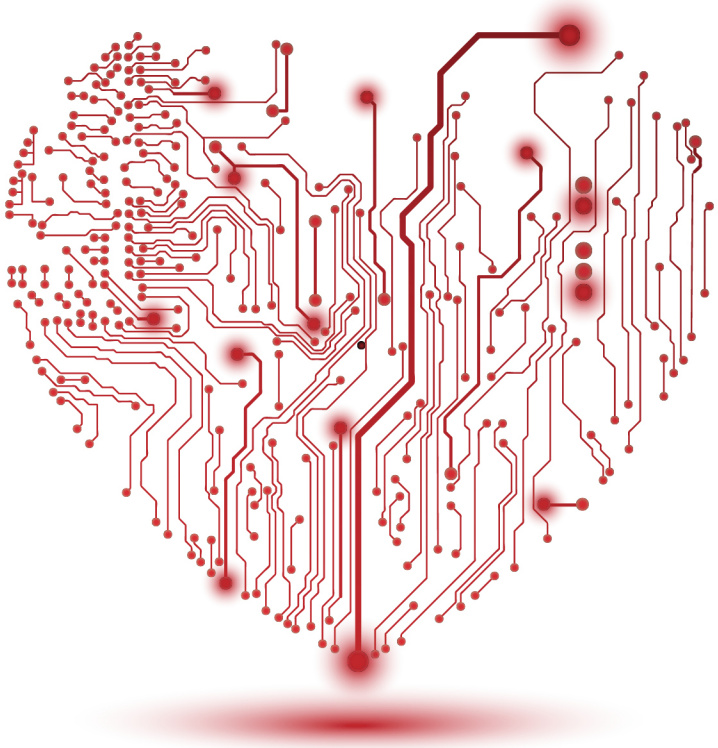
The Pragmatic Programmers

Rails 4 Test Prescriptions

Build a Healthy Codebase



Noel Rappin

Edited by Lynn Beighley

Under Construction: The book you're reading is still under development. As part of our Beta book program, we're releasing this copy well before a normal book would be released. That way you're able to get this content a couple of months before it's available in finished form, and we'll get feedback to make the book even better. The idea is that everyone wins!

Be warned: The book has not had a full technical edit, so it will contain errors. It has not been copyedited, so it will be full of typos, spelling mistakes, and the occasional creative piece of grammar. And there's been no effort spent doing layout, so you'll find bad page breaks, over-long code lines, incorrect hyphenation, and all the other ugly things that you wouldn't expect to see in a finished book. It also doesn't have an index. We can't be held liable if you use this book to try to create a spiffy application and you somehow end up with a strangely shaped farm implement instead. Despite all this, we think you'll enjoy it!

Download Updates: Throughout this process you'll be able to get updated ebooks from your account at *pragprog.com/my_account*. When the book is complete, you'll get the final version (and subsequent updates) from the same address.

Send us your feedback: In the meantime, we'd appreciate you sending us your feedback on this book at *pragprog.com/titles/nrtest2/errata*, or by using the links at the bottom of each page.

Thank you for being part of the Pragmatic community!

Dave & Andy

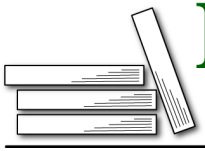
Rails 4 Test Prescriptions

Build a Healthy Codebase

Noel Rappin

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-941222-19-5
Encoded using the finest acid-free high-entropy binary digits.
Book version: B2.0—July 16, 2014

Contents

1. **Changes in the Beta Releases** 1
   Beta 2.0, July 16, 2014 1

2. **A Testing Fable** 3
   Who You Are 5
   The Power of Testing First 5
   What Is TDD Good For? 7
   When TDD Needs Some Help 8
   Words to Live By 10
   A Word About Tools, Best Practices, and Teaching TDD 10
   Coming Up Next 11
   Changes for the Second Edition 12

3. **Test-Driven Development Basics** 13
   Infrastructure 13
   The Requirements 14
   Where to Start? 14
   Running Our test 17
   Passing Our Test 19
   The Second Test 21
   Our First Date 28
   Using the Time Data 31
   What We've Done 34

4. **Test-Driven Rails** 37
   And Now, Let's Write Some Rails 37
   The Days Are Action Packed 43
   Who Controls the Controller? 48
   A Test With A View 52
   What Have We Done? And What's Next? 56

5. **What Makes Great Tests** 57
The Big One 57
The Big Two 57
The More Detailed Five: SWIFT Tests 58

6. **Testing Models** 65
What Can We Do In A Model Test? 65
What Should I Test in a Model Test? 66
Okay, Funny Man, What Makes a Good Set of Model Tests? 66
Refactoring Models 68
A Note on Assertions Per Test 74
Testing What Rails Gives You 76
Testing ActiveRecord Finders 78
Modeling Data 80

7. **Adding Data To Tests** 81
What's the Problem? 82
Fixtures 83
Factories 89
Dates and Times 99
Summary 104

8. **Using Mock Objects** 105
Mock Objects Defined 106
Installing Mocha 107
Creating Stubs in Mocha 108
Mock Expectations 111
Using Mocks To Simulate Rails Save 113
Using Mocks To Specify Behavior 115
More Expectation Annotations 118
Mock Tips 121
Coming Up Next 123

9. **Testing Controllers And Views** 125
Testing Controllers 126
Simulating Requests in a Controller Test 127
Evaluating Controller Results 129
Testing Routes 132
View Tests 133
Testing Helper Methods 133
Testing View Markup 137

Presenters 141
Testing Mailers 144
Managing Controller and View Tests 148

10. Driving Behavior With RSpec 149
Getting Started With RSpec 150
RSpec in 10 Minutes 152
The "It" Factor 154
Great "Expect"ations 155
Mocking RSpec 159
Let RSpec Eat Cake 163
RSpec On Rails 165
RSpec and Models 165
Controllers in RSpec 167
RSpec and Views 169
RSpec Routing Tests 171
RSpec and Helpers 171
Write Your Own RSpec Matchers 172
RSpec and Sharing 175
Data About Metadata 177
What's Next 177

11. Integration Testing with Capybara and Cucumber 179
What to Test in an Integration Test 179
Setting up Capybara 182
Outside in Testing 183
Using Capybara 184
Making the Capybara Test Pass 189
Retrospective 196
Cucumber 197
Setting Up Cucumber 197
Writing Cucumber Features 198
Writing Cucumber Steps 200
More Advanced Cucumber 203
Is Cucumber Worth It? 205
Looking Ahead 206

12. Testing For Security 207
User Authentication and Authorization 207
Adding users and roles 209
Restricting Access 214

More Access Control Testing 218
Using Roles 219
Protection against form modification 225
Mass Assignment Testing 227
Other Security Resources 230

13. **Testing External Services** 231
External Testing Strategy 231
Our Service Integration Test 233
VCR 235
Client User tests 239
Why An Adapter? 241
Adapter tests 242
Testing for Error Cases 244
Smoke tests and VCR options 244
The World is a Service 246

CHAPTER 1

Changes in the Beta Releases

Beta 2.0, July 16, 2014

We have two new chapters for you:

- Chapter 12, *Testing For Security*, on page 207
- Chapter 13, *Testing External Services*, on page 231

Bug fixes

CHAPTER 2

A Testing Fable

Imagine two programmers working on the same task. Both are equally skilled, charming and delightful people, motivated to do a high-quality job as quickly as possible. The task is not trivial but not wildly complex either; for the sake of discussion, lets say it's a new user logging in to a website and entering some detailed pertinent information.

The first developer, who we'll call Sam, says, "This is pretty easy, and I've done it before. I don't need to write tests." And in five minutes Sam has a working method ready to verify.

Our second developer is named Jamie. Jamie says, "I need to write some tests." Jamie starts writing a test, which takes about five minutes. Five additional minutes later, Jamie also has a working method ready to verify. Because this is a fable, we are going to assume that Sam is allergic to automated testing, while Jamie is similarly averse to manually verifying against the app in the browser.

At this point, you might expect me to say that even though it has taken Jamie more time to write the method, Jamie has written code that is more likely to be correct, robust, and easy to maintain. That's true. I am going to say that. But I'm also going to say that there's a good chance Jamie will be done before Sam.

Let's watch our programmers as they keep working. Sam has a five-minute lead, but both of them need to verify their work. Sam needs to test in a browser; we said the task requires a user to log in. Let's say it takes Sam one minute to log in and perform the task to verify the code in a development environment. Jamie verifies by running the test—that takes about ten seconds. (At this point Jamie has to run only one test, not the entire suite.)

Let's say it takes each developer three tries to get it right. Since running the test is faster than verifying in the browser, Jamie gains a little bit each try. After verifying the code three times, Jamie is only two and half minutes behind Sam. (In a slight nod to reality, let's assume that both of them need to verify one last time in the browser once they think they are done. Since they both need to do this, it's not an advantage for either one.)

At this point, with the task complete, both break for lunch (a burrito for Jamie, an egg salad sandwich for Sam). After lunch, they start on the next task, which is a special case of the first task. Jamie has most of the test setup in place, so writing the test only takes two minutes. Still, it's not looking good for Jamie, even after another three rounds trying to get the code right. Jamie remains a solid two minutes behind Sam.

