Professional Developent with Rails

Fast, Light, Awesome.

Brian Hogan

Professional Developent with Rails: Fast, Light, Awesome.

Brian Hogan

Table of Contents

1. Introduction 1
    1. A behavior-driven approach to development 1
    2. Assumptions 1
    3. Conventions 1
    4. A little about Rails 2
    5. About the author 2
2. Workspace Setup 3
    1. Setting up Ruby and Rails 3
        1.1. Setting Up Windows 3
        1.2. Setting up OS X 3
    2. Updating RubyGems 4
        2.1. Windows 4
        2.2. Mac 4
    3. Installing Rails 4
        3.1. Windows 4
        3.2. Mac 4
    4. SQlite3 4
        4.1. Windows 5
        4.2. Mac 5
    5. Installing Subversion and GIT 5
        5.1. Windows 5
        5.2. Mac 5
3. Building A PasteBin 6
    1. Creating a new project 6
    2. Creating a new post 6
        2.1. Creating a model and a controller by creating a resource 6
    3. Test First, Code Later 7
        3.1. A clean test 8
    4. Implementing our validations 8
        4.1. Ensuring that postings are private by default 8
        4.2. Validate the title 9
        4.3. Validate the Code 10
    5. Running tests with Autotest 10
    6. Testing our "Create Post" interface 10
        6.1. The "New Snippet" form 11
        6.2. Saving a new post 13
    7. Displaying a snippet 16
        7.1. Testing the template itself 16
        7.2. Using a mock with the Show page 16
        7.3. Santizing output 18
        7.4. DRY up the tests 19
    8. Setting up the Site's Main Page 20
        8.1. Routing 20
        8.2. Setting a Default Route 20
    9. Test it out! 20
        9.1. Welcome To Rails 21
        9.2. Where's the submit button? 21
    10. Tweaking the Interface 22
        10.1. Displaying your Error Messags 22
        10.2. Prettying Up the Output 22
        10.3. Add a link to create a new post 22
    11. Writing an Integration test 23
        11.1. The flow 23

11.2. Use Webrat to test the flow 23
12. Summary 24
Index 25

Chapter 1. Introduction

Rails is popular because you can use it to build sites quickly, but experienced Rails developers know that the real power in Rails lies in the ability to build *quality* applications quickly. Those screencasts where they build a blog in ten minutes look pretty neat, but scaffolding and code generation only take you so far. With an extra ten minutes, you can turn a barely-working app into a rock solid app, and this book will show you how that's done.

1. A behavior-driven approach to development

Most books out there walk you through building an application quickly and then talk about testing as an afterthought. This book will make testing the central point, and we'll explore various methods of testing Rails applications as we progress through our project.

Instead of focusing on what the code does, we'll focus on user interaction and how the application should work. We'll write simple user stories as we go to capture our requirements, and we'll turn those into tests for our controllers and models. We'll even make sure that important user interface elements exist and that they perform the correct functions.

2. Assumptions

First, I assume you're an experienced programmer. The concepts in this book aren't going to be aimed at the absolute beginner. You certainly don't need to be profiicient with Rails, but web development skills such as SQL, HTML, and some server-side programming language are key.

I'm going to make reference to models, views, and controllers, so I'll assume you know something about the MVC pattern and what the functions of those pieces should be.

I'll be talking about REST a lot in this book. I assume you've at least heard about it. There are enough resources out there that cover REST in great detail, so I won't repeat them here. I want the focus to stay on rapid development with testing.

Finally, I assume that you have an open mind, and that you'll be receptive to learning something new. I'll do my part to introduce topics at a progressive flow, saving the really complex things for later, but I'm asking you up front not to jump around. Read this thing from the beginning to the end and work through the examples.

3. Conventions

There's going to be a lot of commands in this book. I'm going to refer to the *terminal* a lot. When I do, that means I want you to be using a command prompt or Terminal session . Windows users may not be too used to doing things with the command prompt, but they will be after working through this book.

Links to files are going to be relative to the Rails' application root. Let's say you had a Rails app at `c:\rails\pastebin`. If I told you to open `app/views/posts/new.html.erb`, you should look for that file within your application's directory.

Code snippets will all have line numbers. These don't correspond to line numbers in the actual program. They're merely there to help you tell when code in this book crosses pages, and will make it easier to tell when something wraps.

Here's an example:

```
1 def test_should_do_something_cool
```



```
 2           p = Post.new
 3           p.title = "This is the title of the snippet"
 4           p.code = "Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do
eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis
nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute
irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur.
Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit
anim id est laborum.
 5           "
 6           assert p.save
 7 end
 8
```

4. A little about Rails

Rails is an MVC-based web development framework written in Ruby. It powers, or helps to power, sites like Hulu, yellowpages.com, Twitter, Basecamp, Penny Arcade, Wis.dm, and many many more. It takes a very opinionated approach to web application development which means you can develop more without worrying about where things go or what things should be called.

Much of the way Rails works is based on the Ruby language and its ability to do a lot of dynamic operations at runtime. The framework takes some of the mundane work away from you if you follow the conventions because Rails leverages Ruby to write code for you at runtime. You won't need to write too many SQL statements because the common cases are all taken care of for you.

You should be skeptical of this, because skepticism has made Rails extremely powerful as more bright minds come to the table and work to iron out flaws.

