration file, which we’ll complete with the remaining columns, shown in bold in listing 10.5.

Listing 10.5 Adding columns to the users table

class AddColumnsToUser < ActiveRecord::Migration

def self.up

**add\_column :users, :description, :string, :default => '', :limit => 140**

**add\_column :users, :favourites\_count, :integer, :default => 0, :null => false**

**add\_column :users, :followers\_count, :integer, :default => 0, :null => false**

**add\_column :users, :following, :boolean, :default => false**

**add\_column :users, :friends\_count, :integer, :default => 0, :null => false**

**add\_column :users, :location, :string, :limit => 140, :default => ''**

**add\_column :users, :profile\_image\_url, :string, :limit => 500**

**add\_column :users, :protected, :boolean, :default => false**

**add\_column :users, :users\_count, :integer, :default => 0, :null => false**

**add\_column :users, :time\_zone, :string, :limit => 140**

**add\_column :users, :url, :string, :limit => 500**

**add\_column :users, :utc\_offset, :string, :limit => 50**

end

def self.down

**remove\_column :users, :description**

**remove\_column :users, :favourites\_count**

**remove\_column :users, :followers\_count**

**remove\_column :users, :following**

**remove\_column :users, :friends\_count**

**remove\_column :users, :location**

**remove\_column :users, :profile\_image\_url**

**remove\_column :users, :protected**

**remove\_column :users, :users\_count**

**remove\_column :users, :time\_zone**

**remove\_column :users, :url**

**remove\_column :users, :utc\_offset**

end

end

In the listing above we can see 2 new methods for migrations, I think their names are self-explanatory and with this we are ready to commit the migrations to the database by executing rake db:migrate. Feel free to play around with migrations to explore what other possibilities they have to offer as we’ve only just scratched the surface.

Now that we know about creating models and migrations it’s time to move on to a discussion on models and preparing those for our twitter clone. We’ll look at adding validation, relations and some business rules.

10.4.2 Adding behavior to the models

We can describe this behavior in an RSpec spec. The minimum behavior the status model should satisfy is shown in the output (listing 10.6) of the irake spec:doc command for the Status spec .

10.6: Output for the status spec

Status class with fixtures loaded

- should count six Statuses

Status class with fixtures loaded A

for persistence it

- should create a status

- should not create an empty instance

- user\_id cannot be nil

- should be invalid without a text

- should be valid with a blank source\_url

- should be valid with a source\_url of nil

- should have a valid url when source\_url is not blank or nil

Status class with fixtures loaded B

as relations it

- should have a user which is the author

- can have a user to which this status is a reply

- can have a status to which this can be a reply

Status class with fixtures loaded C

finders

- should find all the statuses for a user including his friends

- should find the last 20 status updates for the public timeline

- should find the status updates for this user only

- should find all the replies by this user only

1. Verifying validations
2. Verifying relations
3. Verifying finders

This output (listing 10.6) shows us that we first verify if we have provided enough data to test by verifying the count. The output can be viewed in 3 blocks: one that describes its validation, one that describes the relations and a last one that describes the special finders.

This is not a complete spec but for our purposes it should definitely give you an idea of how you can use RSpec to describe the behavior of your application. Now how would we go about translating that list of specs into an RSpec example group? Before diving into the code listing of the example group I’d like to say that we won’t be including all the specs in this chapter because that would make it too long. I’ll show you the specs for the status model and the rest are provided with the sample code of this chapter. Listing 10.7 shows the initial listing for the status spec.

I should probably also say that the way these specs are structured isn’t completely according to the rules but it made it easier to explain what’s going on. You would probably want to do some more reading on the subject of BDD at <http://behaviour-driven.org/>.

Listing 10.7: The example group for the Status model

require File.dirname(\_\_FILE\_\_) + '/../spec\_helper'

context "Status class with fixtures loaded" do

fixtures :users, :follower\_users, :statuses

specify "should count six Statuses" do

Status.count.should == 6

end

# more specs and contexts should come here

end

The way BDD operates is you define a context in that context you can add specifications. So the first context we set up is that we’ll be dealing with the Status class and the status class has the fixtures with our test data loaded. We know we need to have 6 statuses in the database when we start our test run so we verify that in our first specification. Now that this specification passes we know we’ve got all the test data we need and can start to move on to the more useful specs.

Defining validation and persisitence

The first piece of functionality I generally start with is the validation and persistence behavior of the model. So we define a new context in which we will group the examples that have to do with verifying this persistence behavior as shown in listing 10.8. The first thing we specify is that we should be able to save a new status when we give it all the necessary attributes. The second specification verifies that we cannot create a status when none of the attributes have a value. In these specifications we make use of the create\_status helper method. When you’re writing tests or specs the principle of DRY still applies so it is probably a good idea to group common code into helper methods and such.

Listing 10.8: Verifying persistence

context 'for persistence it' do

specify "should create a status" do

status = create\_status

violated "#{create\_status.errors.full\_messages.to\_sentence}" if status.new\_record?

end

specify "should not create an empty instance" do

Status.new.should\_not be\_valid

end

end

protected

def create\_status(options={})

Status.create({

:source => "ironruby in action",

:source\_url => "http://manning.com/carrero",

:text => "the random status description",

:user\_id => 1

}.merge(options))

end

At this point we can start looking at implementing the specs that will verify the validation. A status update doesn’t make sense without a text and a user that did the status update you can check out listing 10.9 for those specs.

