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 2

brief contents

2 Testing saves your bacon

This chapter covers

- Introducing testing approaches
- Test-driven development with MiniTest
- Behavior-driven development with RSpec

Chapter 1 presented an extremely basic layout of a Rails application and an example of using the scaffold generator. One question remains, though: how do you make your Rails applications maintainable?

> **ABOUT THE SCAFFOLD GENERATOR** We won't use the scaffold generator for the rest of the book because people tend to use it as a crutch, and it generates extraneous code. There's a thread on the rubyonrails-core mailing list where people have discussed the scaffold generator's downsides: http://mng.bz/g33u.

The answer is that you write automated tests for the application as you develop it, and you write these all the time. By writing automated tests for your application, you can quickly ensure that your application is working as intended. If you don't write tests, your alternative is to check the entire application manually every time you make a change, which is time consuming and error prone. Automated testing saves you a ton of time in the long run and leads to fewer bugs. Humans make mistakes; programs (if coded correctly) don't. We'll do it correctly from step one.[1]

[1] Unlike certain other books.

In the Ruby world, a huge emphasis is placed on testing, specifically on *test-driven development* (TDD) and *behavior-driven development* (BDD). This chapter covers two testing tools—MiniTest and RSpec—in a basic fashion so you can quickly learn their formats.

By learning good testing techniques now, you'll have a solid way to make sure nothing is broken when you start to write your first real Rails application. If you don't write tests, there'll be no automatic way of telling what might be going wrong in your code.

A cryptic yet true answer to the question "Why should I test?" is "Because you're human." Humans—the large majority of this book's audience—make mistakes. It's one of our favorite ways to learn. Because humans make mistakes, having a tool to inform us when we make one is helpful, isn't it? Automated testing provides a quick safety net to inform developers when they make mistakes. And by *they*, of course, we mean *you*. We want you to make as few mistakes as possible. We want you to save your bacon!

2.1 Using TDD and BDD to save your bacon

In addition to catching errors, TDD and BDD give you time to think through your decisions before you write any code. By first writing a test for the implementation, you are (or, at least, you should be) thinking through the implementation: the code you'll write *after* the test and how you'll make the test pass. If you find the test difficult to write, then perhaps the implementation could be improved. Unfortunately, there's no clear way to quantify the difficulty of writing a test and working through it, other than to consult with other people who are familiar with the process.

Once the test is implemented, you should go about writing some code that your test can pass. If you find yourself working backward—rewriting your test to fit a buggy implementation—it's generally best to rethink the test and scrap the implementation. Test first, code later.

TDD is a methodology consisting of writing a failing test case first (usually using a testing tool such as MiniTest), then writing the code to make the test pass, and finally refactoring the code to make it neater and tidier. This process is commonly called *red-green-refactor*. The reasons for developing code this way are twofold. First, it makes you consider how the code should be running before it's used by anybody. Second, it gives you an automated test you can run as often as you like to ensure that your code is still working as you intended. This book uses the MiniTest tool for TDD.

BDD is a methodology based on TDD. You write an automated test to check the interaction between the different parts of the codebase rather than to test that each part works independently. Two tools used for BDD when building Rails applications are RSpec and Cucumber. This book relies heavily on RSpec and forgoes Cucumber.

CUCUMBER VS. OTHER TOOLS Cucumber was used in earlier editions of this book, but the community has drifted away from using it, as there are other tools (like Capybara, mentioned later) that provide a very similar way to test, but in a much neater, pure-Ruby syntax.

Let's begin by looking at TDD and MiniTest.

2.2 Test-driven development basics

Automated testing is much, much easier than manual testing. Have you ever gone through a website and manually filled in a form with specific values to make sure it conforms to your expectations? Wouldn't it be faster and easier to have the computer do this work? Yes, it would, and that's the beauty of automated testing: you won't spend your time manually testing your code, because you'll have written test code to do that for you.