By embracing conventions and opinons, you are free to focus on your customers' needs. It's easier than ever to bring a product to market with Rails.

5. About the author

I started doing basic web design and development in 1995 and began building dynamic web applications soon after, working primarily with ASP and Microsoft technologies before embracing PHP and finally Ruby on the Windows platforms. I enjoy writing and teaching about technology and have enjoyed working with developers all over the world on Ruby-related projects.

I hope that everyone who reads this takes something valuable away.

Chapter 2. Workspace Setup

Setting up your workstation so you can build Rails applications involves several steps, and the steps differ depending on your operating system. This section describes the simplest methods of installation for Mac OS X and Windows.

1. Setting up Ruby and Rails

1.1. Setting Up Windows

it's easy to set up your Windows machine in order to develop a Rails applications. While there are many prepacked solutions, we'll do the manual installation, as it is the most flexible.

Just download the One Click Ruby Installer[1] and install it using the default options and you'll have Ruby installed.

1.2. Setting up OS X

So you have a shiny Mac with Leopard on it and you're ready to learn Rails? Well, it may surprise you to learn that Leopard actually ships with Rails preinstalled. The folks at Apple were so smitten with Ruby on Rails that they decided to include everything you'd need to build a simple application. Unfortunately most of it is woefully out of date.

MacPorts

In order to install some of the dependencies you need to work with Rails, you need to install the XCode Tools from your Tiger or Leopard CD. You can also create an Apple Developer Account and get them there. Once you have XCode Tools installed, download Macports from http://www.macports.org/install.php and run the installer.

From your terminal, type `port help` to make sure that the port command is available.

Setting your path manually

The MacPorts installer is supposed to make changes to your path, but it often times can't for any number of reasons. To fix the problem, open a terminal and enter the following commands that will append the appropriate line to your `.bash_profile` account.

```
cd ~

touch .bash_profile

cat >> .bash_profile <<EOF

export PATH=/opt/local/bin:/opt/local/sbin:$PATH

EOF
```

Close your terminal and open it again so the changes take effect.

Test out MacPorts again by typing `port help` into the newly opened *Terminal*.

[1] One Click Ruby Installer: http://rubyforge.org/frs/download.php/43428/ruby186-27_rc1.exe

2. Updating RubyGems

RubyGems, or more commonly known as Gems, are libraries that developers use to build better applications. Libraries like the Twitter gem, for example, let a developer post message to Twitter, or retrieve a list of their friends. The Rails framework is distributed as a Gem, and so we use the RubyGems package manager to upgrade Rails and its dependencies.

2.1. Windows

Open a Command Prompt and type

```
gem update --system
```

2.2. Mac

Open *Terminal* and type the following command to update RubyGems to the latest version.

```
sudo gem update --system
```

3. Installing Rails

You'll use RubyGems to install the Rails framework and its dependencies. When you run the installation command, Rails installs and then the RubyGems system builds documentation on your local machine. If you have a previous version of Rails already installed, the installer will leave it alone, allowing you to develop applications for old and newer versions of Rails.

3.1. Windows

Open a new Command Priompt and type

```
gem install rails
```

to install the framework.

3.2. Mac

Macs already have Rails installed, but it's out of date. Update Rails by entering

```
sudo gem install rails
```

in a *Terminal* window.

4. SQlite3

Rails development got a lot easier when SQLite3 became the default database for new projects. *SQLite3* is a small file-based database that, while not really a good fit for production applications that need to scale, is great for doing a small rapidly-designed application. With SQLite, there's no setup and no permissions to worry about when you start your Rails project.

4.1. Windows

Visit http://www.sqllite.org and download the SQLite3 DLL file. Extract it to a location on your path (c:\ruby\bin is a great location).

Open a new *Command Prompt* and type `gem install sqlite3 --version=1.2.2`

The version number is important because it's the last version that worked for Windows.

4.2. Mac

Install SQlite3 by typing `sudo port install sqlite3` in a Terminal window.

5. Installing Subversion and GIT

Subversion and Git both version control systems that allow developers to "check in" and "check out" their code. A versioning system also allows you to revert code changes to previous versions, and is an essential component when doing any sort of professional development.

The Rails world is in a state of flux now, with developers leaving Subversion and moving towards Git. There are advantages and disadvantages to each, but the reason you need both is so you can easily consume plugins and libraries developed by others.

Rails plugins are usually distributed via Subversion or Git, so you need both installed on your workstation, even if you never plan to use them for anything else.

5.1. Windows

Download and install Subversion from http://www.collab.net/downloads/subversion/ and install it, accepting all of the defaults.

Download Msysgit from http://msysgit.googlecode.com/files/Git-1.6.0.2-preview20080923.exe and install it, again accepting the default options.

5.2. Mac

Set up SVN with `sudo port install subversion`

Set up Git with `sudo port install git-core +svn`

Chapter 3. Building A PasteBin

We'll build a simple Pastebin application which people can use to share code snippets with others. Pastebins are very popular in forums and help channels because they provide an easy way for you to show example code or ask people what's wrong with what you did, without flooding the chatroom with many lines of code.

This chapter will walk you through getting the initial pastebin application started. At the end, you'll be able to create new pastes and share them with others.