Let's get to the punch line. Sam and Jamie are both conscientious programmers, and they want to clean their code up with a little refactoring. Now Sam is in trouble. Each time Sam tries the refactoring, it takes two minutes to verify both tasks, but Jamie's test suite still takes only about ten seconds. After three more tries to get the refactoring right, Jamie finishes the whole thing and checks it in three and a half minutes ahead of Sam. (Jamie then catches a train home and has a pleasant evening. Sam just misses the train, and gets caught in a sudden rainstorm. If only Sam had run tests)

My story is simplified, but look at all the things I didn't assume. I didn't assume that the actual time Jamie spent on task was smaller, and I didn't assume that the tests would help Jamie find errors more easily — although I think Jamie would, in fact, find errors more easily. (Of course, I also didn't assume that Jamie would have to track down a broken test in some other part of the application, either.)

It is frequently faster to run multiple verifications of your code as an automated test than to always check manually. And that advantage is only going to increase as the code gets more complex. And the automated check is going to do a better job of not forgetting steps.

There are many beneficial side effects of having accurate tests. You'll have better-designed code in which you'll have more confidence. But the most important benefit is that if you do testing well, your work will go faster. You may not see it at first, but at some point in a well-run test-driven project, you'll notice that you have fewer bugs and that the bugs that do exist are easier to find. It will be easier to add new features and easier to modify existing ones. You'll be doing better on the only code-quality metric that has any validity: how easy it is to find incorrect behavior and add new behavior.

One reason why it is sometimes hard to pin down the benefit of testing is that good testing often just feels like you are doing a really good job programming.

Of course, it doesn't always work out that way. The tests might have bugs. They might be slow. Environmental issues may mean things that work in a test environment won't work in a development environment. Code changes will break tests. Adding tests to already existing code is a pain. Like any other programming tool, there are a lot of ways to cause yourself pain with testing.

Who You Are

The goal of this book is to show you how to apply automated testing and a test-driven process as you build your Rails application. We will see what tools are available and explore when those tools are best used. Tools come and tools go, so what I'm really hoping is that you come away from this book committed to the idea of writing better code through the small steps of a Test-Driven Development (TDD) or Behavior Driven Development (BDD) process.

There are some things I'm assuming about you.

I'm assuming that you are already comfortable with Ruby and Rails and that you don't need this book to explain how to get started creating a Rails application in and of itself. I am not assuming you have any particular familiarity with testing frameworks or testing tools used within Rails.

Over the course of this book, we'll go through the tools that are available for writing tests, and we'll talk about them with an eye toward making them useful in building your application. This is Rails, so naturally I have my own opinions, but all the tools and all the advice has the same goal: to help you to write great applications that do cool things and still catch the train home.

The Power of Testing First

The way to success with Test-Driven Development is trusting the process. The classic process goes like this:

1. Create a test. The test should be short and test for one thing in your code. The result of the test should be deterministic.

2. Make sure the test fails. Verifying the test failure before you write code helps ensure that the test really does what you expect.

3. Write the simplest code that could possibly make the test pass. Don't worry about good code yet. Don't look ahead. Sometimes, just write enough code to clear the current error.

4. After the test passes, refactor. Clean up duplication. Optimize. Create new abstractions. This is where design happens, so don't skip this. Remember to run the tests at the end to make sure you haven't changed any behavior.

Repeat until done. This will, in theory, ensure that your code is always as simple as possible and always is completely covered by tests. We'll spend most of the rest of this book talking about how to best manage this process using the tools of the Rails ecosystem and while solving the kinds of problems that you get in a modern web application. And we'll talk about the difference between "in theory" and "in practice".

If you use this process, you will find that it changes the structure of the code you write. The simple fact that you are continually aligning your code to the tests causes code that is made up of small methods, each of which does one thing. These methods tend to be loosely coupled and have minimal side effects.

As it happens, the hallmark of easy to change code is small methods that do one thing, are loosely coupled, and have minimal side effects. This list should seem familiar.

I used to think it was a coincidence that tested code and good code have similar structures. I've come to realize that the commonality is a direct side effect of building the code in tandem with the tests. In effect, the tests act as a universal client for the entire code base, guiding all the code to have clean interactions between parts because the tests, acting as a third-party interloper, have to get in between all the parts of the code in order to work. Metaphorically, your code has more surface area, and less work happening behind the scenes where it is hard to observe.

This theory explains why testing works so much better when the tests come first. Even waiting just a little bit to write tests is significantly more painful. When the tests are written first, in very close intertwined proximity to the code, then the tests drive the code's structure and encourage the code to have the good high-cohesion/low-coupling structure.

When the tests come later, they have to conform to the existing code, and it's amazing how quickly code written without tests will move toward low-cohesion and high-coupling forms that are much harder to cover with tests. If your only experience with writing unit tests comes only long after the initial code

was written, the experience was likely quite painful. Don't let that turn you away from a TDD approach; the tests and code you will write with TDD are much different.

When you are truly writing code that is "test-driven", then the tests are the final source of truth in your application. This means that when there is a discrepancy between the code and the tests, your first assumption is that the test is correct and the code is wrong. If you are writing tests after the code, then your assumption must be that the code is the source of truth. As you write your code using Test-Driven Development, keep in mind the idea that the tests are the source of truth and are guiding the structure of the code.

Prescription 1	In a Test-Driven process, if it is difficult to write tests for a feature, strongly consider the possibility that the underlying code needs to be changed.

What Is TDD Good For?

The primary purpose of the TDD style of testing is to go beyond mere verification and use tests to improve the structure of the code. That is, TDD is a software development technique masquerading as a code verification tool.

Automated developer tests are a wonderful way of showing that the program does what the developer thinks it does, but they are a lousy way of showing that what the developer thinks is what the program actually should do. "But the tests pass!" is not likely to be comforting to a customer when the developer's assumptions are just flat-out wrong. I speak from painful experience.

Automated developer testing is not a substitute for acceptance testing with users or customers (which can itself be partially automated via a tool like Cucumber) or for some kind of Quality Assurance phase where users or testers pound away at the actual program trying to break something.

TDD does not replace the role of a traditional software tester, it is a development process that produces better and more accurate code. A separate verification phase run by people who are not the original developers is still a good idea. For a good overview of more traditional exploratory testing, read *Explore It! [Hen13]*

Verification is valuable, but the idea of verification can be taken too far. You sometimes see an argument against Test-Driven Development that runs like this: "The purpose of testing is to verify that my program is correct. I can never prove correctness with 100 percent certainty. Therefore, testing has no

value." (Behavior-Driven Development and RSpec were created, in part, to combat this attitude.) Ultimately, though, testing has a lot of positive benefits to offer for coding, even beyond verification.

Preventing regression is often presented as one of the paramount benefits of a test-driven development process. And if you are expecting me to disagree out of spite, you're out of luck. Being able to squash regressions before anybody outside of your laptop sees them is one of the key ways in which strict testing will speed up your development over time.

Another common benefit you may have heard in connection with automated tests is that they provide an alternate method of documenting your program. The tests, in essence, provide a detailed, functional specification of the behavior of the program.

That's the theory. My experience with tests acting as documentation is mixed, to say the least. Still, it's useful to keep this in mind as a goal, and most of the things that make tests work better as documentation will also make the tests work better, period.

To make your tests effective as documentation, focus on giving your tests names that describe the tests reason for existence, keeping tests short, and refactoring out common setup and assertion parts. The documentation advantage of refactoring includes removing clutter from the test itself—when a test has a lot of raggedy setup and assertions, it can be hard for a reader to focus on the important features. As we'll see, a test that requires a lot of tricky setup often indicates a problem in the underlying code. Also, with common features factored out, it's easier to focus on what's different in each individual test.

In a testing environment, blank-page problems are almost completely nonexistent. I can always think of something that the program needs to do, so I write a test for that. When you're working test-first, the actual order in which pieces are written is not so important. Once a test is written, the path to the next one is usually clearer, and so on, and so on.

When TDD Needs Some Help

Test-Driven Development is very helpful, but it's not going to solve all of your development problems by itself. There are areas where developer testing doesn't apply or doesn't work very well.

I mentioned one case already—developer tests are not very good at determining whether the application is behaving correctly according to requirements. Strict

TDD is not very good at acceptance testing. There are, however, automated tools that do try to tackle acceptance testing. Within the Rails community, the most prominent of these is Cucumber; see Chapter 11, *Integration Testing with Capybara and Cucumber*, on page 179. Cucumber can be integrated with TDD — you'll sometimes see this called outside-in testing. That's a perfectly valid and useful test paradigm, but it's an extension of the classic TDD process.

Testing your application assumes that you know the right answer to specify. And although you will have clear requirements or a definitive source of correct output some of the time, other times you don't know what exactly the program needs to do. In this exploratory mode, TDD is less beneficial, because it's hard to write tests if you don't know what assertions to make about the program. Often this lack of direction happens during initial development or during a proof of concept. I also find myself in this position a lot when view testing—I don't know what to test for until I get some of the view up and visible.

The TDD process has a name for the kind of exploratory code you write while trying to figure out the needed functionality. This kind of programming is called a *spike*, as in, "I don't know if we can do what we need with the Twitter API; let's spend a day working on a spike for it." When working in spike mode, TDD is generally not used, but it's also the expectation that the code written during the spike is not used in production; it's just a proof of concept.

