Rails 4 IN ACTION

Ryan Bigg
Yehuda Katz
Steve Klabnik
Rebecca Skinner

SAMPLE CHAPTER

MANNING

Rails 4 in Action

by Ryan Bigg
Yehuda Katz
Steve Klabnik
and Rebecca Skinner

Chapter 1

brief contents

1 Ruby on Rails, the framework

This chapter covers

- Introducing Ruby on Rails
- Benefits of Rails
- Developing an example Rails application

Welcome aboard! It's great to have you with us on this journey through the world of Ruby on Rails. Ruby on Rails is known as a powerful web framework that helps developers rapidly build modern web applications. In particular, it provides lots of niceties to help you in your quest to develop a full-featured, real-world application, and be happy doing it. Great developers are happy developers.

If you're wondering who uses Rails, there are plenty of companies that do: Twitter, Hulu, and Urban Dictionary, just to name a few. This chapter will teach you how to build a very small and simple application, right after we go through a brief description of what Ruby on Rails actually *is*. Within the first couple of chapters, you'll have the solid foundations for an application, and you'll build on those throughout the rest of the book.

1.1 Ruby on Rails overview

Ruby on Rails is a framework built on the Ruby language—hence the name *Ruby on Rails*. The Ruby language was created back in 1993 by 松本行弘 (Yukihiro "Matz" Matsumoto) of Japan and was released to the general public in 1995. Since then, it has earned both a reputation and an enthusiastic following for its clean design, elegant syntax, and wide selection of tools available in the standard library and via a package management system called *RubyGems*. It also has a worldwide community and many active contributors continuously improving the language and the ecosystem around it. We're not going to go into great depth about the Ruby language in this book though, because we'd rather talk about Ruby on Rails.

RUBY LANGUAGE For a full treatment of the Ruby language, we highly recommend *The Well-Grounded Rubyist* by David A. Black (Manning, 2014).

The foundation for Ruby on Rails was created during 2004 when David Heinemeier Hansson was developing an application called Basecamp. For his next project, the foundational code used for Basecamp was abstracted out into what we know today as Ruby on Rails, released under the MIT License (http://en.wikipedia.org/wiki/MIT_License).

Since then, Ruby on Rails has quickly progressed to become one of the leading web development frameworks. This is in no small part due to the large community surrounding it, contributing everything from documentation to bug fixes to new features for the framework.

This book is written for version 4.2 of the framework, which is the latest version of Rails. If you've used Rails 3.2, you'll find that much feels the same, but Rails has learned some new tricks as well.

RAILS VERSION DIFFERENCES The upgrade guides and release notes provide a great overview of the new features, bug fixes, and other changes in each major and minor version of Rails. They can be found under "Release Notes" on the RailsGuides page: http://guides.rubyonrails.org/.

1.1.1 Benefits

Ruby on Rails allows for the rapid development of applications by using a concept known as *convention over configuration*. A new Ruby on Rails application is created by running the application generator, which creates a standard directory structure and the files that act as a base for every Ruby on Rails application. These files and directories provide categorization for pieces of your code, such as the app/models directory for containing files that interact with the database and the app/assets directory for assets such as stylesheets, JavaScript files, and images. Because all of this is already there, you won't be spending your time configuring the way your application is laid out. It's done for you.

How rapidly can you develop a Ruby on Rails application? Take the annual Rails Rumble event. This event brings together small teams of one to four developers around the world to develop Ruby on Rails[1] applications in a 48-hour period. Using

Rails, these teams deliver amazing web applications in just two days.[2] Another great example of rapid development of a Rails application is the 20-minute blog screencast recorded by Yehuda Katz (http://vimeo.com/10732081). This screencast takes you from having nothing at all to having a basic blogging and commenting system.

Once learned, Ruby on Rails affords you a level of productivity unheard of in other web frameworks, because every Ruby on Rails application starts out the same way. The similarity between the applications is so close that the paradigm shift between different Rails applications isn't tremendous. If and when you jump between Rails applications, you don't have to relearn how it all connects—it's mostly the same. The Rails ecosystem may seem daunting at first, but Rails conventions allow even the new to seem familiar very quickly, smoothing the learning curve substantially.

1.1.2 Ruby gems

The core features of Rails are split up into many different libraries, such as *Active Record*, *Active Support*, *Action Mailer*, and *Action Pack*. These are called *Ruby gems*, or *gems* for short. These gems provide a wide range of methods and classes that help you develop your applications. They eliminate the need for you to perform boring, repetitive tasks—such as coding how your application hooks into your database—and let you get right down to writing valuable code for your business.

GEM VERSIONS The libraries that make up Rails share the same version number as Rails, which means that when you're using Rails 4.2, you're using the 4.2 version of the sub-gems. This is helpful to know when you upgrade Rails, because the version number of the installed gems should be the same as the version number of Rails.

Ever wished for a built-in way of writing automated tests for your web application? Ruby on Rails has you covered with *MiniTest*, which is part of Ruby's standard library. It's incredibly easy to write automated test code for your application, as you'll see throughout this book. Testing your code saves your bacon in the long term, and that's a fantastic thing. We'll touch on MiniTest in the next chapter before moving on to RSpec, which is the testing framework preferred over MiniTest by the majority of the community, and is a little easier on the eyes, too.

In addition to testing frameworks, the Ruby community has produced many high-quality gems for use in your day-to-day development with Ruby on Rails. Some of these libraries add functionality to Ruby on Rails; others provide ways to turn alternative markup languages such as Markdown (see the redcarpet gem at https://rubygems.org/gems/redcarpet) and Textile (see the RedCloth gem at https://rubygems.org/gems/RedCloth) into HTML. Usually, if you can think of it, there's a gem out there that will help you do it.

---

[1] And now other Ruby-based web frameworks, such as Sinatra.

[2] To see an example of what's come out of previous Rails Rumbles, take a look at the alumni archive: http://railsrumble.com/entries/winners.

Noticing a common pattern yet? Probably. As you can see, Ruby on Rails (and the great community surrounding it) provides code that performs the trivial application tasks for you, from setting up the foundations of your application to handling the delivery of email. The time you save with all of these libraries is immense! And because the code is open source, you don't have to go to a specific vendor to get support. Anyone who knows Ruby will help you if you're stuck. Just ask.

1.1.3 Common terms

You'll hear a few common Ruby on Rails terms quite often. This section explains what they mean and how they relate to a Rails application.

MVC

The *model-view-controller* (MVC) paradigm isn't unique to Ruby on Rails, but it provides much of the core foundation for a Ruby on Rails application. This paradigm is designed to keep the logically different parts of the application separate while providing a way for data to flow between them.