On the off chance that you break something, the tests are there to tell you the what, when, how, and why of the breakage. Although tests can never be 100% guaranteed, your chances of getting this information without first having written tests are 0%. Nothing is worse than finding out through an early morning phone call from an angry customer that something is broken. Tests help prevent such scenarios by giving you and your client peace of mind. If the tests aren't broken, chances are high (although not guaranteed) that the implementation isn't either.

Sooner or later, it's likely that something in your application will break when a user attempts to perform an action you didn't consider in your tests. With a base of tests, you can easily duplicate the scenario in which the user encountered the breakage, generate your own failed test, and use this information to fix the bug. This commonly used practice is called *regression testing*.

It's valuable to have a solid base of tests in the application so you can spend time developing new features *properly*, rather than fixing the old ones you didn't do quite right. An application without tests is most likely broken in one way or another.

2.2.1 Writing your first test

The first testing library for Ruby was Test::Unit, which was written by Nathaniel Talbott back in 2000 and is now part of the Ruby standard library. The documentation for this library gives a fantastic overview of its purpose, as summarized by the man himself:

> *The general idea behind unit testing is that you write a test method that makes certain assertions about your code, working against a test fixture. A bunch of these test methods are bundled up into a test suite and can be run any time the developer wants. The results of a run are gathered in a test result and displayed to the user through some UI.*
>
> —Nathaniel Talbott

The UI Talbott references could be a terminal, a web page, or even a light.[2]

In Rails 4, Test::Unit has been superseded by MiniTest, which is a library of a similar style but with a more modern heritage. MiniTest is also part of the Ruby standard library.

A common practice you'll hopefully have experienced by now in the Ruby world is to let the libraries do a lot of the hard work for you. Sure, you *could* write a file yourself that loads one of your other files and runs a method and makes sure it works, but why

[2] Such as the one GitHub has made: http://github.com/blog/653-our-new-build-status-indicator.

do that when MiniTest already provides that functionality for such little cost? Never reinvent the wheel when somebody's done it for you.

Now you'll write a test, and you'll write the code for it later. Welcome to TDD.

TRYING OUT MINITEST

To try out MiniTest, first create a new directory called chapter_2, and in that directory make a file called example_test.rb. It's good practice to suffix your filenames with *_test* so it's obvious from the filename that it's a test file. In this file, you'll define the most basic test possible, as shown in the following listing.

Listing 2.1 chapter_2/example_test.rb

```
require "minitest/autorun"

class ExampleTest < Minitest::Test
  def test_truth
    assert true
  end
end
```

To make this a MiniTest test, you begin by requiring minitest/autorun, which is part of Ruby's standard library. This provides the `Minitest::Test` class inherited from on the next line. Inheriting from this class provides the functionality to run any method defined in this class whose name begins with `test`.

To run this file, you run `ruby example_test.rb` in the terminal, from inside the chapter_2 directory. When this code completes, you'll see some output, the most relevant being the last three lines:

```
.

Finished in 0.001245s, 803.2129 runs/s, 803.2129 assertions/s.

1 runs, 1 assertions, 0 failures, 0 errors, 0 skips
```

The first line is a singular period. This is MiniTest's way of indicating that it ran a test and the test passed. If the test had failed, it would show up as an `F`; if it had errored, an `E`. The second and third lines provide statistics on what happened—specifically that there was one test and one assertion, and that nothing failed, there were no errors, and nothing was skipped. Great success!

The `assert` method in your test makes an assertion that the argument passed to it evaluates to `true`. This test passes given anything that's not `nil` or `false`. When this method fails, it fails the test and raises an exception. Go ahead and try putting `1` there instead of `true`. It still works:

```
Finished tests in 0.001071s, 933.7068 tests/s, 933.7068 assertions/s.

1 runs, 1 assertions, 0 failures, 0 errors, 0 skips
```

In the following listing, you remove the `test_` from the beginning of your method and define it as a `truth` method.

Listing 2.2 chapter_2/example_test.rb, alternate truth test

```
def truth
  assert true
end
```

When you run the test again with `ruby example_test.rb`, MiniTest tells you there were no tests specified:

```
0 runs, 0 assertions, 0 failures, 0 errors, 0 skips
```

See, no tests! Remember to always prefix MiniTest methods with `test`!