When view testing, or in other non-spike situations where I'm not quite sure what output to test for, I often go into a "test-next" mode, where I write the code first, but in a TDD-sized small chunk, and then immediately write the test. This works as long as I make the switch between test and code frequently enough to get the benefit of having the code and test inform each other's design.

TDD is not a complete solution for verifying your application. We've already talked about acceptance tests, but it's also true that TDD tends to be thin in terms of the quantity of unit tests written. For one thing, a strict TDD process would never write a test that you expect to pass before writing more code. In practice, though, you will do this all the time. Sometimes I see and create an abstraction in the code, but there are still valid test cases to write. In particular, I'll often write code for potential error conditions even if I think they are already covered in the code. It's a balance, because you lose some of the benefit of TDD by creating too many test cases that don't drive code changes. One way to keep the balance is to make a list of the test cases before you

start writing the tests—that way you'll remember to cover all the interesting cases.

And some things are just hard. In particular, some parts of your application are going to be very dependent on an external piece of code in a way that makes it hard to isolate them for unit testing. Mock objects can be one way to work around this issue. But there are definitely cases where the cost of testing a feature like this is higher than the value of the tests. To be clear, I don't think that is a common occurrence, but it would be wrong to pretend that there's never a case where the cost of the test is too high.

Recently, there has been discussion in the Rails community over whether the design benefits of TDD are even valuable, you may have heard the phrase "test-driven design damage." I strongly believe that TDD, and the relatively smaller and more numerous classes that a TDD process often brings, do result in clearer and more valuable code. But the TDD process is not a replacement for good design instincts, it's still possible to create bad code when testing, or even to create bad code in the name of testing.

Words to Live By

- Any change to the logic of the program should be driven by a failed test.
- If it's not tested, it's broken.
- Testing is supposed to help for the long term. The long term starts tomorrow, or maybe after lunch.
- It's not done until it works.
- Tests are code; refactor them too.
- Start a bug fix by writing a test.
- Tests monitor the quality of your code base. If it becomes difficult to write them, it often means you code base is too interdependent.

A Word About Tools, Best Practices, and Teaching TDD

We should have a quick word about tools.

There are two test libraries in general use in the Rails community: MiniTest and RSpec, meaning I had a choice of which tool to use as the primary library in this book.

MiniTest is part of the Ruby Standard Library (although as I write this, that fact is up for future discussion), and is therefore available everywhere you

might use Ruby (1.9 and up). It has a straightforward syntax that is the Ruby translation of the original SUnit and JUnit libraries, and it is the default alternative for a new Rails project.

RSpec is a separate testing tool specifically designed to support an emphasis on specifying behavior rather than implementation. So, rather than using terms like "test" and "assert", RSpec uses "expect" and "describe". It's hard to say for sure, but it's likely that more serious Rails test-driven code is written using RSpec than MiniTest. RSpec has a quirky, metaprogrammed syntax that has a certain "love it or hate it" vibe. It's more flexible, which also means more complicated, and it probably has a larger ecosystem of related tools.

In the end, I decided that MiniTest was a better fit for the learning purposes of this book, even though I tend toward RSpec in my personal use. While we will cover RSpec, and you will also be able to download a version of the sample code that uses RSpec, for teaching purposes it is my experience that it is easier to learn testing practices first and RSpec syntax second than it is to learn both testing practice and RSpec at the same time.

Which leads to a more general point – sometimes the best practice for learning something isn't the best practice for experts. In some cases in this book, we'll use more verbose or explicit versions of tools or tests to make it clear what the testing is trying to do and how. Where there's a significant difference between what might be expert practice, I'll try and note it in the text.

Coming Up Next

The next two chapters of this book will walk through a tutorial creating tests for a new Rails application. In Chapter 3, *Test-Driven Development Basics*, on page 13, we will start testing without using Rails-specific features, then in Chapter 4, *Test-Driven Rails*, on page 37, we will start to test Rails functionality.

After that, we'll spend a few chapters going through the basic blocks of a Rails program, testing Models, then testing Controllers and views. In that sections, we'll also talk about mock objects, a powerful tool for limiting where a test executes within your application.

Then we'll talk about alternative tools, spending a chapter on RSpec, a chapter on the integration test tools Capybara and Cucumber, and then JavaScript tools like Jasmine.

Once we are done talking about tools, we'll talk about specific scenarios for testing, including testing for security and testing third party services, and the very common case where you need to add tests to untested legacy code. We'll also talk about how to improve your test environment and run your tests quickly. And we will end with a discussion of what makes testing and tests most valuable.

Changes for the Second Edition

A lot has changed in the Rails testing world over the past five years, even if the general principles stayed more or less the same. This book has been substantially rewritten from its first edition, with almost no part of the book unchanged. Here's a more complete list of changes. (Not all of these changes will be in the early beta versions.)

- All tools have been upgraded to their latest versions: Rails 4.1.x, Minitest 5.3.x, RSpec 3, and so on.
- The opening tutorial was completely re-written. It's an all new example which provides, I hope, a more gentle introduction to testing in Rails.
- The code samples in general are better. In the first book, a lot of the samples after the tutorial were not part of the distributed code. Most of the samples in this book will tie back to the tutorial, and are runnable.
- The JavaScript chapter is nearly completely new to reflect changes both in tools and in the scope of JavaScript in most Rails appictions.
- There is an all-new chapter on testing external services
- There is an all-new chapter on testing for security
- A new chapter on debugging and troubleshooting. (Not in the initial beta)
- A new chapter on running tests more efficiently, looking at both the Spring/Zeus preloader option and the don't load Rails, plain old Ruby object option.
- Somewhat more emphasis, I hope, on using testing in practice, somewhat less on duplicating reference information.
- Some thing that were full chapters in the first book are de-emphasized, and covered sparingly if at all: Shoulda (since it's not really used anymore), Rails core integration tests (in lieu of spending more time on Capybara), Rcov, and Rails core performance testing.

Most importantly the entire community, including myself, has had five more years of experience with these tools, building bigger and better applications, learning what tools work, what tools scale, and what tools don't. This version of the book reflects these changes.

CHAPTER 3

Test-Driven Development Basics

You have a problem.

You are the team leader for a development team that is distributed across multiple locations. You'd like to be able to maintain a common list of tasks for the team, along with the state of the task, which pair of developers the task is assigned to, and so on. For some reason, none of the existing tools that do this are suitable (work with me here, folks) and so you've decided to roll your own. We'll call it Gatherer.

As you sit down to start working on Gatherer, your impulse is going to be to just start writing code. That's a great impulse, and we're just going to turn it about ten degrees east. Instead of starting off by writing code, we're going to start off by writing tests.

In our introductory chapter, we've already talked about why you might work test-first. In this chapter, we'll look at the basic mechanics of a TDD cycle by building a feature in a Rails application. We're going to start by creating some business logic with our Models, because model logic is the easiest part of a Rails application to test—in fact, most of this chapter won't touch Rails at all. In the next chapter, we'll start testing of the controller and view parts of the Rails framework.

Infrastructure

First off, we'll need a Rails application. We'll be using Rails 4.1.1 and Ruby 2.1.2, though we'll try not to use any Ruby 2.0 specific features.

We'll start by generating the Rails application from the command line:

```
% rails new gatherer
```

This will create the initial directory structure and code for a Rails application, it will also run bundle install to load initial gems. We assume that you are already familiar with Rails core concepts, so we're not going to spend a lot of time re-explaining them. If you are not familiar with Rails, *Agile Web Development with Rails [RTH11]* is still the gold standard for getting started.

We need to create our databases. For ease of setup and distribution, we'll stick to the Rails default, which is SQLite.

```
% cd gatherer
% rake db:create:all
% rake db:migrate
```

We need the db:migrate call, even though we haven't actually created a database migration because it sets up the schema.rb file that Rails uses to rebuild the test database.

The Requirements

The business logic we need to build is about forecasting the progress of a project. We want to be able to predict the end date of a project, and whether that project is or is not on schedule.

In other words: given a project, and a set of tasks, some of which are done and some of which are not, use the rate at which tasks are being completed to project the end date of the project. Also, compare that projected date to a deadline to determine if the project is on time.

This is a good example problem for TDD because, while I have a sense of what the answer is, I don't have a very strong sense of the best way to structure the algorithm. TDD is very helpful in guiding me toward reasonable code design.

Where to Start?

"Where do I start testing?" is one of the most common questions that people have when they start with TDD. Traditionally, my answer is a somewhat glib "start anywhere". While true, this is somewhat less than helpful.

A good option for starting a TDD cycle is the initialization state of the objects or methods under test. Another is the "happy path"—a single representative example of the error-free version of the algorithm. Which to choose depends on how complicated the feature is. In this case, the feature is sufficiently complex that I'll to start with the initial state and move to the happy path.

As a rule of thumb, if it takes more than one or two steps to define an instance of the application, then I'll start with initialization only.

> **Prescription 2** A good place to start a TDD process is with initializing objects. Another good place is to use the test to define what you want a successful interaction of the feature to look like.

This application is made up of projects and tasks. A newly initialized project would have no tasks? What can we say about that brand new project? Well, if there are no outstanding tasks, then there's nothing more to do. A project with nothing left to do is done. The initial state, then, is a project with no tasks and we can specify that the project is done. That's a design decision, we could also specify that a project with no tasks is in some kind of empty state.