1. Creating a new project

When you paste your code to a pastebin, you usually are redirected to a new page that shows the code you pasted. You can then give that URL to someone else and they can view your code. We'll implement something just like that, with no security mechanism or required logins.

Create a new Rails project called *pastebin*. I'll be using command-line examples in this book, but you can feel free to use your IDE if you'd like.

At a terminal window or command prompt, type

```
rails pastebin
```

This creates a new Rails application using the *SQLite* database. Change to that directory by typing

```
cd pastebin
```

2. Creating a new post

Posts are the heart of our application, so it makes sense to tackle thse first.

Let's take a look at a typical story which we can use to drive the development of this project.

```
In order to share a snippet with others on the web
as a guest user
I can visit the site
 and I see the new snippet form
 and I enter a title
 and I enter the code
 and I click Submit
Then a new public snippet is saved
  and I see the new snippet
  and the code is nicely formatted
  and I see a link to create another snippet.
```

We can learn a lot from that story. We know what the required fields are and what they're called, we get a glimpse of what our form should look like and what our workflow will be, and we also can determine our database structure.

2.1. Creating a model and a controller by creating a resource

A *resource* , or more specifically, A RESTful resource, according to the Rails documentation, is " in basic terms, is something that can be pointed at and it will respond with a representation of the data requested.

In real terms this could mean a user with a browser requests an HTML page, or that a desktop application requests XML data. " A Rails resource consists of a controller and, optionally, a model. In our case, we need a controller to handle requests and a model to represent the post.

Models in Rails hold our business logic., and they also talk to the database. Our data validations and our querying will all be done in our models. Models are just classes that extend from a Rails framework class called `ActiveRecord::Base`, but instead of manually creating the file, we'll use a generator to create the model. A generator also creates the tests we'll use to unit test the model, and it will generate a script to set up our database table that will interact with this model.

At the command prompt, in your application's folder, type

```
ruby script/generate resource Post title:string code:text private:boolean
```

The generator creates several files, including a migration.

Migrations

Migrations are database definitions written in Ruby. They make it very easy to define your database tables no matter what type of database backend you'll be using. Our example uses SQLite3, but when it comes time to move to production and use MySQL or Oracle, all we have to do is run a script which will recreate the database on the new target database. The target database is specified in `config/database.yml` and can be different for production, testing, and development.

When you created the model with the generator, you passed along the database fields on the command line. This generator automatically adds two additional fields to the table which will record the created at and updated at dates of the record automatically.

We'll visit migrations more throughout this example project, as we use them to add indexes and add columns to tables.

Migrations need to be applied though. At your command prompt, type

```
rake db:migrate
```

and the tables will be created in your development database.

Update your testing database!

Rails uses a separate database for unit testing, and you need to keep it up to date. The `rake db:migrate` command only operates on the default development database. You should update the test database with

```
rake db:test:clone
```

3. Test First, Code Later

One of the things you hear in the Rails community is TATFT, which stands for "Test all the f*@#king time.". The best Rails developers are known for writing unit and functional tests for all of the code they write. At first, this can seem like an overwhelming task, but if you get into the habit of writing tests before you write your actual program code, you will very quickly become proficient at TATFT, but you will produce better-quality applications.

If the concept sounds foreign and abstract, then that's because it can be.

3.1. A clean test

Let's write a simple baseline test that will always prove our model works like we expect. We'll set the title and code fields and then we'll try to save it. The save should be successful, and then we havea nice safeguard against another developer who might change validations or callbacks without changing tests.

Open `test/unit/user_test.rb` and add this method:

```
1  def test_should_create_new_post_when_all_requirements_have_been_fullfilled
2     post = Post.new
3     post.title = "My test"
4     post.code = %Q{
5       def test
6         puts "Hello world"
7       end}
8    assert post.save
9  end
```

Run the test from your terminal with

```
rake test:units
```

We don't have anything in our code that will prevent this test from passing, so we'll know right away if we introduce any problems down the road.

4. Implementing our validations

We'll test each validation rule separately before we implement the rule. Good testers only test one thing in each test case because it makes it more clear where the fault is when new errors occur.

4.1. Ensuring that postings are private by default

One of the rules in our story states that new postings should be private by default, meaning that they shouldn't show up in the listing of public posts. We will eventually make these available via a hashed url. These posts will be private in the sense that you have to know the URL to the posting if you want to see it.

When writing tests first, you don't need to know how this behavior will get implemented. Right now you've probably already thought of at least three ways to implement this rule, but you need to concentrate on what the outcome should be instead of the implementation. You work out the implementation only after you have good tests in place.

```
1       def test_should_set_private_to_false_by_default
2         p = Post.create :title => "foo", :code => "bar"
3         assert p.private == false  # because Ruby treats nil and false as false
4       end
5
```

In this test, we're just creating a new post and then we're checking to make sure that the private method on that object returns false. That's it. We don't care how it happened, we just care that it worked.

Run

```
rake test:units
```


to ensure your test breaks. It's always a good idea to run your tests after each change, even if you know it will fail, because there are situations where you can write a bad test that passes when it shoudln't.

Open up the file `app/models/post.rb` and add a private method that sets the private field to false, and add a callback to invoke that method on creation.

```
1     before_create :set_privacy_off_by_default
2
3       private
4
5         def set_privacy_off_by_default
6           self.private = false
7           return true
8         en
```

Run

```
rake test:units
```

and this time your tests should all pass. You can move on to the next requirement.