In applications that don't use MVC, the directory structure and how the different parts connect to each other are commonly left up to the original developer. Generally, this is a bad idea because different people have different opinions about where things should go. In Rails, a specific directory structure encourages developers to conform to the same layout, putting all the major parts of the application inside an app directory.

This app directory has three main subdirectories: models, controllers, and views:

- *Models* contain the *domain logic* of your application. This logic dictates how the records in your database are retrieved, validated, or manipulated. In Rails applications, models define the code that interacts with the database's tables to retrieve and set information in them. Domain logic also includes things such as validations or particular actions to be performed on the data.
- *Controllers* interact with the models to gather information to send to the view. They're the layer between the user and the database. They call methods on the model classes, which can return single objects representing rows in the database or collections (arrays) of these objects. Controllers then make these objects available to the view through instance variables. Controllers are also used for permission checking, such as ensuring that only users who have special permission to perform certain actions can perform those actions, and users without that permission can't.
- *Views* display the information gathered by the controller, by referencing the instance variables set there, in a developer-friendly manner. In Ruby on Rails, this display is done by default with a templating language known as *Embedded Ruby* (ERB). ERB allows you to embed Ruby into any kind of file you wish. This template is then preprocessed on the server side into the output that's shown to the user.

The assets, helpers, and mailers directories aren't part of the MVC paradigm, but they're also important parts of Rails:

- The *assets* directory is for the static assets of the application, such as JavaScript files, images, and Cascading Style Sheets (CSS), for making the application look pretty. We'll look more closely at this in chapters 3 and 4.
- The *helpers* directory is a place to put Ruby code (specifically, modules) that provide helper methods for just the views. These helper methods can help with complex formatting that would otherwise be messy in the view or is used in more than one place.
- Finally, the *mailers* directory is a home for the classes of your application that deal with sending email. In previous versions of Rails, these classes were grouped with models, but they have since been given their own home. We'll look at them in chapter 12.

REST

MVC in Rails is aided by *Representational State Transfer* (REST; see http://en.wikipedia.org/wiki/Representational_state_transfer for more information). REST is the convention for *routing* in Rails. When something adheres to this convention, it's said to be *RESTful*. Routing in Rails refers to how requests are routed within the application—how URLs map to the controller actions that should process them. You'll benefit greatly by adhering to these conventions, because Rails provides a lot of functionality around RESTful routing, such as determining where a form can submit data.

1.1.4 Rails in the wild

One of the best-known sites that runs Ruby on Rails is GitHub. GitHub is a hosting service for Git repositories. The site was launched in February 2008 and is now the leading Git web-hosting site. GitHub's massive growth was in part due to the Ruby on Rails community quickly adopting it as their de facto repository hosting site. Now GitHub is home to over a million repositories for just about every programming language on the planet. It's not exclusive to programming languages, either; if it can go in a Git repository, it can go on GitHub. As a matter of fact, this book and its source code are kept on GitHub!

You don't have to build huge applications with Rails, either. There's a Rails application that was built for the specific purpose of allowing people to review the previous edition of this book, and it was just over 2,000 lines of code. This application allowed reviewers during the writing of the book to view the book's chapters and leave notes on each element, leading to a better book overall.

Now that you know what other people have accomplished with Ruby on Rails, it's time to dive into creating your own application.

1.2 Developing your first application

We covered the theory behind Rails and showed how quickly and easily you can develop an application. Now it's your turn to get an application going. This will be a simple application that can be used to track items that have been purchased: it will track the name and the price for each item.

First you'll learn how to install Rails and use the scaffold generator that comes with it.

1.2.1 Installing Rails

To get started, you must have these three things installed:

- Ruby
- RubyGems
- Rails

If you're on a UNIX-based system (Linux or Mac), we recommend that you use ruby-install (http://github.com/postmodern/ruby-install) to install Ruby and RubyGems. For Windows, we recommend the RubyInstaller application (http://rubyinstaller.org). There's a complete installation guide for Ruby and Rails on Mac OS X, Linux, and Windows in appendix A.

Before proceeding, let's check that you have everything. Type these commands, and check out the responses:

```
$ ruby -v
ruby 2.2.1p85 (2015-02-26 revision 49769) [x86_64-linux]
$ gem -v
2.4.6
$ rails -v
Rails 4.2.0
```

If you see something that looks close to this, you're good to go! You might see `[x86_64-darwin14]` instead of `[x86_64-linux]`, or a slightly different patch (`p` number), but that's okay. These particular values are the ones we're using right now and we've tested everything in the book against them; as long as you have Ruby 2.1 or later, Rails 4.2 or later, and RubyGems 2.2 or later, everything should be fine.

If you don't get these answers, or you get some sort of error message, please be sure to complete this setup before you try to move on; you can't just ignore errors with this process. Certain gems (and Rails itself) only support particular versions of Ruby, so if you don't get this right, things won't work.

1.2.2 Generating an application

Now that Rails is installed, to generate an application, you run the `rails` command and pass it the `new` argument and the name of the application you want to generate: `things_i_bought`. When you run this command, it creates a new directory called things_i_bought, which is where all your application's code will go.

Don’t use reserved words for application naming

You can call your application almost anything you wish, but it can’t be given a name that’s a reserved word in Ruby or Rails. For example, you wouldn’t call your application *rails*, because the application class would be called `Rails`, and that would clash with the `Rails` constant within the framework. Names like *test* are also forbidden.

When you use an invalid application name, you’ll see an error like one of these:

```
$ rails new rails
Invalid application name rails, constant Rails is already in use.
Please choose another application name.

$ rails new test
Invalid application name test. Please give a name which does not match
one of the reserved rails words.
```

The application you’ll generate will be able to record purchases you’ve made. You can generate it using this command:

```
$ rails new things_i_bought
```

The output from this command may seem a bit overwhelming at first, but rest assured, it’s for your own good. All the directories and files generated provide the building blocks for your application, and you’ll get to know each of them as we progress. For now, you’ll learn by doing, which is the best way. Let’s get rolling.