We don't have any infrastructure in place yet, so I need to create the test file myself—I'm deliberately not using Rails generators right now. We're using Minitest, so the code goes in the Rails test directory parallel to where the application code will go in the app directory. We think this is a test of a project model, so we'll put the code in test/models/project_test.rb. Again, we're making very small design decisions here.

Here's our test of the initial state of a project.

basics/01/gatherer/test/models/project_test.rb

```
Line 1 require 'test_helper'
   2
   3 class ProjectTest < ActiveSupport::TestCase
   4
   5   test "a project with no tasks is done" do
   6     project = Project.new
   7     assert(project.done?)
   8   end
   9
  10 end
```

Let's talk about this test at two levels: the logistics of the code and also what this test is doing for us in our TDD process.

There are four interesting Minitest and Rails features in this file:

- Requiring test_helper
- Defining the test class
- Writing the test
- Asserting a particular state

On line 1, we require the file test_helper, which contains Rails and application related setup common to all tests. We'll peek into that file next chapter, when we talk about more Rails-specific test features.

We define our test class on line 3. In Minitest, you can't put a test method just anywhere, tests need to be methods of a subclass of the class Minitest::Test. Rails ActiveSupport provides a subclass called ActiveSupport::TestCase which provides a handful of shortcuts and goodies.

On line 5, we use one of those goodies, the test method. In standard Minitest, a test method is any method whose name starts with test_, as in test_this_thing_with_a_long_name. The Rails extension allows you to just say test "some string" followed by a block. Rails uses metaprogramming to convert test "some string" into an actual method called test_some_string, which invokes the block and is, by virtue of the name, executed by Minitest during a test run. I find the test_long_name syntax to be significantly less readable , so we'll be using the shortcut here. (Quick note for RSpec fans: unlike RSpec, the name munging that means that two tests in the same class can not have the same string description, since they'd resolve to the same Minitest name.)

Inside our test method we do two things. First, we create a Project instance. Then, on line 7, we make our first assertion, namely that the method call project.done? will result in a true value. The assert method is the most basic of Minitest's assertions. It takes one argument, and the assertion passes if the argument is true (for Ruby values of true), and fails if the argument is false.

Minitest defines about a dozen or so assertion methods, and also defines their opposites (for example, assert passes if the argument is true, while refute passes if the argument is false). Here are the six assertions that you probably will use most frequently. For a full list, see http://docs.seattlerb.org/minitest/Minitest/Assertions.html.

Assertion	Passes if
assert(test)	test is true
assert_block block	associated block returns true
assert_equal(expected, actual)	expected == actual
assert_includes(collection, object)	collection.include?(object)
assert_match(expected, actual)	expected =~ actual
assert_raises(exception) block	the associated block raises the exception

All Minitest assertions share two useful features.

First, They all take an optional message argument as the last argument. For example:

```
assert_equal("Noel", author.name, "Incorrect author name")
```

If the assertion fails, the message is output to the console. I don't normally use message arguments, because the messages are more overhead and clutter the tests. But if you are in a situation where documentation is particularly important, messages can be useful to describe the intent of a test.

Also, every assert method has an opposing refute method, as in refute_equal, refute_match, and so on. The refute methods pass where the assert methods would fail, so refute_equal passes if the two arguments are not equal. These are occasionally useful, but go light on them, I think most people find negative logic harder to reason about.

So, from a Minitest perspective we're creating an object and asserting an initial condition. What are we doing from a TDD perspective and why is this useful?

Small as it might seem, we've performed a little bit of design. We are starting to define the way in which parts of our system communicate with each other, and the tests are ensure the visibility of important information in our design.

This small test makes three claims about our program:

- There is a class called Project
- You can query instances of that class as to whether they are done
- A brand new instance of Project qualifies as done.

This last assertion isn't inevitable – we could say that you aren't done unless there is at least one completed task, but it's a choice we make in the design of our application.

Running Our test

Having written our first test, we'd like to execute it. Rails provides some standard Rake tasks for running all or part of the test suite.

The one you want to use most of the time, which is not the Rails default, is rake test:all, which grabs any file ending with _test.rb in the test directory or any of its subdirectories and executes them through Minitest. If you want to run a single file, then the way to do that in Minitest and Rails is the somewhat cumbersome rake test TEST=test/models/project_test.rb. For now, we're going to assume that we're using rake test:all. Later in *the (as yet) unwritten chp.environment* **I don't know how to generate a cross reference to chp.environment**, we'll cover some better ways to focus test execution.

What happens when we run the test?

It fails. We haven't written any code yet.

That's funny. What really happens?

When you run `rake test:all`, the Rake task identifies any file matching the pattern `test/**/*_test.rb` and passes them all to Minitest. (The related task `rake test:all:db` first resets the database using `db:test:prepare`.) Once Minitest gets the list of matching files, it does the following for each file.

- The Ruby interpreter loads the file. In a Rails context, the line `require test_helper` is important, as the `test_helper` file includes global and Rails-specific setup. Although it's pretty rare to have any class-level initialization in a test file, you'll sometimes have additional classes in the file besides the test class itself (you may define a dummy class that minimally implements a module you are testing, for example).
- Inside any subclass of `Minitest::TestCase`, Minitest identifies test methods in the file, either because the method name starts with `test` or because we're using the ActiveSupport `test` method directly.

That gives Minitest a list of test methods to run. For each of those methods, it does the following:

- Load or reset all fixture data. *Fixtures* are a Rails mechanism that defines global ActiveRecord data that is available to all tests. By default, fixtures are added once inside a database transaction that wraps all the tests. At the end of the test, the transaction is rolled back, allowing the next test to continue with a pristine state. More on fixtures in *Fixtures*, on page 83.
- Run all setup blocks. We'll talk about those more in a moment, when they become useful.
- Run the actual test method. The method execution ends when a runtime error or a failed assertion is encountered. If neither of those happens, the the test method passes. Yay!
- Run all teardown blocks. Teardown blocks are declared similarly to setup blocks, but their use is much less common.
- Rollback or delete the fixtures as described in step 1. The result of each test is passed back to the test runner for display in the console or IDE window running the test.

Minitest Test Execution shows the flow:

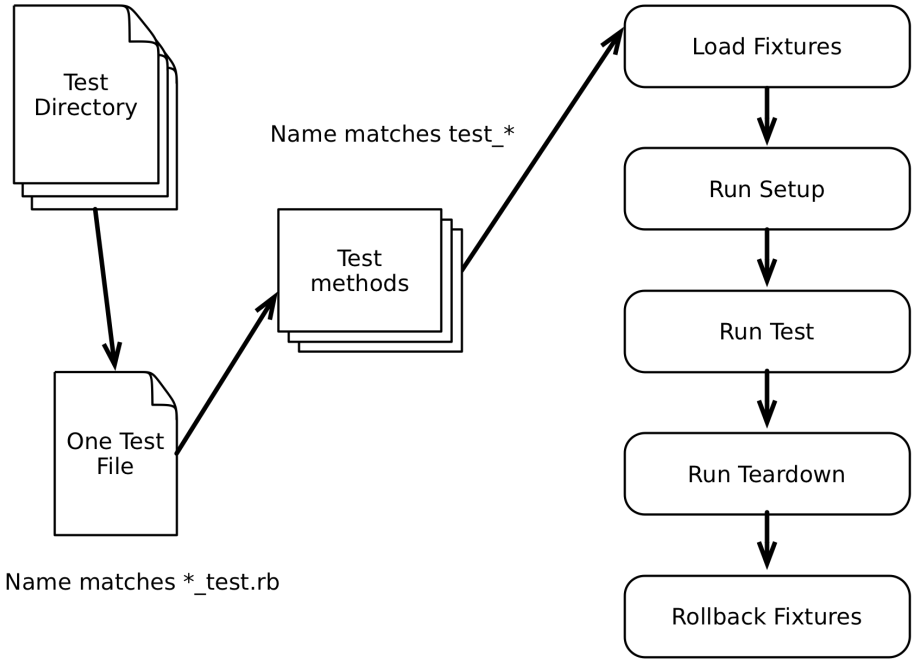


Figure 1—Minitest Test Execution

In our specific case, we've got one file, and one test method, and if we run things, we fail pretty quickly Here's the slightly edited output

```
$  rake test:all
E
Finished tests in 0.009388s, 106.5190 tests/s, 0.0000 assertions/s.

  1) Error:
ProjectTest#test_a_project_with_no_tasks_is_done:
NameError: uninitialized constant ProjectTest::Project
    test/models/project_test.rb:7:in `block in <class:ProjectTest>'

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

We've got one `E` for error – error because it's a runtime error, as opposed to `F` for failure when an assertion method fails.

Passing Our Test

Now it's finally time to make our first test pass.

But how?

It seems like a straightforward question, but it has a couple of different answers.

- The purist way. "Do the simplest thing that could possibly work." In this case "work" means "minimally pass the test without regard to the larger context". Or it might even mean "write the minimum amount of code to clear the current error without regard to the larger context"
- The practical way. Which is to write the code you know you need to eventually write, effectively skipping steps in the purist way that seem too small to be valuable.

Somewhere in-between is the teaching way, which is how I can best explain to you how and why Test-Driven Development works without either getting bogged down in details or skipping too many steps.

Let's make this test pass. The first error we need to clear is the `uninitialized constant: Project` error. So, put this in `app/models/project.rb`.