Listing 10.9: Ensuring compulsory attributes

specify "user\_id cannot be nil" do

lambda do

status = create\_status(:user\_id => nil)

status.errors.on(:user\_id).should\_not be\_nil

end.should\_not change(Status, :count)

end

specify "should be invalid without a text" do

lambda do

status = create\_status(:text => nil)

status.errors.on(:text).should\_not be\_nil

end.should\_not change(Status, :count)

end

In the first specification we say that we want to create a status but with a missing user\_id. In the following line we check if there are error messages for the user\_id attribute. We did wrap those lines in a lambda so we can also verify that the count of the statuses hasn’t changed. The specification for the text attribute is very similar except that we do the verification on the text attribute instead of the user\_id attribute.

When you run the specs using irake spec they should fail because we haven’t implemented anything on the model yet to actually do that validation. The changes we’ve got to make to the Status model are shown in listing 10.10.

Listing 10.10: Adding the obligated properties to the Status model

class Status < ActiveRecord::Base

validates\_presence\_of :text

validates\_presence\_of :user\_id

end

We just tell the model that we want to validate the presence of a text attribute and the user\_id attribute. If you were to run the specs now, they should pass. Our next specs have to do with the source\_url. As we’ve seen earlier the source\_url can be empty but if it isn’t the value should be a valid url. Listing 10.11 shows the code for the specs of this piece of functionality.

Listing 10.11: Specifying the behaviour of source\_url

specify "should be valid with a blank source\_url" do

lambda do

status = create\_status(:source\_url => "")

violated "#{status.errors.full\_messages.to\_sentence}" if status.new\_record?

end.should change(Status, :count)

end

specify "should be valid with a source\_url of nil" do

lambda do

status = create\_status(:source\_url => nil)

violated "#{status.errors.full\_messages.to\_sentence}" if status.new\_record?

end.should change(Status, :count)

end

specify "should be a valid url when source\_url is not blank or nil" do

lambda do

status = create\_status(:source\_url => "htt://dld.")

status.errors.on(:source\_url).should\_not be\_nil

end.should\_not change(Status, :count)

end

Again we’ve got the red/green/refactor triumvirate that guides us to the development process. So at this point we’ve got specs but they would obviously fail because the functionality hasn’t been implemented yet on the Status model, take a look at Listing 10.12 for the implementation of that functionality.

Listing 10.12: The implementation of the Status model so far

class Status < ActiveRecord::Base

validates\_presence\_of :text

validates\_presence\_of :user\_id

validates\_format\_of :source\_url,

:with => /^https?:\/\/([a-zA-Z0-9\_-]+\.[a-zA-Z0-9\_-]+)+(\/\*[A-Za-z0-9\/-\_&:?+=\/\/.%]\*)\*/i,

:allow\_nil => true,

:allow\_blank => true

end

As you can probably see that implementation is really complicated. The most complicated part about it is probably the regular expression. We use the validates\_format\_of method with the source\_url as attribute. Then we specify a regular expression to validate the url format. And lastly we tell the function that we’re ok with nils and blank values.

That is all there is to it to implement the validation for the status model. Next we’ll be looking at how to implement the relationships between the status model and the user model. But before you do that it might be wise to run your specs once more to see that they all pass so you can move on with a little bit of confidence.

Running specs automatically

Instead of running the specs manually every time you’ve implemented a little bit of functionality you can also use autotest which will run your specs automatically every time you save a relevant file. To do that you would have to install the ZenTest gem which includes the autotest command. You can install this gem by executing igem install ZenTest.

And in a terminal window you navigate to the root of your rails project and type ir script/autospec. When you execute the command it will run all the specs. I will have autospec running 95% of the time when I’m developing rails applications.

Ok now that we are sure that our specs pass we can start thinking about the relations between the User and Status model. Earlier we’ve seen that a status has an author and it can be in reply to a user. In addition to these relationships a status can also be a reply to another status. That are 3 relations and listing 10.13 shows the code for these specs.

Defining the relationships

Rails has a couple of different ways to define relationships between models. We won’t discuss all of them as this book isn’t really about rails but there is plenty of information available on the Internet or in other books about this subject. Table 10.1 shows the different types of relationships in Ruby on Rails with a brief description of what they mean.

Table 10.1: The available relationship types in Ruby On Rails

|  |  |
| --- | --- |
| Relationship type | Description |
| belongs\_to | This is the table that holds a foreign key |
| has\_many | Can have 1 or more dependants (always returns a collection) |
| has\_one | Can have 1 dependant (always returns the first item if more are found) |
| has\_and\_belongs\_to\_many | Joined through a join table |

In the discussion we will have on this subject we won’t touch all of them. I couldn’t find a good place to use the has\_one relationship. The has\_one relationship is very much like the has\_many relationship with the only difference that it just returns one element instead of a collection.

Listing 10.13: Specs for the relationship between Status and User

context "as relations it" do

specify "should have a user which is the author" do

status = statuses(:first)

status.should\_not be\_nil

status.user.should\_not be\_nil

status.user.id.should == 1

end

specify "can have a user to which this status is a reply" do

status = statuses(:fourth)

status.should\_not be\_nil

status.in\_reply\_to\_user.should\_not be\_nil

status.in\_reply\_to\_user.id.should == 2

end

specify "can have a status to which this can be a reply" do

status = statuses(:third)

status.should\_not be\_nil

status.in\_reply\_to\_status\_id.should == 2

status.in\_reply\_to\_status.should\_not be\_nil

status.in\_reply\_to\_status.id.should == 2

end

end

The first new spec we add to our class is going to verify that the user\_id attribute on the Status model is provided because it is compulsory. Next we’re going to check if there is a user available to which this status update is a reply. After that we’re verifying if we can have a status to which this status is a reply. The code for these 3 assertions is pretty similar so I’ll just explain one. We use the fixtures to retrieve a status with our predefined test data. We verify if the status isn’t nil so that we’re sure we’ve got an actual object. Next we check that the foreign key attribute has the expected value. And lastly we verify if we also have an actual status object that is retrieved using this foreign key. Now that we have the specs we still have to implement that behavior on the Status model as shown in listing 10.14.