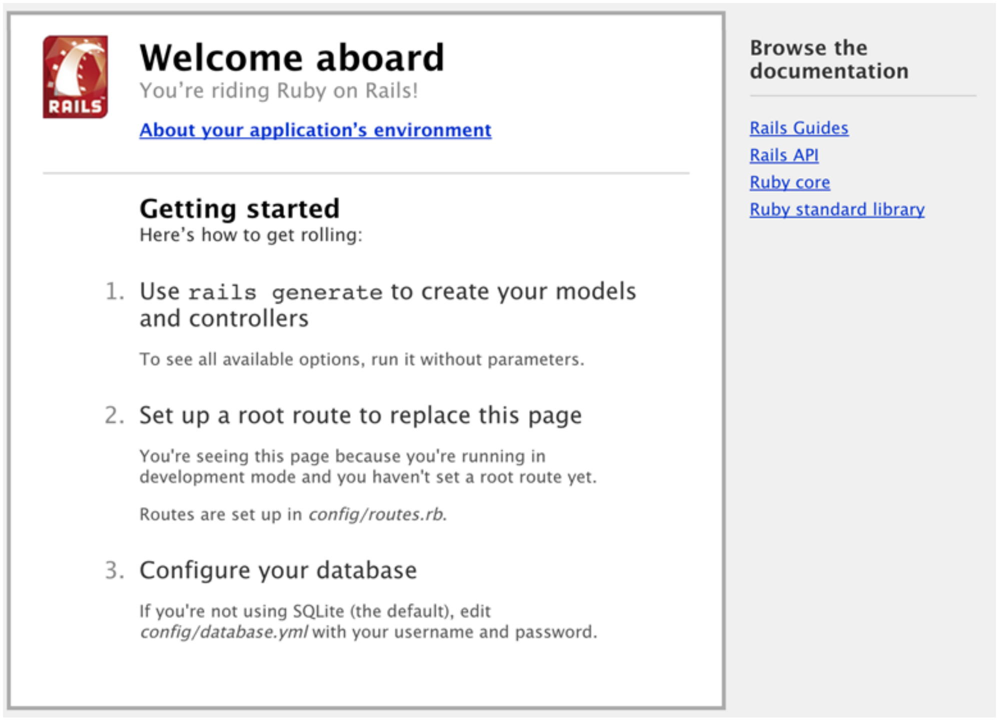
1.2.3 Starting the application

To get the server running, you must first change into the newly created application’s directory and then start the application server:

```
$ cd things_i_bought
$ rails server
```

The `rails server` command (or `rails s`, for short) starts a web server on your local address on port 3000 using a Ruby standard library web server known as WEBrick. It will say “starting in development on http://localhost:3000”, which indicates that the server will be available on port 3000 on the loopback network interface of this machine. To connect to this server, go to http://localhost:3000 in your favorite browser. You’ll see the Welcome Aboard page, which is famous in Rails (see figure 1.1).

On the right side of the Welcome Aboard page are four links to more documentation for Rails and Ruby. The first link takes you to the official Rails Guides page, which will give you great guidance that complements the information in this book. The second link takes you to the Rails API, where you can look up the documentation for classes and methods in Ruby. The final two links take you to documentation about Ruby itself.


Welcome aboard
You're riding Ruby on Rails!
About your application's environment

Getting started
Here's how to get rolling:

1. Use `rails generate` to create your models and controllers

   To see all available options, run it without parameters.

2. Set up a root route to replace this page

   You're seeing this page because you're running in development mode and you haven't set a root route yet.

   Routes are set up in *config/routes.rb*.

3. Configure your database

   If you're not using SQLite (the default), edit *config/database.yml* with your username and password.

Browse the documentation

Rails Guides
Rails API
Ruby core
Ruby standard library

Figure 1.1 Welcome aboard!

If you click the About Your Application's Environment link, you'll find your Ruby, RubyGems, Ruby on Rails, and Rack versions and other environmental data. One of the things to note here is that the output for Environment is "development." Rails provides three environments for running your application: *development*, *test*, and *production*. How your application functions can depend on the environment in which it's running. For example, in the development environment, classes aren't cached, so if you make a change to a class when running an application in development mode, you don't need to restart the server. The same change in the production environment would require a restart.

1.2.4 Scaffolding

To get started with this Rails application, you can generate a *scaffold*. Scaffolds in Rails provide a lot of basic functionality and are generally used as temporary structures for getting started, rather than for full-scale development. Generate a scaffold by running this command:

```
$ rails generate scaffold purchase name:string cost:decimal
```

When you used the `rails` command earlier, it generated an entire Rails application. You can use this command within an application to generate a specific part of the application by passing the generate argument to the rails command, followed by what it is you want to generate. You can also use rails g as a shortcut for rails generate.

The scaffold command generates a model, a controller, views, and tests based on the name passed after scaffold in this command. These are the three important parts needed for your purchase tracking. The model provides a way to interact with a database; the controller interacts with the model to retrieve and format its information and defines different actions to be performed on this data; and the views are rendered by the controller and display the information collected within them.

Everything after the name for the scaffold defines the *fields* for the database table and the *attributes* for the objects of this scaffold. Here you tell Rails that the table for your purchase scaffold will contain name and cost fields, which are a string and a decimal, respectively.[3] To create this table, the scaffold generator generates what's known as a *migration*. Let's look at what migrations are.

1.2.5 Migrations

Migrations are used in Rails as a form of version control for the database, providing a way to implement incremental changes to the database schema. They're usually created along with a model or by running the migration generator. Each migration is timestamped right down to the second, which provides you (and anybody else developing the application with you) an accurate timeline of your database. When two developers are working on separate features of an application and both generate a new migration, this timestamp will stop them from clashing.

Let's open the only file in db/migrate now and see what it does. Its contents are shown in the following listing.

Listing 1.1 db/migrate/[date]_create_purchases.rb

```
class CreatePurchases < ActiveRecord::Migration
  def change
    create_table :purchases do |t|
      t.string :name
      t.decimal :cost

      t.timestamps null: false
    end
  end
end
```

Migrations are Ruby classes that inherit from ActiveRecord::Migration. Inside the class, one method is defined: the change method.

Inside the change method, you use database-agnostic commands to create a table. When this migration is run forward, it will create a table called purchases with a name

[3] Alternatively, you can store the amount in cents as an integer and then do the conversion back to a full dollar amount. For this example, we're using decimal because it's easier to not have to define the conversion. It's worth noting that you shouldn't use a float to store monetary amounts, because it can lead to incorrect rounding errors.

column that's a string, a `cost` column that's a decimal, and two timestamp fields. These timestamp fields are called `created_at` and `updated_at`, and are automatically set to the current time when a record is created or updated, respectively. This feature is built into Active Record. If there are fields present with these names (or `created_on` and `updated_on`), they'll be automatically updated when necessary.