```
class Project
end
```

This is a minimal way to clear the error. (Technically not true, I could just declare a constant `Project = true` or something like that, but there's purist, and then there's crazy.) But the test still doesn't pass. If we run the tests now, we get `NoMethodError: undefined method done?`". But that's simple to clear:

```
class Project
  def done?
  end
end
```

And when I do this, and run `rake test:all` again, I finally get something interesting. Instead of an `E`, I get an `F`, and the message `Failed assertion, no message given` with a line number pointing to our one remaining assertion.

We've now passed out of the realm of syntax and runtime errors into the realm of assertion failures – our test runs, but the code does not behave as expected.

Luckily, that has a simple fix:

basics/01/gatherer/app/models/project.rb

```
class Project
  def done?
    true
  end
end
```

And, the test passes! We're done! Ship it!

Okay, we're not exactly done. We have made the test pass. Which actually only gets us 2/3 of the way through the TDD cycle. We've done Red (the failing test), and Green (the passing) test, and now we are into Refactor. However, we've written almost no code, so I think we can safely say that there are no refactorings indicated at this point.

At this point, I suspect that if you are inclined to be skeptical of Test-Driven Development, I haven't convinced you yet. We've gone on for a couple of pages, written one line of code and that line of code clearly isn't even final. I reiterate that in practice, this doesn't take much time. If we weren't stopping to explain each step, this would only take a couple of minutes, and some of that time – like setting up the Project class, would need to be done anyway.

In fact, we haven't exactly done nothing—we've defined and documented a subtle part of how our Project class behaves, and we will also find out immediately if the class ever breaks that behavior. As I've said, though, documentation and regression are only part of what makes Test-Driven Development powerful, we need to get to the design part. And for that, we need to write more tests.

The Second Test

One nice feature of Test-Driven Development at this stage is that making one test pass often points the way to the next test. The goal of the next test cycle is to write a test that fails given the current code. At this point, the code says that done? is always true, so we should create a case where done? is false.

basics/02/gatherer/test/models/project_test.rb

```
test "a project with an incomplete task is not done" do
  project = Project.new
  task = Task.new
  project.tasks << task
  refute(project.done?)
end
```

This test is similar to the first test, but now we've got a second class, Task, and a related attribute of the Project class, tasks. This time, we're assuming that a new task is undone, and therefore a project with an undone task is not done. We're using refute, the negation of assert, to express that expectation, because we're just clever that way.

Our first failure is the missing Task constant:

basics/02/gatherer/app/models/task.rb

```
class Task

end
```

And now we have simple, but still incomplete, done? logic to make the test pass – a project is done if it has no tasks:

basics/02/gatherer/app/models/project.rb

```
class Project
  attr_accessor :tasks

  def initialize
    @tasks = []
  end

  def done?
    tasks.empty?
  end
end
```

The second test now passes. And, we have a clear candidate for our next test – the distinction between complete and incomplete tasks. I'm also starting to think about how we get from there to sizing tasks and projecting project due dates.

With the test passing, we enter the refactoring phase. And even with the small amount of code written so far, I've got two refactoring jobs I'm thinking of.

If you've done Rails programming before, you may have noticed that I have not yet made Project and Task subclasses of ActiveRecord::Base, meaning that I haven't connected them to the database via ActiveRecord.

My reason for that is a purist reason—I haven't added any tests that would need ActiveRecord functionality to pass. (To a slightly lesser extent, I also don't want to further break the flow here to describe the data migrations and the like). That may seem pedantic—there's a pretty good chance that both these classes will become ActiveRecord classes—but it's not inevitable. And if you are using tests to drive design, it makes sense not to let the design leap ahead of the tests, but rather to use the tests to suggest the structure of the code.

For example, it's not a completely crazy design to say that Projects might never have data of their own and therefore might never need to be an ActiveRecord object. Somewhat more plausibly, the date projection code that we're writing now might eventually wind up in some kind of dedicated calculator object separate from the ActiveRecord layer. In either case, there's no need for our design to get ahead of our tests.

In addition to examining code for potential refactorings, it's also a good idea to look at the tests for duplication. In this case, we have a single line of com-

mon setup, namely project = Project.new, which is shared between the two tests that we have already written. Hold that thought.

Meantime, what's remaining of our definition of done? is the distinction between complete and not-complete tasks. Let's start with that, with a test for Task:

basics/02/gatherer/test/models/task_test.rb
```
require 'test_helper'

class TaskTest < ActiveSupport::TestCase

  test "a completed task is complete" do
    task = Task.new
    refute(task.complete?)
    task.mark_completed
    assert(task.complete?)
  end

end
```

This test actually makes two assertions, which I normally try to avoid, but the two assertions in this test are pretty intimately related—it would be more awkward to separate them. I create a new Task, assert that it is, at that point, not complete, complete the task, then assert that it is, in fact complete.

By including both assertions in the same test, we've basically written a shortcut-proof test—we have to put the real logic in the code to get both halves of this test to pass. More importantly, this test is completely at the API level and makes no claim about the underlying mechanism for representing completed tasks. This is great, because it means we can change the implementation without breaking tests as long as the API still works. Over-reliance on implementation details is a major cause of test fragility, so when you can describe the behavior rather than the implementation, you should do so.

Prescription 3 When possible, write your tests to describe your code's behavior, not its implementation.

We make that pass with some simple methods in Task

basics/03/gatherer/app/models/task.rb
```
class Task

  def initialize
    @completed = false
  end

  def mark_completed
```

```
    @completed = true
  end

  def complete?
    @completed
  end

end
```

This implementation probably won't survive long – tasks will probably grow more states. But for now, it works, and it's entirely possible that tasks will never get more complicated.

> **Prescription 4** Keeping your code as simple as possible allows you to focus complexity on the areas that really need complexity.

We can ensure the project's ability to determine if it is complete with a very similar project test:

basics/03/gatherer/test/models/project_test.rb

```
test "a project is only done if all its tasks are done" do
  project = Project.new
  task = Task.new
  project.tasks << task
  refute(project.done?)
  task.mark_completed
  assert(project.done?)
end
```

Which then passes by adding logic to our project's `done?` method.

basics/03/gatherer/app/models/project.rb

```
def done?
  tasks.reject(&:complete?).empty?
end
```

I can't think of an easy way to break the `done?` method as it currently stands, so it is… well, done.

Moving on. We need to be able to calculate how much of a project is remaining, and also the rate of completion, and then put them together to determine a projected ending date.

Now that we've got the basic infrastructure in place, we can go a little bit faster, which manifests itself in a test that has a little more setup. Our next test is for the project to be able to calculate how much work is remaining. I like to take a moment before I write the test to think about what the test needs. The typical structure of a test has three parts:

- Given: What data does the test need? This test needs a project, at least one complete task and at least one incomplete task.
- When: What action is taking place? We're calculating the remaining work.
- Then: What behavior do we need to specify? The result of the work calculations.

With that brief mental debate over, we write our test. For reasons that will be clear in a moment, we'll start a new test file.

basics/04/gatherer/test/models/project_with_data_test.rb

```
require 'test_helper'

class ProjectWithDataTest < ActiveSupport::TestCase

  test "a project can tell how much is left" do
    project = Project.new
    done = Task.new(size: 2, completed: true)
    small_not_done = Task.new(size: 1)
    large_not_done = Task.new(size: 4)
    project.tasks = [done, small_not_done, large_not_done]
    assert_equal(7, project.total_size)
    assert_equal(5, project.remaining_size)
  end

end
```

This test also has two assertions, but they are closely related and share a setup, so bear with me. I've set up a test where a project has two complete tasks and one incomplete task, and I'm asserting that the project correctly calculates size and size remaining. (I should say, that in RSpec, which has slightly better structures for sharing common setup, I'd be more likely to split this into two tests).

I've also made a couple of minor style choices that make the test easier to manage. I've given all the task objects meaningful names so that I can tell each object's reason for being in the test at a glance. If the tasks had descriptions or names I'd also give them meaningful data so that if the object gets printed to the terminal it's easy to tell which object it is. Also, the specific score numbers that I'm using for each are deliberate. Each task has a different score, and none of the two scores add up to the third, which is a very small thing that makes it harder to get a false positive test.

Prescription 5	Choose your test data and test variable names to make it easy to diagnose failures when they happen. Meaningful names and data that doesn't overlap are helpful.

This test fails first on the creation of Task.new(size: 2, completed: true) – Task isn't an ActiveRecord yet, so we don't have the hash argument by default. If this wasn't a book example, I would bring in ActiveRecord here, but I don't want to stop to define the migrations since they are basically irrelevant to the current point. We'll cover ActiveRecord when we bring in more Rails features.

basics/04/gatherer/app/models/task.rb

```
class Task

  attr_accessor :size, :completed

  def initialize(options = {})
    @completed = options[:completed]
    @size = options[:size]
  end
```

We than can make this pass with a couple more single line methods in Project.

basics/04/gatherer/app/models/project.rb