2.2.2 Saving bacon

Let's make this a little more complex by creating a bacon_test.rb file in the same folder and writing the test shown next.

Listing 2.3 chapter_2/bacon_test.rb

```
require "minitest/autorun"

class BaconTest < Minitest::Test
  def test_saved
    assert Bacon.saved?
  end
end
```

Of course, you want to ensure that your bacon (both the metaphorical and the crispy kinds) is always saved, and this is how you do it. If you now run the code to run this file, `ruby bacon_test.rb`, you'll get an error:

```
1) Error:
BaconTest#test_saved:
NameError: uninitialized constant BaconTest::Bacon
    bacon_test.rb:5:in `test_saved'
```

Your test is looking for a constant called `Bacon` when you call `Bacon.saved?`, and it can't find it because you haven't yet defined the constant.

For this test, the constant you want to define is a `Bacon` class, and you can define this class before or after the test. Note that in Ruby you usually must define constants and variables before you use them, but in MiniTest tests, the code is only run when MiniTest finishes evaluating it, which means you can define the `Bacon` class after the test. In the next listing, you follow the more conventional method of defining the class above the test.

Listing 2.4 chapter_2/bacon_test.rb, now with `Bacon` class

```
require "minitest/autorun"

class Bacon
end
```

```
class BaconTest < Minitest::Test
  def test_saved
    assert Bacon.saved?
  end
end
```

Upon rerunning the test, you get a different error:

```
1) Error:
BaconTest#test_saved:
NoMethodError: undefined method `saved?' for Bacon:Class
    bacon_test.rb:8:in `test_saved'
```

Progress! It recognizes there's now a `Bacon` class. But there's no `saved?` method for this class, so you must define one.

Listing 2.5 `Bacon` class in chapter_2/bacon_test.rb

```
class Bacon
  def self.saved?
    true
  end
end
```

One more run of `ruby bacon_test.rb`, and you can see that the test is now passing:

```
.

Finished tests in 0.000596s, 1677.8523 tests/s, 1677.8523 assertions/s.

1 runs, 1 assertions, 0 failures, 0 errors, 0 skips
```

Your bacon is indeed saved! Now any time you want to check whether it's saved, you can run this file. If somebody else comes along and changes that `true` value to a `false`, the test will fail:

```
F

Finished in 0.001037s, 964.3825 runs/s, 964.3825 assertions/s.

  1) Failure:
BaconTest#test_saved [bacon_test.rb:11]:
Failed assertion, no message given.
```

MiniTest reports "Failed assertion, no message given" when an assertion fails. You should probably make that error message clearer! To do so, you can specify an additional argument to the `assert` method in your test, like this:

```
def test_saved
  assert Bacon.saved?, "Our bacon was not saved :("
end
```

Now when you run the test, you get a clearer error message:

```
1) Failure:
BaconTest#test_saved [bacon_test.rb:11]:
Our bacon was not saved :(
```

And that, our friend, is the basics of TDD using MiniTest. Although we won't use this method in the book, it's handy to know about, because it establishes the basis for TDD in Ruby, in case you wish to use it in the future. MiniTest is also the default testing framework for Rails, so you may see it around in your travels.

From this point on, we'll focus on pure RSpec, which you'll use to develop your next Rails application.

2.3 Behavior-driven development basics

BDD is similar to TDD, but the tests for BDD are written in an easier-to-understand language so that developers and clients alike can clearly understand what's being tested. The tool you'll use for all BDD examples in this book is RSpec.

RSpec tests are written in a Ruby domain-specific language (DSL), like this:

```
RSpec.describe Bacon do
  it "is edible" do
    expect(Bacon).to be_edible
  end
end
```

The benefits of writing tests like this are that clients can understand precisely what the test is testing and then use these steps in acceptance testing; a developer can read what the feature should do and then implement it; and finally, the test can be run as an automated test. With tests written in a DSL, you have the three important elements of your business (the clients, the developers, and the code) all operating in the same language.