Listing 10.14: Implementing the relations on the Status model

class Status < ActiveRecord::Base

validates\_presence\_of :text

validates\_presence\_of :user\_id

validates\_format\_of :source\_url,

:with => /^https?:\/\/([a-zA-Z0-9\_-]+\.[a-zA-Z0-9\_-]+)+(\/\*[A-Za-z0-9\/-\_&:?+=\/\/.%]\*)\*/i,

:allow\_nil => true,

:allow\_blank => true

**belongs\_to :user**

**belongs\_to :in\_reply\_to\_user, :class\_name => "User", :foreign\_key => "in\_reply\_to\_user\_id"**

**belongs\_to :in\_reply\_to\_status, :class\_name => "Status", :foreign\_key => "in\_reply\_to\_status\_id"**

**has\_many :replies, :class\_name => "Status", :foreign\_key => "in\_reply\_to\_status\_id"**

end

I’ve highlighted the new lines in the status class in bold. A relation has to be defined in both the entity that has the foreign key and the entity that is being referenced; we’ll look at modifying the user model as soon as we’re done with the status model. First we add the user relation as a belongs\_to relationship.

Because we followed the convention and have an attribute user\_id ActiveRecord will figure it out for automatically that we mean the User model. The next relation we add is the belongs\_to relationship of a user to which this status is a reply. This does require some more configuration, because the foreign key field is called differently than the convention. So we say that we’re interested in the model User and that the foreign key is in\_reply\_to\_user\_id. The next relationship is a self-referencing relation. This means that we have to define both sides of the relationship on the status model. We define a belongs\_to relationship in\_reply\_to\_status\_id that references the class Status through the foreign key. We also define a has\_many relationship that uses the class Status and is referenced by the in\_reply\_to\_status\_id foreign key.

The next bit of work we have to do is on the User model because it has been referenced twice with a belongs\_to relationship from the Status model we have to make sure the corresponding has\_many relationship is defined on the User model. For brevity we haven’t included the code for this functionality in the chapter but it is included in the samples for this chapter. We’ve seen how to define relationships and how to test for them through the Status model. If you want to learn more about Ruby on Rails there are some excellent books available from Manning to get you up to speed.

This also concludes our discussion on relations in Ruby on Rails and it is time to implement the queries that we’re going to use to expose our data over the web.

Adding the queries to the models

So far we’ve been dealing with persistence and validation, now the time has come to start retrieving some of our data so that we will have something to show later on. Again for brevity we will only discuss one finder method but in the samples there are a couple more implemented. We’re primarily interested in the timeline\_with\_friends\_for finder method. This will give us back all our tweets and the tweets of all the people we’re following. We will start again by writing down the specs for this finder method and then we’ll implement the actual functionality. Listing 10.15 shows the specs we’re going to add to the status example group.

Listing 10.15: The specs for the finders on the status model

context "finders" do

specify "should find all the statuses for a user including his friends" do

result = Status.timeline\_with\_friends\_for :user\_id => users(:aaron).id

result.should\_not be\_empty

result.length.should == 5

end

end

Because we defined fixtures with test data earlier we’ve got predictable results. That means we can verify easily if we find the right amount of statuses. We call the timeline\_with\_friends\_for method and pass it the id of the user for who we want to retrieve the status updates. Next we first check that the collection isn’t empty and after that we check if the collection has 5 elements like we expect it to. This brings us back to the red in the red/green/refactor mantra. Listing 10.16 shows how we’re going to implement this method on the status model.

Listing 10.16: Implementing the friends\_with\_timeline\_for method

class << self

def timeline\_with\_friends\_for(options={})

options[:where] = "( user\_id = :user\_id or user\_id in (select user\_id from follower\_users where follower\_id = :user\_id) )"

find :all, extract\_timeline\_options(options)

end

private

def extract\_timeline\_options(options={})

opts = { :where => "( user\_id = :user\_id )", :page => 1, :count => DEFAULT\_PAGESIZE, :sort => DEFAULT\_SORT }.merge(options)

limit = opts[:count].to\_i

limit = 200 if limit > 200

offset = (opts[:page].to\_i - 1) \* limit

{

:conditions => build\_conditions\_from(opts),

:order => opts[:sort],

:include => [:user],

:limit => limit,

:offset => offset

}

end

def add\_since\_if\_specified(whr, par, opts)

if opts.respond\_to?(:since)

whr = whr.blank? ? "( created\_at >= :since )" : "#{whr} AND ( created\_at >= :since )"

par[:since] = opts[:since].to\_s(:db)

end

end

def add\_since\_id\_if\_specified(whr, par, opts)

if opts.respond\_to?(:since\_id)

whr = whr.blank? ? "( id >= :since\_id )" : "#{whr} AND ( id >= :since\_id )"

par[:since\_id] = opts[:since\_id]

end

end

def build\_conditions\_from(opts)

opts.symbolize\_keys!

whr = opts[:where]

par = { :user\_id => opts[:user\_id] }

add\_since\_if\_specified whr, par, opts

add\_since\_id\_if\_specified whr, par, opts

[whr, par]

end

end

This method is a class method, as we’ve seen in chapter 2 there are several ways to define a class method in Ruby. I personally prefer this one but that is mostly a matter of taste. There are a bunch of things going on in these methods. The extract\_timeline\_options is a method that allows for reuse in more finders because that logic is always pretty similar.