Tip

You know you're done implementing when your test passes. Testing before you code is a great motivator because you really see how much you've accomplished, even if you can't show visual results to someone yet.

4.2. Validate the title

We'll validate that the title must exist by attempting to save a record with no title. One approach would be a test like this:

```
1       def test_should_require_presence_of_title
2         p = Post.create(:code => "blah")
3         assert p.valid? == false
4       end
5
```

This test sets all the other fields correctly except for the title. It's a valid approach but one I dislike because it is kind of brittle. If you added a new field that was required besides the title and code, this test would break for the wrong reason and you wouldn't know why. You could solve that problem by making a test helper that acts as a new project factory so you only have to change the valid fields in one place, but you can do a much safer and less brittle test by just learning a little about the internals of Rails validations.

When you validate fields using ActiveRecord, you simply store error messages in a collection which can be accessed by the object's `errors` method. This collection object has methods that give you easy access to the individual error messages. Armed with this knowledge, you can test for specific error messages if you know what to look for.

Add this test to `test/unit/post_test.rb`

```
1       def test_should_require_presence_of_title
2         p = Post.create
3         assert p.errors.on(:title).include? "must be filled in"
4       end
5
```

The `on` method returns either a string or an array depending on how many errors are associated with that field. Fortunately the `include` method exists on both `String` and `Array` so we can test the value of the message regardless of its type.

Run

```
rake test:units
```

and the test fails, which is expected.

Open up `app/models/post.rb` and add a validation rule for the title:

```
1     validates_presence_of :title, :message => "must be filled in"
```

Run your tests again with `rake test:units` to ensure that everything still works.

4.3. Validate the Code

You'll ensure that a record is invalid if the code field is left blank by writing almost exactly the same test as the previous example.

Add this test to `test/unit/post_test.rb`

```
1       def test_should_require_presence_of_code
2         p = Post.create
3         assert p.errors.on(:code).include? "must be filled in"
4       end
```

To make this test pass, open up `app/models/post.rb` and add a validation rule for the title:

```
1       validates_presence_of :description, :message => "must be filled in"
```

Run your tests again with `rake test:units` and everything should pass. This process may seem repetitive, but that's because it is. Let's make this repetition less painful by making those tests run for us whenever we save files.

At this point, the business logic is implemented. We've written tests to ensure that invalid records aren't saved, and a test to prove that a valid record does save.

5. Running tests with Autotest

If you test all the time, you should use *Autotest* which is part of the *ZenTest* gem.

Provided that you installed the gem, you can start the test runner from your command prompt with `autotest`. This command runs all of your tests the first time, and then only runs tests affected by your changes on subsequent runs. From this point on, I'll assume you're using *autotest*.

6. Testing our "Create Post" interface

We've got solid backend logic and now we can safely move on to working with the front-end pieces. Up to now, you've used unit tests, which are a fundamental part of Ruby's built-in test suite. Controllers in Rails are a little more complicated, and to make testing easier, Rails provides a different kind of test, called a *functional test*.

Functional tests are like unit tests for controllers. Each test case in a functional test will test a specific action or method of a controller.

6.1. The "New Snippet" form

There are a few things you have to test for when you're displaying a page to the users. First, you want to make sure you're showing them the right template, and then you want to make sure that you're showing the right fields on the page. You might also want to test to make sure that the form submits to the appropriate server-side function. All of these are easy to test for in a very short amount of time with functional tests.

Ensuring the right template is displayed

If we follow the RESTful design principles of rails, then the New page should be displayed after sending a GET request to the `new` action. In turn, the `new` should, by default, render a template called `new.html.erb`

Armed with this knowledge we can write the first test. Add the following test to `test/functional/posts_controller_test.rb`

```
1           def test_display_new_post_form
2             get :new
3             assert_template "new"
4           end
```

Autotest should pick up the changes and run the new test. You'll get an error because you haven't actually implemented the `new` action in the controller

Add this code to `app/controllers/posts_controller.rb`:

```
1
2           def new
3           end
```

The method is empty, but there's a hidden bit of default behavior that goes on here. If no specific call to `render` exists in the controller, it will simply look for a file in `app/views/[controller_name]/[action_name]`. This assumption keeps you from having to write extra code, and is just one of the many opinions about Rails you'll encounter.

After you save the file, *Autotest* will complain once more, telling you that the template was not found. If you look in `app/views/posts`, you will not see a `new.html.erb` in that controller. Create one right now and then Autotest should be happy..

Note

It's worth noting here that the empty `new` action could be omitted from the controller. Rails will take the request and look for an action in the controller first, but if it doesn't find one, it will go look for a view. You should leave this empty method alone though becuase we'll be adding to it in the next step.

Testing form actions

Web applications all work pretty much the same way. Your users interact with you through links and forms and you take that data and respond to it in some fashion. When you build forms, it's really important to make sure that the form field goes to the right server-side action.

In a RESTful Rails app, the "New" form should POST to the `create` action of the controller. We'll use a special assertion called `assert_select` to read through the HTML rendered by Rails and make sure it has what we expect.