```
class Project
  attr_accessor :tasks

  def initialize
    @tasks = []
  end

  def done?
    tasks.reject(&:complete?).empty?
  end

  def total_size
    tasks.sum(&:size)
  end

  def remaining_size
    tasks.reject(&:complete?).sum(&:size)
  end

end
```

And the test passes.

This time, in the refactoring step, we actually have stuff to do. Let's do the code refactoring first. In Project, we have two methods both taking a list of incomplete tasks. We can extract that to common code:

basics/05/gatherer/app/models/project.rb

```
class Project
  attr_accessor :tasks
```

```
  def initialize
    @tasks = []
  end

  def incomplete_tasks
    tasks.reject(&:complete?)
  end

  def done?
    incomplete_tasks.empty?
  end

  def total_size
    tasks.sum(&:size)
  end

  def remaining_size
    incomplete_tasks.sum(&:size)
  end

end
```

This doesn't make our code shorter, but it does wrap a slightly opaque functional call containing a negative condition in a method with a semantically meaningful name. And if the definition of completeness changes, we only have to change one location.

We still have potential duplication in the Project class – two methods that call sum(&:size) on a list of tasks. I don't have an obvious place to put that method, though, short of creating a TaskList class. I don't see creating a TaskList as a simplification at this time, so we'll hold off. (A reviewer suggested this might also lead us to question if our Project class is actually the correct abstraction, and if TaskList might be better.)

I don't like having both the total size and the remaining size in the same test. If they are in the same test and the first assertion fails, the second assertion will not be evaluated, giving me an incomplete picture of my code. I don't want to duplicate the lines of setup though. Luckily Minitest and Rails ActiveSupport make it easy to create a setup method that can be shared by multiple tests in the same file.

basics/05/gatherer/test/models/project_with_data_test.rb

```
require 'test_helper'

class ProjectWithDataTest < ActiveSupport::TestCase

  setup :create_project_with_data
```

```
  def create_project_with_data
    @project = Project.new
    done = Task.new(size: 2, completed: true)
    small_not_done = Task.new(size: 1)
    large_not_done = Task.new(size: 4)
    @project.tasks = [done, small_not_done, large_not_done]
  end

  test "a project can tell its total size" do
    assert_equal(7, @project.total_size)
  end

  test "a project can tell how much is left" do
    assert_equal(5, @project.remaining_size)
  end

end
```

We've used the setup class method, which is part of ActiveSupport, to declare that the instance method create_project_with_data is part of our setup. This means the create_project_with_data method will be executed once before every individual test in the class. Also notice that @project is now an instance variable, allowing it to be created in the setup method and used in each test in the class. You can have multiple setup declarations, which are executed in the order in which they appear in the file. If you want something to happen at the end of each test, you use teardown instead of setup.

In standard Minitest, each test in a class shares the same setup, which is why I put these tests in a different file, to allow them to have a different setup than the initialization tests we wrote earlier.

We've split the tests to give us independent execution at the performance cost of running the setup method an extra time. Luckily for us, that cost is negligible at the moment. We're going to try to keep it that way.

Our First Date

We've got part of our Project API down, now we need to use that to calculate a projected completion date. The requirement is to calculate the end date of the project based on the number of tasks finished in the last three weeks. We're going to appropriate the agile term "velocity" to describe the rate of task completion. Which means we need to distinguish between tasks that finished in the last three weeks and tasks that did not.

Which means we have to deal with dates.

I'm sorry.

Programming with dates and times is the worst. Time is especially problematic in testing because tests work best when each test run is identical. However, owing to the nature of the universe, the current time inexorably changes from test run to test run. This can lead to all kinds of fun including tests that fail on or after a particular day, or tests that only pass at certain times of day. We're going to try to avoid all of that.

We're testing bottom-up, so it's a good idea to start at the smallest unit of code that we can think of. In this case, that's having Task instances be aware of whether they have been completed in the three-week window.

In the interest of keeping all of us sane and not walking through another set of trivial tests, I'll present the entire set Task tests. A test completed in the last three weeks counts toward velocity, which implies two negative cases: an incomplete test, and a test that was completed longer ago. To be clear, I wrote and passed them one at a time, but I don't think we need to walk through all those steps a second time.

basics/06/gatherer/test/models/task_test.rb

```
Line 1 test "an uncompleted task does not count toward velocity" do
   -   task = Task.new(size: 3)
   -   refute(task.counts_toward_velocity?)
   -   assert_equal(0, task.points_toward_velocity)
     5 end
   -
   - test "a task completed long ago does not count toward velocity" do
   -   task = Task.new(size: 3)
   -   task.mark_completed(6.months.ago)
    10   refute(task.counts_toward_velocity?)
   -   assert_equal(0, task.points_toward_velocity)
   - end
   -
   - test "a task completed recently counts toward velocity" do
    15   task = Task.new(size: 3)
   -   task.mark_completed(1.day.ago)
   -   assert(task.counts_toward_velocity?)
   -   assert_equal(3, task.points_toward_velocity)
   - end
```

We've got a couple of changes to the Task class implied in these tests.

First off, we've changed the existing mechanism for completing a task. On lines 9 and 16, we've changed the mark_completed call to take an optional date argument indicating the date completed. We'd like to do this without touching the existing tests that use mark_completed with no argument.

We've added two related methods to the Task class, namely counts_toward_velocity? and points_toward_velocity. According to our requirements, the counts_toward_velocity?

method returns true if the task has been completed in the last three weeks. As a matter of code style, we're naming the method and testing the behavior, rather than testing against the specifics of the implementation. By testing against the behavior, we hope that we will be better able to deal with the inevitable requirements changes.

The points_toward_velocity method is a little trickier—if the task counts toward velocity, then the size of the task is returned. If not, the method returns zero. This is an example of designing the class interface via tests. The idea is to keep all the logic for tasks inside the Task class. Specifically, I want a project to be able determine how much is remaining without having to query the task twice—once to determine the status of the task and again to determine its size.

The resulting Task class looks like this:

basics/06/gatherer/app/models/task.rb

```
class Task

  attr_accessor :size, :completed_at

  def initialize(options = {})
    mark_completed(options[:completed]) if options[:completed]
    @size = options[:size]
  end

  def mark_completed(date = nil)
    @completed_at = (date || Time.current)
  end

  def complete?
    !completed_at.nil?
  end

  def counts_toward_velocity?
    return false unless complete?
    completed_at > 3.weeks.ago
  end

  def points_toward_velocity
    if counts_toward_velocity? then size else 0 end
  end

end
```

Purely as a matter of testing style, notice the way dates are specified on lines 9 and 16 of the test file. I'm using the Rails helpers to concisely specify the dates relative to the current time, 6.months.ago for the out-of velocity task, and

1.day.ago for the in-velocity task. If I had specified an actual yesterday's date, then eventually the passage of time would push that date beyond the three week threshold and the test would fail. Using relative dates reduces that problem.

There's a different interesting question about the test design and the dates. Neither 6 months or 1 day is particularly close to the boundary. Shouldn't I test more days, or test something closer to the boundary? This question reflects the difference between testing as a design aid and testing for verification. In strict TDD, you would try to avoid writing a test that you expect to pass, because a passing test doesn't normally drive you to change the code.

I would only write a boundary condition test if I had reason to think that my implementation might fail in a boundary condition. Which is quite possible for dates and times, and often if I'm dealing with SQL date ranges versus Ruby date ranges, or if time zones are involved, I add tests near the boundary with the purpose of trying to break the implementation and catch an off-by-one error. I wouldn't write a series of tests for every length of time completed one day through six months, since I would expect all those tests to pass.

Using the Time Data

With the task tests passing, it's time to switch our attention back to the Project test. We need to make a slight tweak to our project_with_data_test setup so that we have tasks that are in and out of the three-week velocity window:

basics/06/gatherer/test/models/project_with_data_test.rb

```
def create_project_with_data
  @project = Project.new
  newly_done = Task.new(size: 3, completed: 1.day.ago)
  old_done = Task.new(size: 2, completed: 6.months.ago)
  small_not_done = Task.new(size: 1)
  large_not_done = Task.new(size: 4)
  @project.tasks = [newly_done, old_done, small_not_done, large_not_done]
end
```

We've added one more completed task, and are using the ability to pass a completed date to differentiate the two.

Now the calculations we need to actually determine the projected status of the project are straightforward math based on this data.

basics/06/gatherer/test/models/project_with_data_test.rb

```
Line 1 test "a project knows its velocity" do
   -   assert_equal(3, @project.completed_velocity)
   - end
   -
```

```
5 test "a project knows its rate" do
-   assert_equal(1.0 / 7, @project.current_rate)
- end
-
- test "a project knows its projected time remaining" do
10   assert_equal(35, @project.projected_days_remaining)
- end
-
- test "a project can tell if it is on schedule" do
-   @project.due_date = 1.week.from_now
15   refute(@project.on_schedule?)
-   @project.due_date = 6.months.from_now
-   assert(@project.on_schedule?)
- end
```

You can quibble with some style choices in these tests. Even though the tests are against the Project class, they have a stealth dependency on the Task class also working. That's not ideal, as it makes it harder to determine the cause if the test fails. Later, in Chapter 8, *Using Mock Objects*, on page 105, we'll see some strategies for breaking this kind of dependency in tests.

I also use a couple of different strategies for dealing with math. The assertion on line 6 has a mathematical answer expressed as a math expression in the test (1.0 / 7), while the assertion on line 10 does all the math and spits out the final answer (35). The algebraic version is clearer in the sense that it describes the way the answer is derived (and, in this case, makes it easier to express a floating-point answer), whereas the numerical version can seem magical – why 35? However, the downside of having code expressions in test assertions is that it encourages using the test to directly describe the final code, by copying and pasting the code from the test. It's often a better idea to have the implementation code be as independent as possible from the test itself.

The resulting passing code is almost anti-climactic – we've pushed almost all the conditional logic to the Task, making our Project code straightforward. This is a good sign, and implies that we're factoring the code reasonably.

basics/06/gatherer/app/models/project.rb