When the migration is reverted, Rails will know how to undo it because it's a simple table creation. The opposite of creating a table is to drop that table from the database. If the migration was more complex than this, you'd need to split it into two methods—one called `up` and one called `down`—that would tell Rails what to do in both cases. Rails is usually smart enough to figure out what you want to do, but sometimes it's not clear and you'll need to be explicit. You'll see examples of this in later chapters.

RUNNING THE MIGRATION

To run the migration, type this command into the console:

```
$ bundle exec rake db:migrate
```

Because this is your first time running migrations in your Rails application, and because you're using a SQLite3 database, Rails first creates the database in a new file at db/development.sqlite3 and then creates the purchases table inside that. When you run `bundle exec rake db:migrate`, it doesn't just run the `change` method from the latest migration, but runs any migration that hasn't yet been run, allowing you to run multiple migrations sequentially.

Your application is, by default, already set up to talk to this new database, so you don't need to change anything. If you ever wanted to roll back this migration, you'd use `bundle exec rake db:rollback`, which rolls back the latest migration by running the `down` method of the migration (or reverses the steps taken in the `change` method, if possible).

ROLLING BACK MULTIPLE MIGRATIONS If you want to roll back more than one migration, use the `bundle exec rake db:rollback STEP=3` command, which rolls back the three most recent migrations.

Rails keeps track of the last migration that was run by storing it using this line in the db/schema.rb file:

```
ActiveRecord::Schema.define(version: [timestamp]) do
```

This version should match the prefix of the migration you just created, where `[timestamp]` in this example is an actual timestamp formatted like `YYYYmmddHHMMSS`. Rails uses this value to know what migration it's up to. The remaining content of this file shows the combined state of all the migrations to this point. This file can be used to restore the last known state of your database if you run the `bundle exec rake db:schema:load` command.

You now have a database set up with a purchases table in it. Let's look at how you can add rows to it through your application.

1.2.6 Viewing and creating purchases

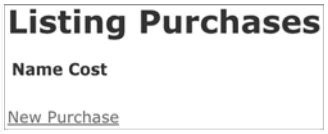


Figure 1.2 **Purchases**

Ensure that your Rails server is still running, or start a new one by running `rails s` or `rails server` again. Start your browser now, and go to http://localhost:3000/purchases. You'll see the scaffolded screen for purchases, as shown in figure 1.2.

No purchases are listed yet, so you can add a new purchase by clicking New Purchase.

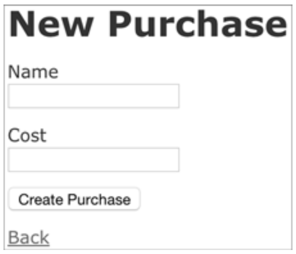


Figure 1.3 **A new purchase**

In figure 1.3, you'll see two inputs for the fields you generated.

This page is the result of rendering the `new` action in the `PurchasesController` controller. What you see on the page comes from the view located at app/views/purchases/new.html.erb, and it looks like the following listing.

Listing 1.2 app/views/purchases/new.html.erb

```
<h1>New Purchase</h1>

<%= render 'form' %>

<%= link_to 'Back', purchases_path %>
```

This is an ERB file, which allows you to mix HTML and Ruby code to generate dynamic pages. The `<%=` beginning of an ERB tag indicates that the result of the code inside the tag will be output to the page. If you want the code to be evaluated but not output, you use the `<%` tag, like this:

```
<% some_variable = "foo" %>
```

If you were to use `<%= some_variable = "foo" %>` here, the `some_variable` variable would be set and the value output to the screen. When you use `<%`, the Ruby code is evaluated but not output.

The `render` method, when passed a string, as in this example, renders a *partial*. A partial is a separate template file that you can include in other templates to repeat similar code. We'll take a closer look at these in chapter 4.

The `link_to` method generates a link with the text of the first argument (`"Back"`) and with an `href` attribute specified by the second argument (`purchases_path`), which is a routing helper that turns into the string /purchases. How this works will be explained a little later when we look at how Rails handles routing.

THE FIRST HALF OF THE FORM PARTIAL

The form partial is at app/views/purchases/_form.html.erb, and the first half of it looks like the following listing.

Listing 1.3 The first half of app/views/purchases/_form.html.erb

```
<%= form_for(@purchase) do |f| %>
  <% if @purchase.errors.any? %>
    <div id="error_explanation">
      <h2><%= pluralize(@purchase.errors.count, "error") %> prohibited
      ➥ this purchase from being saved:</h2>

      <ul>
      <% @purchase.errors.full_messages.each do |message| %>
        <li><%= message %></li>
      <% end %>
      </ul>
    </div>
  <% end %>
...
```

This half is responsible for defining the form by using the `form_for` helper. The `form_for` method is passed one argument—an instance variable called `@purchase`—and with `@purchase` it generates a form. This variable comes from the `new` action of `PurchasesController`, which is shown next.

Listing 1.4 The `new` action of `PurchasesController`

```
def new
  @purchase = Purchase.new
end
```

The first line in this action sets up a new `@purchase` variable by calling the `new` method on the `Purchase` model. This initializes a new instance of the `Purchase` class, but doesn't create a new record in the database. The `@purchase` variable is then automatically passed through to the view by Rails.

So far, all this functionality is provided by Rails. You've coded nothing yourself. With the `scaffold` generator, you get an awful lot for free.

Going back to the app/views/purchases/_form.html.erb partial, the block for the `form_for` is defined between its `do` and the `<% end %>` at the end of the file. Inside this block, you check the `@purchase` object for any errors by using the `@purchase.errors.any?` method. These errors will come from the model if the object doesn't pass the validation requirements set in the model. If any errors exist, they're rendered by the content inside this `if` statement. Validation is a concept covered shortly.

THE SECOND HALF OF THE FORM PARTIAL

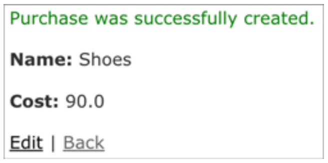
The second half of this partial looks like the following listing.

Listing 1.5 The second half of app/views/purchases/_form.html.erb

```
...
  <div class="field">
    <%= f.label :name %><br>
    <%= f.text_field :name %>
  </div>
  <div class="field">
    <%= f.label :cost %><br>
    <%= f.text_field :cost %>
  </div>
  <div class="actions">
    <%= f.submit %>
  </div>
<% end %>
```

Here, the `f` object from the `form_for` block is used to define labels and fields for your form. At the end of this partial, the `submit` method provides a dynamic Submit button.

Let's fill in this form now and click the Submit button. You should see something similar to figure 1.4. This is the result of your posting: a successful creation of a `Purchase`. Let's see how it got there.