Add this test to `test/functional/posts_controller_test.rb`

```
1
2           def test_new_post_form_should_have_a_form_that_posts_to_create
3             get :new
4             assert_select "form[action=?][method=post]", posts_path
5           end
```

The `assert_select` helper uses CSS selector syntax to grab elements from HTML files. The above example looks for a form that looks like

```
<form method="post" action="/posts">
```

The `posts_path`method is part of the resource-based routing that was configured for you by Rails when you generated the resource.

Tip

Learn more about resource routing at

Save the file and watch `Autotest` choke again because it doesn't find the form on the page. Open up the `new.html.erb` and add this code:

```
      <% form_for @post do |f| %>

       <% end %>
```

Once you save this file, your tests will fail once again because it will be trying to call a method on an object that doesn't exist. This form helper we added uses an instance variable called `@post` which was never previously defined. In this view, the variable was declared on first access and becomes a default object, and so it doesn't have the methods that the form helper expects.

Change the `new` method of `app/controllers/posts_controoler.rb` to:

```
          def new
            @post = Post.new
          end
```

and that will cause your tests to pass once again.

Testing the existence of form fields

It's also important that the form field names match what the receiving server-side code expets, and the easiest way to do that is to decide on them ahead of time and write a test.

Our story says that we need fields for the name of the snippet and the code. The name of the snippet will probably only be one line, but the code will often times require a multiline form field, or a `textarea`. We can use `assert_select` to make sure that the tags exist on the page. Open up the `test/functional/posts_controller.rb` and add this code:

```
1           def test_new_post_form_should_have_title_and_code_fields
2             get :new
3             assert_select "form" do
4               assert_select "input[type=text][name=?]", "post[title]"
5               assert_select "textarea[name=?]", "post[code]"
6             end
7           end
```

The form field names generated by the Rails helpers we'll use in the view place the fields in a hash. So instead of looking for a name of `title`, we'll be looking for `post[title]`.

Open up the `app/views/posts/new.html.erb` and change the code to this:

```
<% form_for @post do |f| %>

 <p>Title<br />
   <%=f.text_field :title %>
 </p>

 <p>Code<br />
   <%=f.text_area :code %>
 </p>

 <% end %>
```

This adds the fields to the page. It won't look pretty by any means, but at least your tests pass. You could pass this page on to a designer and they could improve it and you'd know if they made any crucial mistakes because your tests would start failing.

6.2. Saving a new post

The form sends a POST request to the **`posts`** controller, and one of two things can happen when the data is received: it will either save the record or redisplay the form and tell the user why it didn't work.

Note

Our story never covers the alternate flow. It's up to us to decide what happens here.

We already tested data validations in our unit tests, so we only need to concern ourselves with controller logic here. If the record saves successfully, redirect to the show page of the snippet so the user can send the URL to their friend. If it fails, redisplay the form and show the errors. We only have two tests to write here, and that's it.

Mocks and Stubs

Newcomers to test-driven development might be tempted to construct a valid POST request to test the positive case and then construct an invalid POST request to test the negative case. However, as requirements change, the valid POST request might start looking different. Also, it's not the controller's job to ensure that the data is valid or not, it's the model's responsibility to do that. Controllers should just pass data to models and then return a response.

However, you can't just ignore the models. If you don't pass in valid data, the models are going to throw errors and your tests will break. The solution is to introduce mocks and stubs.

Mocks are phony objeccts that you use in place of your real objects. Mocks are popular in Rails development because they save time. If you use mock objects, you can pretend to hit the database or a remote web service without actually taking the processing time to do it.

Stubs are phony methods on objects, either real ones or mocks. They are the easiest to use and are the quick-win solution for our little problem.

Open up the file `test/test_helper.rb` and add the line

```
require 'mocha'
```

to the top of the file. The *Mocha* library makes it easy to create mocks and stubs in Rails' tests. There are other libraries out there but I'll be focusing on this one for the rest of the excercise.

Implementing behavior with Stubs

Let's review the common record creation pattern in Rails:

```
1           @post = Post.new(:title => "test", :code => "<html><body><div><p>a
simple page</p></div></body></html>"
2           if @post.save
3             # redirect to the show page
4           else
5             # render the same page
6           end
7
```

You'll find this pattern used in many Rails controllers. Take a look at the logic flow. If the `save` method of the model returns `true`, the controller redirects. If it returns `false`, it redisplays the form.

From this example, it's clear that we can avoid all of the model logic simply by forcing the `save` method to return `true` and then ensuring that the redirection occurs.

Add this test to `test/functional/posts_controller.rb`:

```
1          def test_should_redirect_to_show_page_if_creation_is_successful
2             Post.any_instance.stubs(:save).returns(true)
3             Post.any_instance.stubs(:id).returns(1)
4             post "create"
5             assert_redirected_to post_url(1)
6           end
```

The first line of the test replaces the `save` method of any instance of the `Post`model with a stubbed version which will return `true`

The second line does the same thing but with the `id`. The reason for this is that since we've stubbed out the `save` method, the record never gets saved to the database and thus the record's `id` field never gets set. We'll need that ID so we can use it in the redirection URL to display the new record.

Tip

Stubs make it easy for us to return whatever data we want without having to worry about what's actually in the database.

The third line issues a POST request to the controller. Normally a POST request would contain the posted parameters, but since we're stubbing out most of the backend behavior and only testing the controller's response, we don't care about the parameters and can leave them blank.