```
def completed_velocity
  tasks.sum(&:points_toward_velocity)
end

def current_rate
  completed_velocity * 1.0 / 21
end

def projected_days_remaining
  remaining_size / current_rate
end
```

```
def on_schedule?
  (Date.today + projected_days_remaining) <= due_date
end
```

In addition, we need to add attr_accessor :due_date to the Project class.

This passes the tests and moves us into the refactoring phase. I don't see anything in the code that screams for a refactoring (although one reviewer did suggest turning the rate into a Ruby Rational instance). I'm considering extracting the (Date.today + projected_days_remaining) logic to a method called projected_end_date, but don't need to do that at the moment.

We also want to look for potentially dangerous special cases to make sure they work. For example, the case where no tasks have been completed. We can put this test along with the other initialization tests in our original project_test.rb file.

basics/07/gatherer/test/models/project_test.rb

```
test "a project with no completed tasks projects correctly" do
  project = Project.new
  assert_equal(0, project.completed_velocity)
  assert_equal(0, project.current_rate)
  assert(project.projected_days_remaining.nan?)
  refute(project.on_schedule?)
end
```

I've put these all in one test to save space.

The first three assertions in this test pass as-is, the last one needs some code. The nan? assertion may seem a bit strange – Ruby's divide by zero construct is Float::NAN, but Minitest's assert_equal fails if you compare Float::NAN to itself, so we're using the provided predicate. Mostly what we want to make sure is that the projected_days_remaining doesn't raise an exception if there are no tasks.

We can use the same predicate in the code to make the on_schedule? assertion pass:

basics/07/gatherer/app/models/project.rb

```
def on_schedule?
  return false if projected_days_remaining.nan?
  (Date.today + projected_days_remaining) <= due_date
end
```

And the tests pass, which brings us to the refactoring phase. The first thing to notice is that we have a duplicated piece of data, namely the 21-day window for determining whether a task counts toward velocity. This data point is referenced in both Project#current_rate and Task#counts_toward_velocity. They are

pretty clearly the same bit of data – if I changed the time period to two weeks, I'd have to change both places.

That said, it's not clear what to do with this information. To me, the velocity length feels most like a static constant value owned by the Project class, since velocity applied to a single task makes no sense. In code, that looks like the following, with the velocity_length implemented as a class method with a constant return value.

basics/08/gatherer/app/models/project.rb
```
def self.velocity_length_in_days
  21
end
```

I'm using a method rather than a constant because this seems very likely to become dynamic at some point in the future, using a method preserves the API at no additional complexity cost.

The one usage in the Project class changes to:

basics/08/gatherer/app/models/project.rb
```
def current_rate
  completed_velocity * 1.0 / Project.velocity_length_in_days
end
```

And the one usage in Task is now:

basics/08/gatherer/app/models/task.rb
```
def counts_toward_velocity?
  return false unless complete?
  completed_at > Project.velocity_length_in_days.days.ago
end
```

And the tests pass. This structure eliminates the duplicate value, though the way that particular value is needed by both the Project and Task classes makes me wonder if we really just need a VelocityCalculator class.

What We've Done

Using the TDD process of "write a simple test, write simple code to make it pass, refactor" we started our Rails application by creating some business logic.

What has the TDD process given us? We started with a requirement and it was not immediately clear how to turn it into an algorithm. By using TDD, we were able to attack the problem incrementally, choosing to start in a small, well-understood, corner, and moving outward as our understanding of the

problem improves. It allows us to easily change our code structure as we learn more about the solution.

Most importantly, we write better code. The solution we end up with has short, well-named methods, it has logic in its proper place, and it will be easy to adjust as the requirements change.

Now it's time to take this model and integrate it into an actual web application. Let's do some Rails testing.

CHAPTER 4

Test-Driven Rails

In the last chapter, we created some basic functionality for a project management application using Test-Driven Development. The title of this book, though, is *Rails Test Prescriptions*, not *Generic Test Prescriptions*. (As with most generics, if that book did exist, it'd probably be cheaper but with less interesting packaging.)

In this chapter, we will augment our Model testing by testing logic in the Controller and View layers, as well as tests which cover our entire Rails application from request to response, called *end-to-end* tests. We'll be using a tool called Capybara to help manage our end-to-end testing.

A good test suite consists of a few end-to-end tests, a lot of tests that target a single unit, and relatively few tests that cover an intermediate amount of code. Controller and view tests often wind up in that mushy testing middle. However, by moving logic outside the controller and views themselves, we can turn those slower and more fragile middle-ground tests into faster and more robust unit tests.

And Now, Let's Write Some Rails

In order to have a Test-Driven Development process, it's important to have destination in mind. You need to have some requirements in mind when you start. Without some sense of what your code should be doing, it's hard to write tests to describe behavior.

Requirements gathering could be an entire book by itself.(Specifically, this one: *Software Requirements, 2nd Edition [Wie03]*) In our case, we're our own client, and it's a small project, so we don't exactly need military-grade precision. Here's my informal list of the first few things we're going to tackle:

- A user can create a project and seed it with initial tasks using the somewhat contrived syntax of "task name:size"
- A user can enter a task, associate it with a project, and see it on the project page
- A user can change the state of a task, to mark it as done
- A project can display its progress and status using the projection created last chapter.

We'll walk through these one by one, following the basic guideline that any new logic should be driven by a failing test. Let's start with the ability to enter a project.

End-To-End Testing

We're going to follow a testing practice called *Outside-In Testing*, which involves writing an end-to-end test (the "outside") which defines the feature, and then augmenting it with a series of unit tests ("the inside"), which drive the actual code and design.

We'll be using a tool called Capybara to make our end-to-end tests easier to read and write. Capybara allows for easy interaction with the web page and the Document Object Model (DOM). We'll cover features of Capybara as they come up, for full documentation of Capybara, check out its home page at https://github.com/jnicklas/capybara. We'll also cover Capybara and end-to-end testing in more detail Chapter 11, *Integration Testing with Capybara and Cucumber*, on page 179.

We're going to add one testing gem to get started, minitest-rails-capybara.

```
group :test do
  gem "minitest-rails-capybara"
end
```

And reinstall the bundle:

```
% bundle install
```

There's one more piece of setup we need. In the file test/test_helper.rb, add the following single line near the top:

```
require "minitest/rails/capybara"
```

That should be enough for us to start writing tests to support our first feature.

Why Capybara?

> Back at the beginning of the book I wrote at some length about sticking with the Rails core stack because it's the default and it's everywhere. Now, just a couple of chapters later, we're using Capybara, a third-party tool, to write a test.
>
> True. In my defense, where RSpec is larger and harder to explain than the default equivalent (which I say as an RSpec user...), Capybara is easier to use than the Rails integration tests it replaces. Like most of the Rails community, we are going to pretend that Rails default integration tests don't exist.
>
> But... watch this space. Rails creator David Heinemeir Hansson has suggested that Rails integration tests may get augmented in the near future.

Our first test covers the case where a user adds a task to a project. This task is going to be very close to Rails boilerplate, so our end-to-end test actually won't need much augmentation from unit tests. Later in this tutorial, we'll add features that need more business logic.

Let's plan out what this test needs, in terms of Given/When/Then. Since we're starting with empty data. no assumption about background data, we have no setup here. Our When, or action, is the act of filling out a form with project data and submitting. Our Then, or evaluation, is verifying that the new project displays on our list of projects with the entered tasks attached.

The code looks like this:

test_first/01/gatherer/test/integration/add_project_test.rb

```
Line 1  require "test_helper"
        -
        - class AddProjectTest < Capybara::Rails::TestCase
        -   test "a user can add a new project and give it tasks" do
      5     visit new_project_path
        -     fill_in "Name", with: "Project Runway"
        -     fill_in "Tasks", with: "Task 1:3\nTask2:5"
        -     click_on("Create Project")
        -     visit projects_path
     10     assert_content("Project Runway")
        -     assert_content("8")
        -   end
        -
        - end
```

We call this test outside, because it works from outside the Rails stack to define our functionality. We're simulating browser requests and evaluating browser responses. This test is not dependent on the structure of our code.

We have no setup in this test. The action uses a number of Capybara methods to interact with the web page. This section starts on line 5 and ends on line line 8.

In order to get the Capybara methods to work, we need to include the following line in our test/test_helper.rb file, toward the top of the file below the other requires.