Purchase was successfully created.

Name: Shoes

Cost: 90.0

Edit | Back

Figure 1.4 Your first purchase

The Submit button posts the data from the form to the `create` action, which looks like this.

Listing 1.6 The `create` action of `PurchasesController`

```
def create
  @purchase = Purchase.new(purchase_params)

  respond_to do |format|
    if @purchase.save
      format.html { redirect_to @purchase, notice: 'Purchase was successfully
      created.' }
      format.json { render :show, status: :created, location: @purchase }
    else
      format.html { render :new }
      format.json { render json: @purchase.errors, status:
      ➥ :unprocessable_entity }
    end
  end
end
```

Here, you use the same `Purchase.new` method you first saw in the `new` action. But this time you pass it an argument of `purchase_params`, which is actually another method. That method calls `params` (short for *parameters*), which is a method that returns the parameters sent from your form in a `Hash`-like object. We'll talk more about why you need this little dance later (in chapter 3); this is a feature called *strong parameters*. When you pass this `params` hash into `new`, Rails sets the *attributes* (the Rails word for *fields*) to the values from the form.

Inside `respond_to` is an `if` statement that calls `@purchase.save`. This method *validates* the record; and if it's valid, the method saves the record to the database and returns `true`.

If the return value is `true`, the action responds by redirecting to the new `@purchase` object using the `redirect_to` method, which takes either a path or an object that it turns into a path (as seen in listing 1.6). The `redirect_to` method inspects the `@purchase` object and determines that the path required is `purchase_path` because it's an instance of the `Purchase` model. This path takes you to the `show` action for this controller. The `:notice` option passed to `redirect_to` sets up a *flash message*, which is a message that can be displayed on the next request. This is the green text at the top of figure 1.4.

You've seen what happens when the purchase is valid, but what happens when it's invalid? Well, it uses the `render` method to show the `new` template again. We should note here that this doesn't call the `new` action again, it only renders the template.

REDIRECTING VS. RENDERING To call the `new` action again, you'd call `redirect_to new_purchase_path`, but that wouldn't persist the state of the `@purchase` object to this new request without some seriously bad hackery. By re-rendering the template, you can display information about the object if the object is invalid.

You can make the creation of the `@purchase` object fail by adding a validation. Let's do that now.

1.2.7 Validations

You can add validations to your model to ensure that the data conforms to certain rules, or that data for a certain field must be present, or that a number you enter must be greater than a certain other number. You'll write your first code for this application and implement both of these things now.

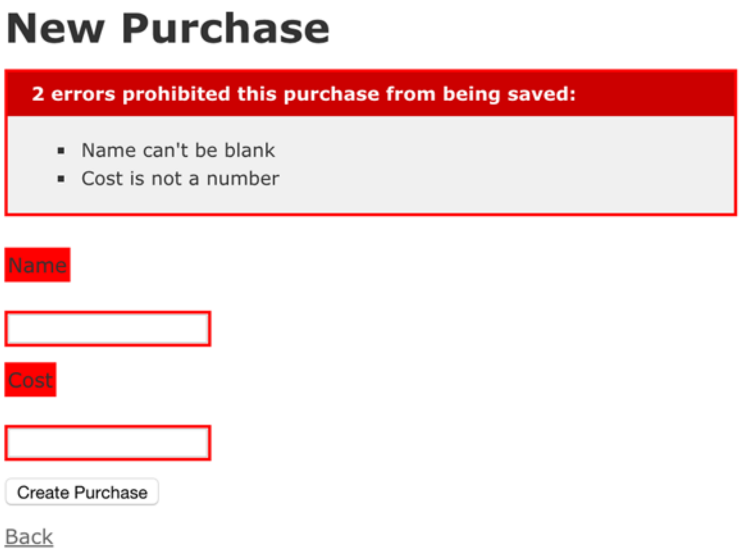
Open your `Purchase` model, and change the entire file to what's shown in the following listing.

Listing 1.7 app/models/purchase.rb

```
class Purchase < ActiveRecord::Base
  validates :name, presence: true
  validates :cost, numericality: { greater_than: 0 }
end
```

You use the `validates` method to define a validation that does what it says on the box: validates that the field is present. The other validation option, `:numericality`, validates that the `cost` attribute is a number and then, with the `:greater_than` option, validates that it's greater than 0.

Let's test these validations by going back to http://localhost:3000/purchases, clicking New Purchase, and clicking Create Purchase. You should see the errors shown in figure 1.5.



New Purchase

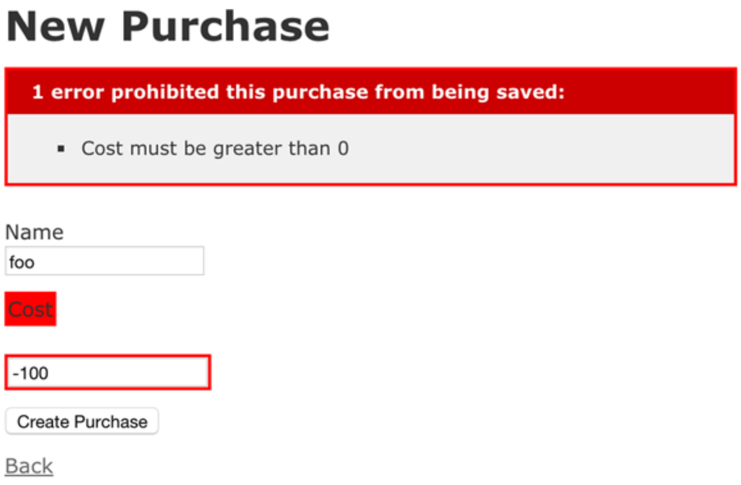
2 errors prohibited this purchase from being saved:

- Name can't be blank
- Cost is not a number

Figure 1.5 Cost must be greater than 0

Great! Here you're told that `name` can't be blank and that the value you entered for `cost` isn't a number. Let's see what happens if you enter `foo` for the Name field and `-100` for the Cost field, and click Create Purchase. You should get a different error for the Cost field now, as shown in figure 1.6.

Good to see! Both of your validations are working. When you change Cost to `100` and click Create Purchase, the value should be considered valid by the validations and take you to the `show` action. Let's look at what this particular action does now.



New Purchase

1 error prohibited this purchase from being saved:

- Cost must be greater than 0

Name: foo

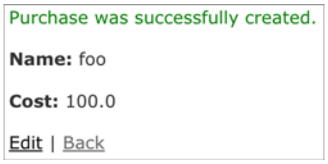
Cost: -100

Figure 1.6 A single purchase

SHOWING OFF

The show action displays the content, as shown in figure 1.7.