ACCEPTANCE TESTING *Acceptance testing* is a process whereby people follow a set of instructions to ensure that a feature is performing as intended.

RSpec is an extension of the methods already provided by MiniTest. You can even use MiniTest methods in RSpec tests if you wish. But we'll use the simpler, easier-to-understand syntax that RSpec provides.

2.3.1 Introducing RSpec

RSpec is a BDD tool written by Steven R. Baker and now maintained by Myron Marston and Andy Lindeman as a cleaner alternative to MiniTest. With RSpec, you write code known as *specs* that contain *examples*, which are synonymous with the *tests* you know from MiniTest. In this example, you'll define the `Bacon` constant and then define the `edible?` method on it.

Let's jump right in and install RSpec. The latest version of the gem (at writing) is 3.2.0, and you can install it by running `gem install rspec -v 3.2.0`. You should see something like the following output:

```
Fetching: diff-lcs-1.2.5.gem (100%)
Successfully installed diff-lcs-1.2.5
Fetching: rspec-support-3.2.2.gem (100%)
```

```
Successfully installed rspec-support-3.2.2
Fetching: rspec-mocks-3.2.1.gem (100%)
Successfully installed rspec-mocks-3.2.1
Fetching: rspec-expectations-3.2.0.gem (100%)
Successfully installed rspec-expectations-3.2.0
Fetching: rspec-core-3.2.2.gem (100%)
Successfully installed rspec-core-3.2.2
Fetching: rspec-3.2.0.gem (100%)
Successfully installed rspec-3.2.0
6 gems installed
```

You can see that the final line says the rspec gem is installed, with the version number specified after the name.

2.3.2 Writing your first spec

When the gem is installed, you can create a new directory called *bacon* for your tests anywhere you like; in that directory, create another directory called *spec*. If you're running a UNIX-based operating system such as Linux or Mac OS X, you can run the command `mkdir -p bacon/spec` to create these two directories. This code will generate a bacon directory, if it doesn't already exist, and then generate a spec directory inside it.

In the spec directory, create a file called bacon_spec.rb. This is the file you'll use to test your currently nonexistent `Bacon` class. Put the code from the following listing in spec/bacon_spec.rb.

Listing 2.6 bacon/spec/bacon_spec.rb

```
RSpec.describe Bacon do
  it "is edible" do
    expect(Bacon.edible?).to be(true)
  end
end
```

You use `RSpec.describe` to describe the behavior of the (currently undefined) `Bacon` class and write an example for it, declaring that `Bacon` is `edible`. The `describe` block contains tests (examples) that describe the behavior of bacon. In this example, whenever you call `edible?` on `Bacon`, the result should be `true`. `expect` and `to` serve a purpose similar to that of `assert`, which is to assert that the object passed to `expect` matches the arguments passed to `to`. If the outcome isn't what you say it should be, then RSpec raises an error and goes no further with that spec.

THERE'S MORE THAN ONE WAY TO WRITE A SPEC

An alternative way to write the spec would be like in the following listing.

Listing 2.7 An alternate way to check if Bacon is edible

```
RSpec.describe Bacon do
  it "is edible" do
    expect(Bacon).to be_edible
  end
end
```

RSpec will internally translate the `be_edible` method call into `edible?`, and call that on `Bacon`. If the overall result of the `Bacon.edible?` statement is *truthy* (anything other than nil or false), then the spec will pass. But for now, we'll stick with the first version—it's a little less magical, and it's easier to see what's going on.

2.3.3 Running the spec

To run the spec, you run `rspec spec` in a terminal inside your bacon directory. You specify the spec directory as the main argument to the `rspec` executable so RSpec will run all the tests in that directory. This code can also take files as its arguments if you want to run tests only from those files.

When you run this spec, you'll get an `uninitialized constant Bacon (NameError)` error, because you haven't yet defined your `Bacon` constant. To define it, create another directory in your Bacon project folder called *lib*, and in this directory, create a file called *bacon.rb*. This is the file where you define the `Bacon` constant, a class.