Finally, we test to ensure that the redirect to the show page worked. [1]

Save the test file and Autotest will bark at you telling you that you don't have a `create` action in your controller. Go add one like this:

```
1           def create
2             @post = Post.new(params[:post])
3             if @post.save
4               redirect_to post_url(@post)
5             end
6           end
```

Notice that we're not handling the `else...` part yet. We didn't write the test for it so we don't implement it yet.

Save the controller and the tests all pass.

The test for the unsuccessful creation is almost identical, but you don't have to stub out the `id` method. Add this test to `posts_controller_test.rb`:

```
1           def test_should_show_the_create_form_page_if_creation_fails
2             Post.any_instance.stubs(:save).returns(false)
3             post "create"
4             assert_template "new"
5           end
```

Once you implement this test, Autotest is going to give you a long stack dump. If you sift through it, you'll see this message:

```
        Missing template posts/create.html.erb
```

We never implemented the `else...` condition so when the record creation failed, it just did the default behavior of rendering a template with the same name as the action, which doesn't exist.

Caution

Unit and functional tests don't just pass or fail. Sometimes they raise exceptions which you'll need to sift through to figure out what happened.

Implement the `else...` condition for your controller. The entire controller action should now look like this:

```
1           def create
2             @post = Post.new(params[:post])
3             if @post.save
4               redirect_to post_url(@post)
5             else
6               render :action => "new"
7             end
8           end
9
```

[1] Remember that the `post_url` method was provided by the Rails RESTful routing mechanism. It requires a single parameter, the ID of the record that you'll eventually process.

At this point, your tests should pass.

While our tests pass, this application fould throw an exception if we tried to use it from the browser because we never implemented the `show` action of the controller, which is what our process is supposed to call upon successful completion. Functional tests aren't meant to be used to test an entire process end-to-end. We'll discuss integration testing later.

Note

You should redirect to a new action after you modify records in your application. This

7. Displaying a snippet

Implementing the show action won't take too long. All we have to do is locate the record, display the page if the record exists, and display a 404 page if it isn't.

7.1. Testing the template itself

Before we get into the really good stuff, we need to do the standard tests first. First, we need to ensure that the right template gets called when we make the request to the `show` action.

```
1
2         def test_should_display_the_show_page
3           get :show, :id=>1
4           assert_template "show"
5         end
6
```

As you can guess, the tests fail because we don't have the `show` action implemented. Add it to `app/controllers/post_controller.rb`

```
1
2         def show
3         end
4
```

and you should get the standard message about the template not existing. Create a new file called `app/views/posts/show.html.erb` and the errors will go away.

7.2. Using a mock with the Show page

We need to test that a record can be displayed by the controller and rendered by a view. We could populate our test database with some records and use one of those in our tests, but if we didn't hit the database when we tested record creation, why should we do that now when we're testing retrieval?

We should be able to assume that the built-in ActiveRecord library knows how to retrieve records from a database as long as we use the provided Finder methods. All we need to test is that the controller responds to the request, and that it uses the right ID when looking up the record. We can use mocks once again.

Create this new test in `test/functional/post_controller_test.rb`:

```
1
```

```
  2         def test_should_find_record_when_displaying_post
  3           post = mock("test") do
  4             stubs(:title).returns("test")
  5             stubs(:code).returns("<b>win!</b>")
  6             stubs(:id).returns(1)
  7           end
  8           Post.expects(:find).with("1").returns(post)
  9           get :show, :id => "1"
 10         end
 11
 12
```

This test sets up a mock object called **`test`** and then sets up stubs for the return values. Then we set an **`expectation`**, a special kind of stub, on the **`find`** class method of the **`Post`** model.

This particular expectation is very specific. We're telling the test that we want the **`find`** method to be called with a specific value (**`'1'`**), and that it should return our mocked model.

> **`expects` vs. `stubs`**
>
> The difference is in the details. Stubs are simply replaced methods that may or may not return values and may or may not be called. They're great for replacing things that are not part of the implementation details or when you don't care if the program uses them.
>
> Expectations are for those cases where you decide that the stubbed methods *must be called*. When you set up an expectation, your tests wil l throw exceptions if that expectation is not met. For example, if you write this expectation:
>
> ```
> Post.expects(:find).with(1).returns(post)
> ```
>
> and your implementation made this call:
>
> ```
> @post = Post.find(2)
> ```
>
> the test would throw an exception. In fact, it's so picky that it expects the type to match, so even
>
> ```
> @post = Post.find('2')
> ```
>
> fails to match.

Implement the controller action like this:

```
@post = Post.find(params[:id])
```

Once you implement that method, your current test passes, but your previous test does not. It fails with a `RecordNotFound` because your previous test didn't stub out the **`find`** method. Replace it with the same mock and stub from before to get it passing.

```
  def test_should_display_the_show_page
    @post = mock("test") do
      stubs(:title).returns("test")
      stubs(:code).returns("<b>win!</b>")
      stubs(:id).returns(1)
    end
    Post.expects(:find).with("1").returns(@post)
    get :show, :id => "1"
    assert_template "show"
  end
```

7.3. Santizing output

Any time you allow users to put input into a page that others can publicly see, you are inviting trouble if you don't sanitize the output. There are numerous vulnerabilities that you hav to watch out for, but Rails provides a very simple mechanism to sanitize output - the `h` method.