The number at the end of the URL, when we're viewing the show action of a project, is the unique numerical ID for this purchase. But what does it mean? Let's look at the view for this show action.


Purchase was successfully created.

Name: foo

Cost: 100.0

Edit | Back

Figure 1.7 **A single purchase**

Listing 1.8 app/views/purchases/show.html.erb

```
<p id="notice"><%= notice %></p>

<p>
  <strong>Name:</strong>
  <%= @purchase.name %>
</p>

<p>
  <strong>Cost:</strong>
  <%= @purchase.cost %>
</p>

<%= link_to 'Edit', edit_purchase_path(@purchase) %> |
<%= link_to 'Back', purchases_path %>
```

On the first line is the notice method, which displays the notice set on the redirect_to from the create action. After that, field values are displayed in p tags by calling them as methods on your @purchase object. This object is defined in the show action of PurchasesController, as shown in the following listing.

Listing 1.9 The show action of PurchasesController

```
def show
end
```

Or is it? It turns out that it's not actually defined here. A before_action is defined.

Listing 1.10 The set_purchase before_action in PurchasesController

```
class PurchasesController < ApplicationController

  before_action :set_purchase, only: [:show, :edit, :update, :destroy]

  ...

  # Use callbacks to share common setup or constraints between actions.
  def set_purchase
    @purchase = Purchase.find(params[:id])
  end

  ...
end
```

This code will be executed before every action given: hence the name before_action. The find method of the Purchase class is used to find the record with the ID of params[:id] and instantiate a new Purchase object from it, with params[:id] being the number on the end of the URL.

Going back to the view (listing 1.8, app/views/purchases/show.html.erb), at the end of this file is link_to, which generates a link using the first argument as the text value, and the second argument as the href for that URL. The second argument for link_to is a method: edit_purchase_path. This method is provided by a method call in config/routes.rb, which we'll look at next.

1.2.8 Routing

The config/routes.rb file of every Rails application is where the application routes are defined in succinct Ruby syntax. The methods used in this file define the pathways from requests to controllers. If you look in your config/routes.rb file, ignoring the commented-out lines for now, you'll see what's shown in the following listing.

Listing 1.11 config/routes.rb

```
Rails.application.routes.draw do
  resources :purchases
end
```

Inside the block for the draw method is the resources method. Collections of similar objects in Rails are referred to as *resources*. This method defines the routes and routing helpers (such as the edit_purchase_path method) to your purchases resources. Look at table 1.1 for a list of the helpers and their corresponding routes. You can see similar output in your terminal if you run the rake routes command inside your things_i_bought directory.

Table 1.1 Routing helpers and their routes

Helper	Route
purchases_path	/purchases
new_purchase_path	/purchases/new
edit_purchase_path	/purchases/:id/edit
purchase_path	/purchases/:id

In this table, :id can be substituted for the ID of a record. Each routing helper has an alternative version that will give you the full URL to the resource. Use the _url extension rather than _path, and you'll get a fully qualified URL such as http://localhost:3000/purchases for purchases_url.

Two of the routes in this table will act differently depending on how they're requested.

The first route, /purchases, takes you to the `index` action of `PurchasesController` if you do a `GET` request. `GET` requests are the standard type of requests for web browsers, and this is the first request you did to this application. If you send a `POST` request to this route, it will go to the `create` action of the controller. This is the case when you submit the form from the `new` view.

The second route that will act differently is /purchases/:id. If you do a `GET` request to this route, it will take you to the `show` action. If you do a `PATCH` request, it will take you to the `update` action. Or you can do a `DELETE` request, which will take you to the `destroy` action.

Let's go to http://localhost:3000/purchases/new now and look at the source of the page. The beginning tag for your form should look like this.

Listing 1.12 HTML source of app/views/purchases/new.html.erb

```
<form accept-charset="UTF-8" action="/purchases"
  class="new_purchase" id="new_purchase" method="post">
```

The two attributes to note here are `action` and `method`. The `action` attribute dictates the URL to where this form goes, and `method` tells the form what kind of HTTP request to make.

How was this tag rendered in the first place? Well, as you saw before, the app/views/purchases/new.html.erb template uses the form partial from app/views/purchases/_form.html.erb, which contains this as the first line:

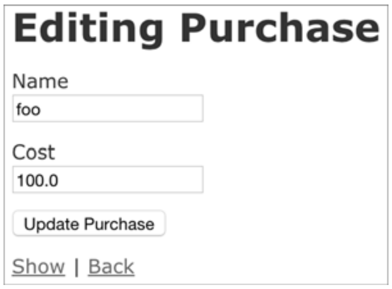
```
<%= form_for(@purchase) do |f| %>
```

This one simple line generates that form tag. When we look at the `edit` action shortly, you'll see that the output of this tag is different, and you'll learn why.

The other route that responds differently is /purchases/:id, which acts in one of three ways. You already saw the first way: it's the `show` action to which you're redirected (via a `GET` request) after you create a purchase. The second of the three ways is when you update a record, which we'll look at now.

1.2.9 Updating

Let's change the cost of the foo purchase now. Perhaps it only cost 10. To change it, go back to http://localhost:3000/purchases and click the "Edit" link next to the foo record. You should see a page that looks similar to the `new` page, as shown in figure 1.8.


Editing Purchase

Name: foo

Cost: 100.0

Figure 1.8 Editing a purchase

This page looks similar because it reuses the app/views/purchases/_form.html.erb partial that was also used in the template for the `new` action. Such is the power of partials in Rails: you can use the same code for two different requests to your application.

The template for this action is shown in the following listing.

Listing 1.13 app/views/purchases/edit.html.erb

```
<h1>Editing Purchase</h1>

<%= render 'form' %>

<%= link_to 'Show', @purchase %> |
<%= link_to 'Back', purchases_path %>
```

For this action, you're working with a preexisting object rather than a new object, which you used in the `new` action. This preexisting object is found by the `edit` action in `PurchasesController`, as shown here.

Listing 1.14 The `edit` action of `PurchasesController`

```
def edit
end
```

Oops: it's not here! The code to find the `@purchase` object is identical to what you saw earlier in the `show` action: it's set in `before_action`, which runs before the `show`, `edit`, `update`, and `destroy` actions.