Listing 2.8 bacon/lib/bacon.rb

```
class Bacon
end
```

You can now require this file in spec/bacon_spec.rb by placing the following line at the top of the file:

```
require "bacon"
```

When you run your spec again, because you told it to load `bacon`, RSpec will have added the lib directory to Ruby's load path on the same level as the spec directory, so it will find lib/bacon.rb for your `require`. By requiring the lib/bacon.rb file, you ensure that the `Bacon` constant is defined. The next time you run the spec, you'll get an undefined method for your new constant:

```
1) Bacon is edible
   Failure/Error: expect(Bacon.new.edible?).to be(true)
   NoMethodError:
     undefined method `edible?' for #<Bacon:0x007f2530184988>
   # ./spec/bacon_spec.rb:5:in `block (2 levels) in <top (required)>'
```

This means you need to define the `edible?` method on your `Bacon` class. Reopen lib/bacon.rb, and add this method definition to the class:

```
def self.edible?
  true
end
```

Now the entire file looks like the following listing.

Listing 2.9 bacon/lib/bacon.rb

```
class Bacon
  def self.edible?
```

```
    true
  end
end
```

By defining the method as `self.edible?`, you define it for the class. If you didn't prefix the method with `self.`, it would define the method for an instance of the class rather than for the class itself.

Running `rspec spec` now outputs a period, which indicates the test has passed. That's the first test—done.

2.3.4 Much more bacon

For the next test, you want to create many instances of the `Bacon` class and have the `edible?` method defined on them. To do this, open lib/bacon.rb and change the `edible?` class method to an instance method by removing the `self.` from before the method, as shown next.

Listing 2.10 bacon/lib/bacon.rb

```
class Bacon
  def edible?
    true
  end
end
```

When you run `rspec spec` again, you'll get the familiar error:

```
1) Bacon is edible
   Failure/Error: expect(Bacon.edible?).to be(true)
   NoMethodError:
     undefined method `edible?' for Bacon:Class
   # ./spec/bacon_spec.rb:5:in `block (2 levels) in <top (required)>'
```

Oops! You broke a test! You should be changing the spec to suit your new ideas before changing the code! Let's reverse the changes made in lib/bacon.rb.

Listing 2.11 bacon/lib/bacon.rb

```
class Bacon
  def self.edible?
    true
  end
end
```

When you run `rspec spec` again, it passes. Now let's change the spec first.

Listing 2.12 bacon/spec/bacon_spec.rb

```
RSpec.describe Bacon do
  it "is edible" do
    expect(Bacon.new.edible?).to be(true)
  end
end
```

In this code, you instantiate a new object of the class rather than use the `Bacon` class. When you run `rspec spec`, it breaks once again:

```
NoMethodError:
  undefined method `edible?' for #<Bacon:0x101deff38>
```

If you remove the `self.` from the `edible?` method, your test will now pass:

```
.

Finished in 0.00167 seconds
1 example, 0 failures
```

2.3.5 Expiring bacon

You can go about breaking your test once more by adding functionality: an `expired!` method, which will make your bacon inedible. This method sets an instance variable on the `Bacon` object called `@expired` to `true`, and you can use it in your `edible?` method to check the bacon's status.

First you must test that this `expired!` method will do what you think it should do. Create another example in spec/bacon_spec.rb so that the whole file looks like the following listing.

Listing 2.13 bacon/spec/bacon_spec.rb

```
require "bacon"

RSpec.describe Bacon do
  it "is edible" do
    expect(Bacon.new.edible?).to be(true)
  end

  it "can expire" do
    bacon = Bacon.new
    bacon.expired!
    expect(bacon).to_not be_edible
  end
end
```

This uses the second format of the assertion—RSpec again translates `be_edible` to `edible?` and calls `bacon.edible?`. But this time it's expected to return something *falsey* (either `nil` or `false`), due to the negative `to_not` (instead of `to`).

If you run `rspec` again, your first spec still passes, but your second one fails because you have yet to define your `expired!` method. Let's do that now in lib/bacon.rb.