Firstly we set the where key in our options hash to the where clause we would add to a SQL statement. The difference is that because we use named parameters (:user\_id) they are properly escaped and you’re safe for SQL injection attacks. Next we’ll normalize our search parameters to include paging, sorting and a maximum number of rows. We do so by calling the extract\_timeline\_options, this method has this name because later on we’ll pass the public class method the options hash we get from the web request and we probably don’t want to pass all the keys in this hash to our find method.

In the extract\_timeline\_options method we first define a hash with default values, which we’ll merge with the options that are provided in the parameters for this method to possibly override some or all of those defaults. Next we ensure that we never return more than 200 items from a find as well as how big the size of a page has to be. Then we create a new hash that contains the keys that the find method from ActiveRecord expects and in doing so we call the build\_conditions\_from method.

In the build\_conditions\_from method we make sure that all the keys in our hash are symbols and not plain strings. Next we assemble the actual where statement that will be executed and return it in a find method friendly format. That ensures we’ve got the right data but we also define the parameters for paging and sorting. The last thing is we also include an :include key in the options hash so that we eager load the authors with the query.

If you save the file and run the specs again it should work properly, provided that your test data is ok. It’s maybe also worth mentioning that there is no test per method. You’re rather testing the public interface of a class and because the public methods use the private methods they are implicitly tested.

At this point we’re done with the models and we can move on to the controllers and the views.

10.5 Talk to the controller because the model ain’t listening

In an MVC world the controller is responsible for taking user input and delegating that to the correct models. When the models are done the controller decides how he’s going to display or return the data.

10.5.1. Responding to different formats

Internally Rails uses a formatter design pattern to enable returning different formats. You can register your own formats in a mime types file. You can find this file in config/intializers/mime\_types.rb when you’re in the root of your app. Rails already listens for a bunch of mime types and extensions. Listing 10.17 shows the mime types that have been registered by default in a rails application.

Listing 10.17: The default recognized mime types

# Build list of Mime types for HTTP responses

# http://www.iana.org/assignments/media-types/

Mime::Type.register "\*/\*", :all

Mime::Type.register "text/plain", :text, [], %w(txt)

Mime::Type.register "text/html", :html, %w( application/xhtml+xml ), %w( xhtml )

Mime::Type.register "text/javascript", :js, %w( application/javascript application/x-javascript )

Mime::Type.register "text/css", :css

Mime::Type.register "text/calendar", :ics

Mime::Type.register "text/csv", :csv

Mime::Type.register "application/xml", :xml, %w( text/xml application/x-xml )

Mime::Type.register "application/rss+xml", :rss

Mime::Type.register "application/atom+xml", :atom

Mime::Type.register "application/x-yaml", :yaml, %w( text/yaml )

Mime::Type.register "multipart/form-data", :multipart\_form

Mime::Type.register "application/x-www-form-urlencoded", :url\_encoded\_form

# http://www.ietf.org/rfc/rfc4627.txt

# http://www.json.org/JSONRequest.html

Mime::Type.register "application/json", :json, %w( text/x-json application/jsonrequest )

The Mime::Type.register method takes 5 arguments at most and the last 3 are optional. The first one is the principal mime type, the second argument is the extension of the file. The 3rd argument is optional and is a collection of alternative mime-types for that extension. The 4th argument is a collection of alternative extensions for the registered mime type. And the last argument is a Boolean indicating whether to lookup the mime type or not.

Now that we know this we also know how we could potentially add other extensions like .pdf or .xls to be recognized and rendered by the controller.

10.5.2. Reaching your actions through routing

A central piece of MVC is routing. Routing maps urls to controllers, actions and their parameters. If we have to get technical then you could possibly say that routing functions as a front controller (http://martinfowler.com/eaaCatalog/frontController.html). This front controller decides which controller to instantiate and next which action to invoke on the controller by parsing the requested url. It might be a good plan to first show you our routing file and then explaining what every line means etc. Listing 10.18 shows the complete routing rules implementation. You can find the routes in a rails app in the folder config and the file routes.rb.

Listing 10.18: Defining the route table for this application

ActionController::Routing::Routes.draw do |map|

map.resources :statuses, :collection => {

:friends\_timeline => :get,

:user\_timeline => :get,

:public\_timeline => :get,

:replies => :get,

:friends => :get,

:followers => :get

}

map.resources :users, :has\_many => [:statuses]

map.connect ':controller/:action/:id'

map.connect ':controller/:action/:id.:format'

map.root :controller => "statuses", :action => "friends\_timeline"

end

Contrary to the intuitive tendency to start at the top we will start our discussion at the bottom of this code listing. The last line map.root maps the “/” url so that we can respond to a request for the root of our web application by using a controller and action method.

The next line is a catch all mapping that routes all requests to controllers and actions with the specified format. These parameters will also be available inside the controller in the params hash.

Next we’ll start entering the realm of RESTful development. Where we talk about resources that we can manipulate by constructing a sentence. For example to request the list of all users one would have to construct the following phrase to send to the webserver: “GET /users”, to update a user with id 3 you would construct the sentence: “PUT /users/3”. In REST the HTTP verb plays an important role in telling the webserver what you want it to do.

When you use resources Rails makes a couple of assumptions, which you can override of change of course. Rails assumes that you have a couple of actions defined on your container. The list of actions can be found in table 10.2.