The `h` method is a *helper*, a term for methods in Rails that are designed to be used in views. Helpers generally output strings, either in plain text or HTML. `link_to`, `image_tag`, and `form_for` are all examples of helpers.

We know we can test to see if a method was called by using an expectation, but in order] to do that here, we need to know what class the helpers are attached to at runtime. It turs out that the helper modules are mixed in to the template renderer which is an instance of `ActionView::Base`. Knowing that, we can add this test to `test/functional/posts_controller_test.rb:`

```
  def test_code_should_be_sanitized_on_show_page
    @post = mock("test") do
      stubs(:title).returns("test")
      stubs(:code).returns("<b>win!</b>")
      stubs(:id).returns(1)
    end
    Post.expects(:find).with("1").returns(@post)
    ActionView::Base.any_instance.expects(:h).with(@post.code)
    get :show, :id => "1"

end
```

Now, we just modify the `app/views/posts/show.html.erb` file and sanitize the code field.

```
<h1><%=@post.title %></h1>
<hr />
<div id="code">
  <%=h(@post.code) %>
</div>
```

Warning

You really should sanitize every field provided by a user that automatically gets posted to the web. That includes flash messages, links, and page titles. If a user can type something in that you will display, sanitize it.

Why don't we sanitize the data on the way in instead?

Sometimes that's a valid option. However, we're building a pastebin, and we want the code to be preserved in its original form. We sanitize the output on the HTML page, but we may want to provide the raw format as text. (In fact, we'll do that later on!) Sometiems it makes sense to store the user's input as they entered it, even if it means we have to be more careful when we present that data.

7.4. DRY up the tests

We've copied the same stubbing and mocking code to four separate tests now. It's time to refactor our tests. We don't want brittle tests. How can we do that effectively though?

Take a look at the code for stubbing and mocking. All of these tests are the same, but each one has a different expectation or assertion. Let's just throw the common stuff into a new method. At the bottom of the `test/functionals/posts_controller_test.rb` file, add this method:

```
def mock_post
  post = mock("test") do
     stubs(:title).returns("test")
     stubs(:code).returns("<b>win!</b>")
     stubs(:id).returns(1)
  end
end
```

Then refactor your tests to use this new method.

```
def test_should_display_the_show_page
  @post = mock_post
  Post.expects(:find).with("1").returns(@post)
  get :show, :id => "1"
  assert_template "show"
end

def test_should_find_record_when_displaying_post
  @post = mock_post
  Post.expects(:find).with("1").returns(@post)
  get :show, :id => "1"
end

def test_code_should_be_sanitized_on_show_page
  @post = mock_post
  ActionView::Base.any_instance.expects(:h).with(@post.code)
  Post.expects(:find).with("1").returns(@post)
  get :show, :id => "1"
end
```

Tip

Methods that don't start with **`test_`** are not included in the test suite, so you can make as many test helpers as you'd like. You can also include these types of methods in `test/test_helper.rb` and share them across tests.

Have you looked at *Autotest* in a while? Everything should still be passing if you've been following along.

8. Setting up the Site's Main Page

When people pull up our new site, we'd like them to quickly be able to add a bit of code, so it makes sense that the first thing they see is the new posting page. We can do that with Rails' routing mechanism.

8.1. Routing

Rails provides a URL rewriting mechanism called *routing* which both constructs and recognizes URLs requested by the user. The routing mechanism is responsible for extracting parameters from the URL, such as the post ID from `/posts/1` as well as determining which controller and action are associated with each URL.

When you program using resources, a lot of this magick happens behind the scenes. You can take advantage of some of the more advanced features of routing to really change how your URLs work.

8.2. Setting a Default Route

Open the file `test/functional/posts_controller_test.rb` and add this new test:

```
1
2         def test_default_route_should_map_to_new_post
3           opts = {:controller => "posts", :action => "new"}
4           assert_recognizes opts, ""
5         end
6
```

This test ensures that requests to `/` will be directed to our `new` action which will display the login page. Unfortunately it doesn't work quite yet because there's no route defined to actually direct the request.

Open the file `config/routes.rb` and locate this line:

```
1
2         # map.root :controller => "welcome"
3
```

Change it to

```
1
2          map.root :controller => "posts", :action => "new"
3
```

9. Test it out!

We've got a fully tested application at this point and we've never once opened up a web browser to see it in action. Let's do that now. Open up a new command prompt in your application's directory and launch the built-in server

```
ruby script/server
=> Booting Mongrel (use 'script/server webrick' to force WEBrick)
=> Rails 2.1.0 application starting on http://0.0.0.0:3000
=> Call with -d to detach
=> Ctrl-C to shutdown server
** Starting Mongrel listening at 0.0.0.0:3000
```

```
** Starting Rails with development environment...
** Rails loaded.
** Loading any Rails specific GemPlugins
** Signals ready.  INT => stop (no restart).
** Mongrel 1.1.4 available at 0.0.0.0:3000
** Use CTRL-C to stop.
```

The server starts up on port 3000, so navigate to http://localhost:3000 and test out your app. What you find will surprise you.