Listing 2.14 bacon/lib/bacon.rb

```
class Bacon
  def edible?
    true
  end
  def expired!
    self.expired = true
  end
end
```

By running `rspec spec` again, you get an `undefined method` error:

```
1) Bacon can expire
   Failure/Error: bacon.expired!
   NoMethodError:
     undefined method `expired=' for #<Bacon:0x007ff116460c58>
   # ./lib/bacon.rb:7:in `expired!'
```

This method is called by this line in the previous listing:

```
self.expired = true
```

To define this method, you can use the `attr_accessor` method provided by Ruby, as shown in listing 2.15; the `attr` prefix of the method means *attribute*. If you pass a `Symbol` (or collection of symbols) to this method, it defines methods for setting (`expired=`) and retrieving the attribute's `expired` values, referred to as a *setter* and a *getter*, respectively. It also defines an instance variable called `@expired` on every object of this class to store the value that was specified by the `expired=` method calls.

> **THE `SELF.` METHOD PREFIX** In Ruby you can call methods without the `self.` prefix. In this case, though, when calling the `expired=` method, you need to specify the prefix or the interpreter will think that you're defining a local variable called `expired`, rather than calling the method. For setter methods, you should always use the prefix.

Listing 2.15 `attr_accessor` method for `Bacon` in bacon/lib/bacon.rb

```
class Bacon
  attr_accessor :expired
  ...
end
```

With this in place, if you run `rspec spec` again, your example fails on the line following your previous failure:

```
1) Bacon can expire
   Failure/Error: expect(bacon).to_not be_edible
     expected `#<Bacon:0x007f0fa5f56cc8 @expired=true>.edible?` to
     return false, got true
   # ./spec/bacon_spec.rb:11:in `block (2 levels) in <top (required)>'
```

Even though this sets the `expired` attribute on the `Bacon` object, you've still hard-coded `true` in your `edible?` method. Now change the method to use the attribute method, as in the following listing.

Listing 2.16 `Bacon#edible?` method

```
def edible?
  !expired
end
```

When you run `rspec spec` again, both your specs will pass:

```
..

Finished in 0.00191 seconds
2 examples, 0 failures
```

Let's go back into lib/bacon.rb and remove the `self.` from the `expired!` method, just to see what happens:

```
def expired!
  expired = true
end
```

If you run `rspec spec` again, you'll see that your second spec is now broken:

```
1) Bacon can expire
   Failure/Error: expect(bacon).to_not be_edible
     expected `#<Bacon:0x007fbc555d0930>.edible?` to return false,
     got true
   # ./spec/bacon_spec.rb:11:in `block (2 levels) in <top (required)>'
```

You can see that your `Bacon` instance (`#<Bacon:0x007fbc555d0930>`) no longer has an `@expired` attribute set to `true`, like you had in the previous failure, because you're not calling the `expired=` method anymore.

Tests save you from making mistakes such as this. If you write the test first and then write the code to make the test pass, you have a solid base and can refactor the code to be clearer or smaller, and finally you can ensure that it's still working with the test you wrote in the first place. If the test still passes, then you're probably doing it right.

If you change this method back now,

```
def expired!
  self.expired = true
end
```

and then run your specs using `rspec spec`, you'll see that they once again pass:

```
..

2 examples, 0 failures
```

Everything's normal and working, which is great!

That ends our little foray into RSpec for now. You'll use it again later when you develop your application. If you'd like to know more about RSpec, Noel Rappin's *Rails 4 Test Prescriptions* (https://pragprog.com/book/nrtest2/rails-4-test-prescriptions) is recommended reading.

2.4 Summary

This chapter demonstrated how to apply TDD and BDD principles to test some rudimentary code. You can (and should!) apply these principles to all the code you write, because testing the code ensures that it's maintainable from now into the future. You don't have to use the gems shown in this chapter to test your Rails application; they're just preferred by a large portion of the community.

You'll apply what you learned in this chapter to build a Rails application from scratch in upcoming chapters. You'll use RSpec and another tool called Capybara to build out acceptance tests that will describe the behavior of your application. Then you'll implement the behavior of the application to make these tests pass, and you'll know you're doing it right when the tests are all green.

Let's get into it!

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]