Table 10.2: The default actions for a RESTful resource

|  |  |  |
| --- | --- | --- |
| Action | Verb | Description |
| index | GET | Returns a list of resources |
| show | GET | Shows a resources |
| new | GET | Initializes a new resource |
| create | PUT | Creates a new resource |
| edit | GET | Shows a resource in edit mode |
| update | PUT | Updates an existing resource |
| destroy | DELETE (in http that is a put) | Deletes an existing resource |

So we define a resource for the users and we say that this resource has a relationship with the resource status, which we’ll define next. You can define relations on resources in the form of has many or has one. This then gives you pretty logical urls that for developers are discoverable but I wouldn’t trust on the average user to master that structure.

The status resource is the most complicated route definition that we have to deal with in this application and that is because on the status controller we’ll define a couple more actions to comply with the twitter API. So we tell Rails that for the statuses controller we want to have a couple more endpoints, and these endpoints return a collection of statuses. And that’s it for our routes.

There is one last thing I want to bring to your attention about routing. If you want to figure out what the names are for your routes then you can do so by asking rake to print out your route table for you by executing the command irake routes. Listing 10.19 shows a snippet of the output of this command, if you’re curious for the full output I invite you to execute the command yourself in the root directory of the rails application.

Listing 10.19: The route table for the application

friends\_timeline\_statuses GET /statuses/friends\_timeline {:controller=>"statuses", :action=>"friends\_timeline"}

formatted\_friends\_timeline\_statuses GET /statuses/friends\_timeline.:format {:controller=>"statuses", :action=>"friends\_timeline"}

user\_timeline\_statuses GET /statuses/user\_timeline {:controller=>"statuses", :action=>"user\_timeline"}

formatted\_user\_timeline\_statuses GET /statuses/user\_timeline.:format {:controller=>"statuses", :action=>"user\_timeline"}

public\_timeline\_statuses GET /statuses/public\_timeline {:controller=>"statuses", :action=>"public\_timeline"}

formatted\_public\_timeline\_statuses GET /statuses/public\_timeline.:format {:controller=>"statuses", :action=>"public\_timeline"}

Unfortunately the output of this command is too wide so the layout of the snippet is completely messed up. Now we know how a request actually figures out which controller to use we can start looking at implementing this controller.

10.5.3 Gluing views and models together with a controller

As you know the controller is the central piece of the MVC architecture that mediates the communication between the view and the model and vice versa. We will discuss the view later and we’ve already got some idea of what to do with models. Because our primary interest is the API side of the twitter application, we’ll make sure that the controller responds to XML and JSON for all the GET requests in addition to html. We know already which mechanism in rails is leveraged to make that happen.

So your controller takes input from the user. This input can be obtained from a number of different sources. It can come from the server variables, it may be defined in the routing table, and it may even come from a query string or form values. So the params hash holds values from all these sources.

The implementation for our controllers is going to be pretty simple because we need to respond to a login request. We also need to allow for sending status updates and retrieving status updates for the currently logged on user and his friends. All in all not such a big chunk of functionality but enough for the sample application from chapter 4 to work without an active internet connection.

We already have a controller that takes care of authenticating a user either through a form submitted by a user from a web browser or by using HTTP basic authentication for API users. That piece of functionality was already done when we generated the user model and session controller with restful authentication. That leaves us with implementing the update and friends\_timeline controller action. We’ll first have to generate a StatusController to implement this functionality. To do this we’re going to run a generate command, because we’re using RSpec for testing and I want to take a head start on the specs we’ve got to write we’ll use their variant of a scaffold. A scaffold is a great starting place but it’s definitely not something you can put in production. So we’re going to run the following command: ir script/generate rspec\_scaffold status text:string --skip --skip-migration. That will create our controller for us without touching any of the files we already created and it will create some rhtml files that contain a field for the text attribute of our status model.

If you want you can take a look in the files the generator created and at the specs that have been generated. You can also run irake spec to check if everything is still working. At this point we’re ready to start coding again. We’re going to start with the specs for the update functionality. Oh right ☺ we used a scaffold that means that that piece of controller functionality is already done.

The only thing we need to do is make sure it responds to JSON requests too and that’s only for compliancy with the twitter API. We will need to add the JSON format in all the actions that return some form of output. To do this in a DRY way we’ll introduce a private method render\_for\_api and replace the respond\_to block with a call to that private method. Listing 10.20 shows the code for that private method and the code for the index action. Notice that I haven’t talked about writing specs first. You can just assume we already did that part; they are available in the sample code that comes with this book but including it here would make this too lengthy.

Listing 10.20: Rendering statuses

private

def render\_for\_api(param)

respond\_to do |format|

format.html

format.xml { render :xml => param.to\_xml(Status.default\_serialization\_options) }

format.json { render :json => param.to\_json(Status.default\_serialization\_options) }

end

end

Our method render\_for\_api takes one parameter and the first thing it does is call the respond\_to method that will figure out which format to pick. When it’s html we don’t have to do anything. But when it’s xml we can see that we call the to\_xml method on the param parameter and we pass that Status.default\_serialization\_options. The to\_xml method takes a hash with options to customize the way the xml is rendered. Since this isn’t knowledge a controller should have but it belongs to the model we’ve defined it as a class method on the Status model. Listing 10.21 shows the code for the default\_serialization\_options method.

Listing 10.21: The default serialization options for the Status model

def default\_serialization\_options

{ :include => [:user] }

end

All this method does is tell the xml serializer that we want to include the user relationship in the xml too so if there is one it should be serialized too.