9.1. Welcome To Rails

The "Welcome to Rails" page is actually a static file located at `public/index.html`. Rails has a content caching mechanism that serves up files from the public folder first, before trying to serve requests via Rails. Since you specifieed no file, the request first looked in the public/ folder for a file called `index.html`, which is a common assumption among web servers for a default page. Had it not found the page, your request would have been passed on to the routing. This approach makes it possible for you to use Rails to build and generate static pages which can then be served by Apache or another static web server, reserving Rails for truly dynamic operations.

This default page should be removed though for our purposes. We can certainly take advantage of this content caching ability later. Delete `public/index.html` and refresh your browser.

9.2. Where's the submit button?

The first thing you probably noticed about the form once it displayed was that there's no submit button so you can't use the site! We never actualyl put one on the page, and we never wrote a test for it.

It's important to realize that testing first won't catch everything. Testing first is only part of the application quality control process. You should be constantly testing, and not just relying on automated tools. A combination of automated testing, browser testing, and even usability testing by others will lead to a better experience for your users.

If you'd like to add a test for the button to your controller test, you could do this:

```
1
2         def test_new_post_form_should_have_submit_button
3           get :new
4           assert_select "form" do
5             assert_select "input[type=submit]"
6           end
7         end
8
```

Then change `app/views/posts/new.html.erb` to this:

```
1
2
3         <% form_for @post do |f| %>
4
5          <p>Title<br />
6          <%=f.text_field :title %>
7          </p>
8
9          <p>Code<br />
10         <%=f.text_area :code %>
11         </p>
```

```
12
13          <%=submit_tag "Post it!" %>
14
15        <% end %>
16
17
```

Redisplay your form in the browser and you should have your submit button.

10. Tweaking the Interface

The application is functional, but there are a few issues that we should address.

10.1. Displaying your Error Messags

Submit a post without entering a title or code and you'll just see a blank page. That's less than helpful. Instead, you want to display the error messages to the user.

Note

I'm not convinced it's worth writing a test for this first, but you certainly could, since you already know how. Do it the same way you testsed for sanitization wit the **h** method.

Add this line to the top of `app/views/posts/new.html.erb`

```
1
2          <%=error_messages_for "post", :title =>"foo" %>
3
4
```

Now you should be able to post new snippets.

10.2. Prettying Up the Output

Create a new post and you'll see that the results page doesn't respect whitespace. Of course, we never told it to. Change the output of the `app/views/posts/show.html.erb` page to this:

```
1
2         <h1><%=@post.title %></h1>
3         <hr />
4         <div id="code">
5           <pre><%=h(@post.code) %></pre>
6         </div>
7
8
```

Wrapping the code with `<pre></pre>` tags fixes the output nicely.

10.3. Add a link to create a new post

Once a user has posted some code, they're left with no way to paste another snippet unless they hit the browser's Back button. It's trivial for us to add a link to create a new post on the `show.html.erb` page. Open it up and add the following code at the bottom of the file:

```
1
2         <hr />
3         <p><%=link_to "Create a new post", new_post_path %></p>
4
5
```

The method `new_post_path` is a helper that returns the URL `/posts/new`. It's another one of the helpers provided by the Rails routing system and the `map.resources :posts` line in `config/routes.rb`.

11. Writing an Integration test

We can script the entire process of visiting the site, creating a post, nad reviewing the results so that we can have another layer of testing that ensures the entire thing works together. Rails provides *integration tests* for this purpose and they're very very similar to functional tests. They're great for testing logic that spans across multiple controllers.

You can use a generator to create the test skeleton and put it in the right folder.

```
ruby script/generate integration CreaeteNewSnippet
    exists  test/integration/
    create  test/integration/create_new_snippet_test.rb
```

11.1. The flow

Let's take a look at this very simple flow for our application:

1. The user goes to the home page (/)
2. The user submits the form data to the /posts/create action
3. The system displays the new post.
4. The user clicks the link to create another snippet.

We've already tested these things separately, but they're always tested in a vaccuum. Integration tests work as if there's an actual session occurring. We can even follow redirects.

11.2. Use Webrat to test the flow

The Webrat gem makes it easy to implement flows by using the DOM to locate elements so that you can programatically "fill in" the forms and click the links with a script.

Install the *Webrat* gem by opening a command prompt and typing

```
gem install webrat
```

or `sudo gem install webrat` everywhere else.

Open `test/integration/create_new_snippet_test.rb` and add

```
1
```

```
2         require 'webrat'
3
```

right below the line that says

```
1
2         require 'test/test_helper'
3
```

Now add this test case:

```
1
2         def test_should_create_new_snippet
3         visits "/"
4         fills_in "post[title]", :with => "hello world"
5         fills_in "post[code]", :with => "<b>Test</b>"
6         clicks_button "Post it"
7         clicks_link "Create a new post"
8         end
9
10
```

Test your integration test with

```
rake test:integration
```

```
Started
.
Finished in 0.688 seconds.

1 tests, 3 assertions, 0 failures, 0 errors
```

12. Summary

We covered a lot of ground in this chapter. In a very short amount of time, we've taken care of the main piece of the system and we've tested it a couple of ways. There are still a few things we may want to implement, such as a plain-text view of the code, the ability for the original creator to remove or update the code, and also to implement a hashing mechanism so that private URLs can be given out for more sensetive posts. All of this is easy to implement though and can be done using the same test-first methodology.

Index

A

autotest, 10

G

generators, 7

M

migrations
  creating tables with, 7
models
  creating, 7

R

resource, 6