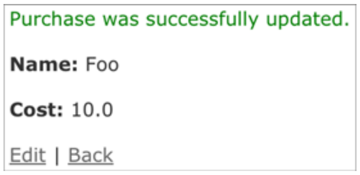
Back in the view for a moment, at the bottom of it you can see two uses of `link_to`. The first creates a "Show" link, linking to the `@purchase` object, which is set up in the `edit` action of your controller. Clicking this link would take you to `purchase_path(@purchase)` or /purchases/:id. Rails will figure out where the link needs to go according to the class of the object given. Using this syntax, it will attempt to call the `purchase_path` method because the object has a class of `Purchase`, and it will pass the object along to that call, generating the URL.

NOTE This syntax is exceptionally handy if you have an object and aren't sure of its type but still want to generate a link for it. For example, if you had a different kind of object called `Order`, and it was used instead, it would use `order_path` rather than `purchase_path`.

The second use of `link_to` in this view generates a "Back" link, which uses the routing helper `purchases_path`. It can't use an object here because it doesn't make sense to. Calling `purchases_path` is the easy way to go back to the `index` action.

Let's try filling in this form—for example, by changing the cost from 100 to 10 and clicking Update Purchase. You'll now see the `show` page but with a different message, as shown in figure 1.9.

Clicking Update Purchase brought you back to the `show` page. How did that happen? Click the Back button on your browser, and view the source of this page, specifically the `form` tag and the tags directly underneath, shown in the following listing.


Purchase was successfully updated.

Name: Foo

Cost: 10.0

Edit | Back

Figure 1.9 Viewing an updated purchase

Listing 1.15 Rendered HTML for app/views/purchases/edit.html.erb

```
...
<form accept-charset="UTF-8" action="/purchases/2" class="edit_purchase"

id="edit_purchase_2" method="post">
  <input name="utf8" type="hidden" value="&#x2713;" />
  <input name="_method" type="hidden" value="patch" />
...
```

The `action` of this `form` points at /purchases/2, which is the route to the `show` action in `PurchasesController`. You should also note two other things. The `method` attribute of this form is a `post`, but there's also the `input` tag underneath.

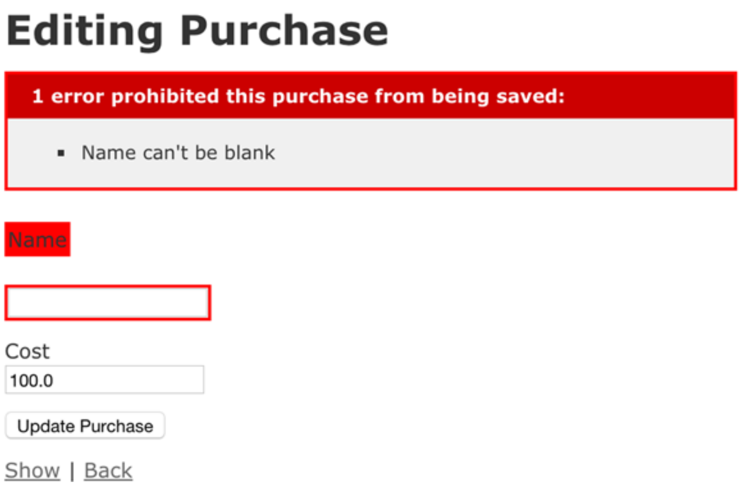
The `input` tag passes through the `_method` parameter with the value set to `patch`. Rails catches this parameter and turns the request from a `POST` into a `PATCH`. This is the second (of three) ways /purchases/:id responds according to the method. By making a `PATCH` request to this route, you're taken to the `update` action in `PurchasesController`. Let's look at this next.

Listing 1.16 The `update` action of `PurchasesController`

```
def update
  respond_to do |format|
    if @purchase.update(purchase_params)
      format.html { redirect_to @purchase, notice: 'Purchase was successfully
      updated.' }
      format.json { render :show, status: :ok, location: @purchase }
    else
      format.html { render :edit }
      format.json { render json: @purchase.errors, status:
      ➥ :unprocessable_entity }
    end
  end
end
```

Just as in the `show` and `edit` actions, the `@purchase` object is first fetched by the call to `before_action :set_purchase`. The parameters from the form are sent through in the same fashion as they were in the `create` action, coming through as `purchase_params`. Rather than instantiating a new object by using the `new` class method, you use `update` on the existing `@purchase` object. This does what it says: updates the attributes. What it doesn't say, though, is that it validates the attributes and, if the attributes are valid, saves the record and returns `true`. If they aren't valid, it returns `false`.

THE `PATCH` METHOD The `PATCH` HTTP method is implemented by Rails by affixing a `_method` parameter on the form with the value of `PATCH`, because the HTML specification doesn't allow the `PATCH` method for form elements. It only allows `GET` and `POST`, as stated here: http://www.w3.org/TR/html401/interact/forms.html#adef-method.


Editing Purchase

1 error prohibited this purchase from being saved:

- Name can't be blank

Figure 1.10 Update fails!

When `update` returns `true`, you're redirected back to the `show` action for this particular purchase by using `redirect_to`. If the `update` call returns `false`, you're shown the `edit` action's template again, just as back in the `create` action where you were shown the `new` template again. This works in the same fashion and displays errors if you enter something wrong.

Let's try editing a purchase, setting Name to blank, and then clicking Update Purchase. It should error exactly like the `create` method did, as shown in figure 1.10.

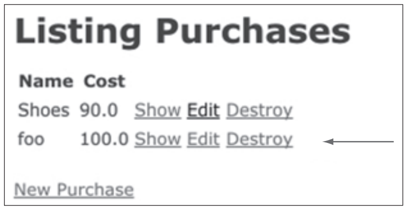
As you can see in this example, the validations you defined in your `Purchase` model take effect automatically for both the creation and updating of records.

What would happen if, rather than updating a purchase, you wanted to delete it? That's built into the scaffold, too.

1.2.10 Deleting

In Rails, *delete* is given a much more forceful name: *destroy*. This is another sensible name, because to destroy a record is to "put an end to the existence of."[4] Once this record's gone, it's gone, baby, gone.

You can destroy a record by going to http://localhost:3000/purchases and clicking the "Destroy" link shown in figure 1.11 and then clicking OK in the confirmation box that pops up.


Listing Purchases

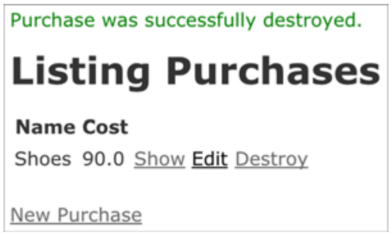
Name	Cost			
Shoes	90.0	Show	Edit	Destroy
foo	100.0	Show	Edit	Destroy

Figure 1.11 Destroy!