The to\_json method on an ActiveRecord model takes the same parameter hash as to\_xml so we can just re-use that and be done with it. Listing 10.22 shows the code for the friends\_timeline action.

Listing 10.22: The friends\_timeline controller action

def friends\_timeline

@status = Status.new

@user = current\_user

opts = conditions

opts[:user\_id] = @user.id

@statuses = Status.timeline\_with\_friends\_for opts

render\_for\_api(@statuses)

end

This action starts with providing an empty status to the @status instance variable. Next we get the currently logged on user by calling the current\_user method that is provided by the restful-authentication plug-in. We then get the parameter hash, the code for that method is provided in listing 10.23, and strip the parameter hash from the controller, action and format keys so that we only have the values we’re actually interested in. In addition to stripping out the controller parameter keys it’s also checks a header If-Modified-Since or a hash key since to implement the twitter API properly the requestor could ask to get the updates from a particular date by either setting this header or by providing a since parameter in the url.

Next we set the user\_id paramter in the hash to the id of the currently logged on user and pass those parameters to the timeline\_with\_friends\_for method on the Status model that was shown in listing 10.16. The result of that call is processed by render\_for\_api and is the result of our controller action. This should give you an idea of how to display data from the database. We’ll take a look at the view that belongs to this action a little bit later. First we’ll check out how we’d go about saving input we get back from a view.

Listing 10.23 The conditions method on the ApplicationController

def conditions

result = params.reject {|k, v| [:controller, :action, :format].include? k.to\_sym }

result.symbolize\_keys!

since = headers['If-Modified-Since']||result[:since]

result[:since] = Time.parse(since) unless since.nil?

result

end

We’ll look at the code that was generated by the scaffold generator for the create action, shown in listing 10.24.

Listing 10.24: The create status action on the controller

def create

@status = Status.new(params[:status])

@status.user = current\_user

respond\_to do |format|

if @status.save

flash[:notice] = 'Status was successfully created.'

format.html { redirect\_to(@status) }

format.xml { render :xml => @status, :status => :created, :location => @status }

format.json { render :json => @status, :status => :created, :location => @status }

else

format.html { render :action => "new" }

format.xml { render :xml => @status.errors, :status => :unprocessable\_entity }

format.json { render :json => @status.errors, :status => :unprocessable\_entity }

end

end

end

In this method we first initialize a new Status object with the input that is in the params hash. This system works because of a naming convention we’ll see a little bit later when we discuss the view. Then we set the user that owns this status to the current user. And after that we’re responding to the different formats we support. When the save is successful we either redirect to the details view and when we’re responding to XML or JSON we return a HTTP status code of 201, which acknowledges that the create was successful and the new status is serialized.

When there is an error for the html view we re-display the form but with some error messages and for the API calls to JSON or XML we send a HTTP status code of 422, which indicates that there was a problem processing the PUT operation. For more information on the HTTP status codes and their mapping table 10.3 lists all the possible status codes and their symbol notation.

Table 10.3: The possible status codes and their mappings to symbols

|  |  |  |
| --- | --- | --- |
| Status code | Status message | Symbol |
| 1xx Informational | | |
| 100 | Continue | :continue |
| 101 | Switching Protocols | :switching\_protocols |
| 102 | Processing | :processing |
| 2xx Success | | |
| 200 | OK | :ok |
| 201 | Created | :created |
| 202 | Accepted | :accepted |
| 203 | Non-Authorative Information | :non\_authorative\_information |
| 204 | No Content | :no\_content |
| 205 | Reset Content | :reset\_content |
| 206 | Partial Content | :partial\_content |
| 207 | Multi-Status | :multi\_status |
| 226 | IM Used | :im\_used |
| 3xx Redirection | | |
| 300 | Multiple choices | :multiple\_choices |
| 301 | Moved Permanently | :moved\_permanently |
| 302 | Found | :found |
| 303 | See Other | :see\_other |
| 304 | Not Modified | :not\_modified |
| 305 | Use Proxy | :use\_proxy |
| 307 | Temporary Redirect | :temporary\_redirect |
| **4xx Client Error** | | |
| 400 | Bad Request | :bad\_request |
| 401 | Unauhtorized | :unauthorized |
| 402 | Payment Required | :payment\_required |
| 403 | Forbidden | :forbidden |
| 404 | Not Found | :not\_found |
| 405 | Method Not Allowed | :method\_not\_allowed |
| 406 | Not Acceptable | :not\_acceptable |
| 407 | Proxy Authentication Required | :proxy\_authentication\_required |
| 408 | Request Timeout | :request\_timeout |
| 409 | Conflict | :conflict |
| 410 | Gone | :gone |
| 411 | Length Required | :length\_required |
| 412 | Precondition Failed | :precondition\_failed |
| 413 | Request Entity Too Large | :request\_entity\_too\_large |
| 414 | Request-URI Too Long | :request\_uri\_too\_long |
| 415 | Unsupported Media Type | :unsupported\_media\_type |
| 416 | Request Range Not Satisfiable | :request\_range\_not\_statisfiable |
| 417 | Expectation Failed | :expectation\_failed |
| 422 | Unprocessable Entity | :unprocessable\_entity |
| 423 | Locked | :locked |
| 424 | Failed Dependency | :failed\_dependency |
| 426 | Upgrade Required | :upgrade\_required |
| 5xx Server Error | | |
| 500 | Internal Server Error | :internal\_server\_error |
| 501 | Not Implemented | :not\_implemented |
| 502 | Bad Gateway | :bad\_gateway |
| 503 | Service Unavailable | :service\_unavailable |
| 504 | Gateway Timeout | :gateway\_timeout |
| 505 | HTTP Version Not Supported | :http\_version\_not\_supported |
| 507 | Insufficient Storage | :insufficient\_storage |
| 510 | Not Extended | :not\_extended |

And with this table we also conclude our discussion on controllers in Rails and move on to the last part of the Model-View-Controller implementation. The views that we’re going to build so that we have an html representation and we can respond to the html format as well as to API calls.

**10.6 Giving the user something tangible: a View!**

Up to this point we’ve got a working application if the only clients we are supporting are machines. But if we want humans to be able to use our application we’re going to have to make it look somewhat good and give a visual representation that is easy to understand.

The Rails framework has a few different tools available to help you with this task and to assist you in reusing as much as possible.

10.6.1 The toolbelt

In a web application you typically have an overall look for your whole application or for different sections of your web application. It would probably be a good idea to not have to duplicate that standard layout for every form and view your going to create. In rails this is addressed by layouts. You save those in the app/views/layouts folder but we’ll discuss them a little bit later in this section.

Then typically there is the actual view on the data returned from the controller action. This is probably different for almost every different action and different controllers and this is what Rails sees as a view. Now among views there can still be things that are common between different views and you would typically like to be able to reuse those things (like form fields for example). You can reuse those html snippets by leveraging partials.

And sometimes you want to encapsulate pieces of html generation and reuse that in different views. These pieces html generation typically require some calculation or more complex processing than is appropriate in a rhtml template. This problem is solved by using helpers to encapsulate that logic and then you can call those methods from inside the rhtml template.

10.6.2 Layouts

When I start to develop an application I like it when it looks a little bit better than bare html elements. For this application I downloaded an XHTML template with proper CSS support to have a somewhat nicer looking overall layout. I got my template from [Andreas Viklund](http://andreasviklund.com/templates/). When you get a template like that, it generally contains 3 types of files. The html file, a css file and a bunch of images.

To make these things work with the rails infrastructure I did the following. I created a file default.html.erb in the layouts folder and copied the contents of my xhtml template in that file. I copied the css files to the folders #{RAILS\_ROOT}/public/stylesheets and the images files I copied into #{RAILS\_ROOT}/public/images. I edited the css files so that the paths to the image files are pointing to the new location.

Then I went into the html template and replaced the 2 css link tags with 2 calls to helper methods that come with the Rails framework, shown in listing 10.25. I also included the javascript libraries so that we can do some validation and have ajax forms. This is all done by helper methods.

Listing 10.25 The helpers to include CSS and JavaScript

<%= stylesheet\_link\_tag "andreas01", :media => "screen,projection" %>

<%= stylesheet\_link\_tag "print", :media => "print" %>

<%= javascript\_include\_tag :defaults %>

The first 2 helper calls look in the folder /public/stylesheets for a file called andreas01.css and a file print.css. We can say for which media we want to include which stylesheet. And the third line includes the prototype libraries that are baked in the rails framework.

When we have this we need to tell our application controller that we want to use the same layout for our whole application. We can define other layouts etc at controller and even at action level. In our ApplicationController class we add the following code layout ‘default’.

So far so good, we’ve got a basic template working but now we still have to include our forms in the template. For this we need to identify which bit of html we need to replace. In the template I chose there is a div with id contentwide. We can just remove the html inside this tag and replace it with <%= yield %>. From now on we’re displaying our views inside the layout. There are plenty more tricks you can do with layouts but this is as far as our discussion will go in this book. For now we’ll move on to the views to see how to use those.

10.6.3 Representing data with Views

A layout is something you probably touch a couple of times during development of you app and changes to that template can have an application wide impact. With views the impact is much smaller but they do require more frequent changes. Whenever a model changes and you want that change to flow through to the UI you will need to change the view.

To help you to work in a DRY fashion the Rails frameworks provides partial views and they behave a little bit like an html include or a user control in Asp.NET. They allow you to group common logic in smaller files to allow for re-use. We’ll start our discussion by talking about the non-partial views and will then move on to a brief discussion of partial views. We will only discuss the view that is necessary to show the friends\_timeline controller action, pretty much like your homepage on the twitter website.

When we were discussing the controllers and we looked at the different formats, we saw how different extensions in the url could be mapped to different mime-types. The views futher build on this. Ruby has a built in templating format: the Embedded Ruby (erb) templating engine. You can recognize this type of templates because they tend to have erb as extension. The rails views allow you to specify for which format you need to take which file by adding the format in front of the .erb extension.

That means you could create a template to generate pdf documents and a template to generate html for the same action. Those 2 files would then be called my\_action.pdf.erb and

my\_action.html.erb. When you specify that the format has to be a pdf document then Rails will pick the pdf template and vice versa.

In our sample application we will only use the html variant. I added the previous tidbit so you know it is easy with rails to cater for different formats and different markups. You could also create an iphone mime mapping and then use my\_action.iphone.erb to generate the iphone specific html.

If we now return to our sample application then we know that in order to create this view we have 2 different pieces of functionality that need to be on that page. Users need to be able to see the status updates and users need to be able to post status updates to the website through the interface. Listing 10.26 shows the template for the friends\_timeline.html.erb file.

Listing 10.26: The view template for our homepage.