[4] As defined by the Mac OS X Dictionary application.

When that record's destroyed, you're taken back to the Listing Purchases page. You'll see that the record no longer exists. You should now have only one record, as shown in figure 1.12.

How does all this work? Let's look at the index template in the following listing to understand, specifically the part that's used to list the purchases.


Purchase was successfully destroyed.

Listing Purchases

Name	Cost			
Shoes	90.0	Show	Edit	Destroy

New Purchase

Figure 1.12 Last record standing

Listing 1.17 app/views/purchases/index.html.erb

```
<% @purchases.each do |purchase| %>
  <tr>
    <td><%= purchase.name %></td>
    <td><%= purchase.cost %></td>
    <td><%= link_to 'Show', purchase %></td>
    <td><%= link_to 'Edit', edit_purchase_path(purchase) %></td>
    <td><%= link_to 'Destroy', purchase, method: :delete, data:

{ confirm: 'Are you sure?' } %></td>
  </tr>
<% end %>
```

In this template, `@purchases` is a collection of all the objects from the `Purchase` model, and `each` is used to iterate over each, setting `purchase` as the variable used in this block.

The methods `name` and `cost` are the same methods used in app/views/purchases/show.html.erb to display the values for the fields. After these, you see the three uses of `link_to`.

The first `link_to` passes in the `purchase` object, which links to the `show` action of `PurchasesController` by using a route such as /purchases/:id, where :id is the ID for this `purchase` object.

The second `link_to` links to the `edit` action using `edit_purchase_path` and passes the `purchase` object as the argument to this method. This routing helper determines that the path is /purchases/:id/edit.

The third `link_to` links seemingly to the `purchase` object exactly like the first, but it doesn't go there. The `:method` option on the end of this route specifies the method `:delete`, which is the third and final way the /purchases/:id route can be used. If you specify `:delete` as the method of this `link_to`, Rails interprets this request as a `DELETE` request and takes you to the `destroy` action in the `PurchasesController`. This action is shown in the following listing.

Listing 1.18 The `destroy` action of `PurchasesController`

```
def destroy
  @purchase.destroy
  respond_to do |format|
    format.html { redirect_to purchases_url, notice: 'Purchase was
    ➥ successfully destroyed.' }
    format.json { head :no_content }
  end
end
```

This action destroys the record loaded by `before_action :set_purchase` by calling `destroy` on it, which permanently deletes the record. Then it uses `redirect_to` to take you to `purchases_url`, which is the route helper defined to take you to http://localhost:3000/purchases. Note that this action uses the `purchases_url` method rather than `purchases_path`, which generates a full URL back to the purchases listing.

That wraps up our application run-through!

1.3 Summary

In this chapter, you learned what Rails is and how to get an application started with it: the absolute bare, bare, *bare* essentials of a Rails application. But look how fast you got going! It took only a few simple commands and an entire two lines of your own code to create the bones of a Rails application. From this basic skeleton, you can keep adding bits and pieces to develop your application, and all the while you get things for free from Rails. You don't have to code the logic of what happens when Rails receives a request or specify what query to execute on your database to insert a record—Rails does it for you.

You also saw that some big-name players—such as Twitter and GitHub—use Ruby on Rails. This clearly answers the question "Is Rails ready?" Yes, it very much is. A wide range of companies have built successful websites on the Rails framework, and many more will do so in the future. Rails also has been around for a decade, and shows no signs of slowing down any time soon.

Still wondering if Ruby on Rails is right for you? Ask around. You'll hear a lot of people singing its praises. The Ruby on Rails community is passionate not only about Rails but also about community building. Events, conferences, user group meetings, and even camps are held around the world for Rails. Attend these, and discuss Ruby on Rails with the people who know about it. If you can't attend these events, you can explore the IRC channel on Freenode *#rubyonrails* and the mailing list *rubyonrails-talk* on Google Groups, not to mention Stack Overflow and a multitude of other areas on the internet where you can find experienced people and discuss what they think of Rails. Don't let this book be your only source of knowledge. There's a whole world out there, and no book could cover it all!

The best way to answer the question "What is Rails?" is to experience it for yourself. This book and your own exploration can eventually make you a Ruby on Rails expert.

When you added validations to your application earlier, you manually tested that they were working. This may seem like a good idea for now, but when the application grows beyond a couple of pages, it becomes cumbersome to manually test it. Wouldn't it be nice to have some automated way of testing your applications? Something to ensure that all the individual parts always work? Something to provide the peace of mind that you crave when you develop anything? You want to be sure that your application is continuously working with the least effort possible, right?

Well, Ruby on Rails does that too. Several testing frameworks are available for Ruby and Ruby on Rails, and in chapter 2 we'll look at the two major ones: MiniTest and RSpec.

WEB DEVELOPMENT/RUBY

Rails 4 IN ACTION

Bigg • Katz • Klabnik • Skinner

Rails is a full-stack, open source web framework powered by Ruby. Now in version 4, Rails is mature and powerful, and to use it effectively you need more than a few Google searches. You'll find no substitute for the guru's-eye-view of design, testing, deployment, and other real-world concerns that this book provides.

Rails 4 in Action is a hands-on guide to the subject. In this fully revised new edition, you'll master Rails 4 by developing a ticket-tracking application that includes RESTful routing, authentication and authorization, file uploads, email, and more. Learn to design your own APIs and successfully deploy a production-quality application. You'll see test-driven development and behavior-driven development in action throughout the book, just like in a top Rails shop.

What's Inside

- Creating your own APIs
- Using RSpec and Capybara
- Emphasis on test-first development
- Fully updated for Rails 4

For readers of this book, a background in Ruby is helpful but not required. No Rails experience is assumed.

Ryan Bigg, **Yehuda Katz**, **Steve Klabnik**, and **Rebecca Skinner** are contributors to Rails and active members of the Rails community.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/rails-4-in-action

Free eBook
SEE INSERT

"There's no better source for Rails 4. This book blows away the competition."
—Damien White, Visoft, Inc.

"A gentle yet thorough guide to Rails 4."
—William Wheeler
ProData Computer Services

"Very clear, with excellent examples. A must-read for everyone in the Rails world."
—Michele Bursi, Nokia

"Well-written, intuitive, and easy to understand."
—Lee Allen
SecuritySession.com

ISBN 13: 978-1-617291-09-8
ISBN 10: 1-617291-09-9
54999
9 781617 291098

MANNING $49.99 / Can $57.99 [INCLUDING eBOOK]