<h2>Timeline with friends for <%= @user.login %></h2>

<%= render :partial => 'update\_status' %>

<%= render :partial => 'timeline' %>

We first provide a header, in this case it is an indicator of what we’re displaying and we use <%= … %> to tell rails that we want to include the result from the called function into the rendered html. Next we include a partial that takes care of the sending of updates. The second partial we’re including is the one that contains the list of status updates for the logged in user and his friends. In the code sample above you can see one helper method the render method. The status update form, as shown in listing 10.27, uses more helpers.

Listing 10.27: The update status form

<div class="status-update" style="display:block;margin:5px;padding:3px;">

<% form\_for(@status) do |f| %>

<%= f.label :text, 'Update: ' %><br />

<%= f.text\_area :text, :cols => 70, :rows => 5, :style => "width:100%" %><br/>

<div style="display:block;text-align:right">

<%= submit\_tag "Update" %>

</div>

<% end %>

</div>

The beginning of this snippet is just plain old HTML. The next line contains the <% … %> notation. That is used to indicate that you’re going to capture a block and want to include that into the rendered html. The form\_for helper method captures a block and will surround the given block with the necessary markup to create a form that contains the elements from the block as well as the form tags and authenticity token. The block uses 1 parameter f, provided as a parameter to the form\_for method. That parameter denotes the form model, this model also has a bunch of helper methods to create form fields etc. We create a label for the text method on the status model as well as a text area for the text method of the status model. On this text area we set a couple of other attributes like the rows and cols. The last helper we see is the submit\_tag that will render out a submit button. Listing 10.27 shows the output of the partial when it is rendered.

Listing 10.27: A rendered partial

<div class="status-update" style="display:block;margin:5px;padding:3px;">

<form action="/statuses" class="new\_status" id="new\_status" method="post">

<div style="margin:0;padding:0">

<input name="authenticity\_token" type="hidden" value="d4d647a1dd10323dd96a7cfc544634f0d4f28ea6" /></div>

<label for="status\_text">Update: </label><br />

<textarea cols="70" id="status\_text" name="status[text]" rows="5" style="width:100%"></textarea><br/>

<div style="display:block;text-align:right">

<input name="commit" type="submit" value="Update" />

</div>

</form>

</div>

I guess there are no big surprises here. It renders out what we expected and told it too. Rails added a hidden field authenticity token. This field helps in ensuring that requests aren’t coming from a different site than yours. Of course this type of security is fairly easy broken but it is still a good idea to use it. There is one last partial left to discuss, the partial that is in charge of displaying the status updates in a list. The code for this partial is shown in listing 10.28.

Listing 10.28: The status update list partial

<div class="timeline">

Count: <%= @statuses.size %><br />

<% for status in @statuses %>

<%= render :partial => 'timeline\_item', **:locals => {:status => status}** %>

<% end %>

</div>

In the listing for the list partial we can see that first we display the total count of items. And then we’re going to display the list. We are looping through the collection of statuses and then we call a different partial that has the markup for a timeline item. The thing to notice in that listing is the use of the :locals hash to pass the currently selected status to the partial. The last listing about view templates is listing 10.29 and shows the content of the timeline\_item partial.

Listing 10.29: The timeline item partial.

<div class="timeline-item" >

<%= link\_with\_login(

status,

image\_tag('http://static.twitter.com/images/default\_profile\_normal.png'   
 , :alt => "profile image"),

:class => "profile-link") %>

<p>

<%= link\_with\_login status %>

<%= status.text %><br />

<%= status.created\_at %> from <a href="<%= status.source\_url %>"><%= status.source %></a>

</p>

</div>

In this partial there are couple of things worth noting. You can see there is a helper method link\_with\_login. This is a helper method we will define in a few moments. We have the status object as a locally scoped variable available in the partial and can access all its methods etc. You don’t have to use the provided helpers you can just use html and add the dynamic bits in between <%= … %> tags. The link\_with\_login helper method is shown in Listing 10.30 and is the only custom helper we’re going to need at this moment.

Listing 10.30: The link\_with\_login view helper method

module StatusesHelper

def link\_with\_login(status, text=nil, options={})

opts = options.collect {|k, v| "#{k}=#{v}"}.join(" ")

"<a href=\"/users/show/#{status.user.login}\" #{opts}>#{text.nil? || text.empty? ? status.user.login : text}</a>"

end

end

This helper method takes 3 parameters a status instance, an optional text and optional html options. We expect the options to be a hash so that we can generate html attributes from it. Helpers allow you to group pieces of functionality together that require more code than just some html generation.

10.7 Conclusion

In this chapter we took a brief tour of what is in the Rails framework. We only scratched the surface though. The rails framework is pretty big and it is a really nice option to build web applications. Rails shines through its extensibility model as well as through its ease of use.

We’ve seen how to create migrations to keep our database changes in one place and with the familiar ruby language. Migrations are a nice way to be able to replay and move back on database schema changes.

After the migrations we saw how the models and more importantly ActiveRecord is used in Rails as an ORM. We’ve seen how to use RSpec to create “unit tests” for our model code. We’ve also seen how we can make the model responsible for input validation and how serialization is customized.

When we were done with our tour of the models we took a look at what the job of a controller is and how we can use it to respond to different output formats. How we delegate calls to the models and how the controller decides to render views. We also talked about the filters to execute logic before and after a controller action.

The last topic of discussion were the views, in this topic we’ve seen how the views use the erb templating system to allow you to create templates for different types of markup formats. We’ve talked about partial views to keep your html templates DRY and we’ve seen how helpers are used to encapsulate common pieces of logic.

This was by no means a full discussion of all the features rails has to offer you, but it should serve as a decent introduction/appetizer. If you like what you’ve seen I suggest you get one of the rails books Manning has to offer.