```
require "minitest/rails/capybara"
```

This test uses the Capybara method visit to simulate a request to our application at the URL that matches the route new_project_path. Once it gets to that page, it uses the Capybara method fill_in to put text in a couple of form fields, then clicking on a button labeled Create Project, using the click_on method. We'll talk in more detail about the Capybara API in Chapter 11, *Integration Testing with Capybara and Cucumber*, on page 179, right now, it's enough to just get the gist of what the test is doing.

Finally, on line 9, we enter the evaluation phase of the test by visiting a route, projects_path, representing our project index page and asserting that the title of the new task appears on the page as well as the total size of the project – a task of size 3 and a task of size 5 means 8. We're not making any assumptions about the layout or presentation of the page, only that the new task name is there. Typically, when doing an end-to-end test, the goal is to have the success criteria be based on something that is visible in a response, rather than checking the database to see if the object is created.

This is a reasonable end-to-end test. It simulates a simple workflow by filling out a form, submitting it, and validating at least part of the resulting data.

There are several reasons why it's valuable to have a test like this one that works from outside the application.

- It makes no assumptions about the structure of the underlying code.
- It forces us to think of our feature in terms of behavior that is visible to a user or client of the application. Obviously, not all features have user-facing components, but where they do, being able to specify correct behavior without regard to the implementation is valuable.
- Eventually, our unit tests are going to focus on as small a part of the code as we can manage. Having one test that makes sure that all those little pieces correctly pass control between them prevents bugs from living in the gaps between those pieces.

Right now, this test will fail. Spectacularly. Absolutely none of the component bits are in place. So what we'll do is take this tiny step by tiny step, in each case minimally clearing the current error.

We can see the first error by running the test using rake test:all.

```
AddProjectTest#test_a_user_can_add_a_a_project_and_give_it_tasks:
NameError: undefined local variable or method `new_project_path' for
        #<AddProjectTest:0x007f83c88894a8>
    test/integration/add_project_test.rb:5:in `block in <class:AddProjectTest>'
```

Since Project isn't a RESTful ActiveRecord resource yet with routes, the test unsurprisingly can't find new_project_path.

Those of you who were reading the previous chapter and wondering when we would push to creating ActiveRecord models, your time has come. We'll use the Rails resource generator, which will create a controller, migration, route, and the like, but doesn't put any code in the generated controller. When we do the rails generate command, Rails will interactively ask us if we want to override the model project.rb and the model test project_test.rb. Don't override! We want to keep our existing code and update the model file by hand.

In the interest of sanity, we'll also update Task right now (otherwise keeping the tests passing while one is an ActiveRecord and the other isn't is a pain).

The exact commands we'll use are these:

```
% rails generate resource project name:string due_date:date
% rails generate resource task project:references
  title:string size:integer completed_at:datetime
```

We're adding two attributes to the project class: name, which we need for this test, and due_date, which is an attribute we added in the previous chapter. The Task model tests a title attribute for this test, and size and completed_at attributes from last chapter. Again, don't override the existing files

Then we change the project.rb file as follows—note that we are removing some code (like the initialize method), which is no longer needed because ActiveRecord is taking over the functionality.

test_first/01/gatherer/app/models/project.rb

```
class Project < ActiveRecord::Base
  has_many :tasks

  def total_size
    tasks.to_a.sum(&:size)
  end

  def remaining_size
```

```
    incomplete_tasks.to_a.sum(&:size)
  end

  def completed_velocity
    tasks.to_a.sum(&:points_toward_velocity)
  end
```

We've added the superclass ActiveRecord::Base, and we've removed the due_date attr_accessor, since ActiveRecord now manages attributes. The relationship to Task is now an ActiveRecord has_many, which also means that the code that calculates sums over the set of tasks needs to use to_a to convert the relationship to a plain array, or else Rails will give you an ugly deprecation warning.

Similarly, we clean up the Task class as follows:

test_first/01/gatherer/app/models/task.rb

```
class Task < ActiveRecord::Base

  def mark_completed(date = nil)
    self.completed_at = (date || Time.current)
  end

  def complete?
    !completed_at.nil?
  end

  def counts_toward_velocity?
    return false unless complete?
    completed_at > Project.velocity_length_in_days.days.ago
  end

  def points_toward_velocity
    if counts_toward_velocity? then size else 0 end
  end

end
```

We need to run our new migration.

```
% rake db:migrate
```

Versions of Rails before 4.1 will also need to run rake db:test:prepare, which keeps the test database in sync with the main schema. Rails 4.1 does this automatically after a migration is executed.

At this point, there are a couple of regression failures in the project_with_data_test.rb file, owing to the somewhat sloppy thing we did in the last chapter where we used a created flag to set a created_at attribute. The flag has gone away, so you need to change a couple of lines in project_with_data_test.rb

from, for example Task.new(size: 2, completed: 6.months.ago) to Task.new(size: 2, completed_at: 6.months.ago). Then rerun the tests to ensure they pass.

The Days Are Action Packed

Running the tests now gives us a different error, since new_project_path has been defined.

```
AddProjectTest#test_a_user_can_add_a_a_project_and_give_it_tasks:
AbstractController::ActionNotFound:
    The action 'new' could not be found for ProjectsController
    test/integration/add_project_test.rb:5:in `block in <class:AddProjectTest>'
```

We need a new action in our Projects controller. Since it is not going to have logic beyond Rails boilerplate, I don't think we need a separate test for it beyond the existing capybara test.

test_first/01/gatherer/app/controllers/projects_controller.rb

```
class ProjectsController < ApplicationController
  def new
    @project = Project.new
  end
end
```

Running the tests now triggers an error because Rails expects there to be a template file at /Users/app/views/projects/new.html.erb. After we create a blank file in that spot, we see an actual Capybara error:

```
Capybara::ElementNotFound: Unable to find field "Name"
     test/integration/add_project_test.rb:6:in `block in <class:AddProjectTest>'
```

Capybara searches for form items by any of their DOM id, form name, or the text of the associated label. In our case, we're using the label, but of course, since the view file is blank, it isn't there. We have three form elements to take care of – a text field for the name, a multiline text area for the tasks, and a submit button.

With the understanding that in a real project we would care about things like "design" and "making it look not ugly", we'll just put in a basic form that matches our needs.

test_first/01/gatherer/app/views/projects/new.html.erb

```
<h1>New Project</h1>

<%= form_for @project do |f| %>
  <%= f.label :name %>
  <%= f.text_field :name %>

  <br />
```

```
  <%= f.label :tasks %>
  <%= text_area_tag :"project[tasks]" %>

  <br />
  <%= f.submit %>
<% end %>
```

This is boilerplate with one exception – we're faking the text area for tasks by using a text_area_tag rather than the ActiveRecord data aware text_area method. This is because we're going to be doing some processing on the list of tasks, and tasks isn't a basic attribute of Project. If you do use f.text_area :tasks, then Rails tries to make the value of the text area the value of the tasks relation and places in that text area an ugly Ruby string representation of the empty relation, which is not what we want.

At this point the test will submit the form and fail. Now the test is looking for the create method in the controller that is invoked by submitting the form.

We have some decisions to make. We have some logic that goes beyond Rails boilerplate – namely, we need to parse that list of tasks and create Task instances out of them when the form is submitted. That code needs to go somewhere, and the unit tests we are about to write against that code need to know where that place is. This is where the design thinking comes in our TDD process.

No matter where we put the actual coding logic, Rails will still insist on the existence of a controller, so we have the separate decision of how to test whatever logic winds up in the controller itself.

Let's start with the business logic, we'll come back to the controller.

There are three locations that are commonly used for business logic that responds user input above and beyond the common "pass the params hash to ActiveRecord#create" Rails behavior.

- Put the extra logic in the controller. This is the Rails core team's preferred method, and if their isn't much logic, it works perfectly fine. In my experience this location doesn't work as well for complex logic. It's challenging to test, awkward to refactor, and difficult to share if that becomes an issue. It also becomes confusing if there is more than one complicated action in the controller.
- Put the extra logic in a class method of the associated model. This was my go-to move for years. It's somewhat easier to test, but still kind of

awkward to refactor – Ruby class method semantics are a pain. It also makes the model more complicated.

- Create a class to encapsulate the logic and workflow. This tends to be my first choice these days. It's the easiest to test, and the best able to manage complexity changes as they come. The main downside is you wind up with a lot of little classes, but I tend to prefer having a lot of little pieces anyway.

So, we're creating a new class. I'll stress here that this design is not the only way to go here, and if you feel the complexity of this particular action doesn't warrant its own class, that's fine. I'd like to show how moving logic outside of Rails objects works in a testing project. There is no consistent generic name for a logic class like this, we're going to call them Actions. Other names you might see in use include Service, Workflow, Context, Use Case, Concern and Factory.

Our action class needs to create a project from a name and a list of tasks. Let's start with the name:

test_first/01/gatherer/test/actions/creates_project_test.rb

```
require 'test_helper'

class CreatesProjectTest < ActiveSupport::TestCase

  test "creates a project given a name" do
    creator = CreatesProject.new(name: "Project Runway")
    creator.build
    assert_equal "Project Runway", creator.project.name
  end

end
```

It's a very straightforward test, which is the point. By not having this logic in the controller, we don't really need to do anything fancy to test it. We're calling the class CreatesProject, because I kind of like having action classes that aren't nouns. Alternate naming conventions might include CreateProject, ProjectCreator, or ProjectFactory.

And the passing code looks like this:

test_first/01/gatherer/app/actions/creates_project.rb

```
class CreatesProject

  attr_accessor :name, :task_string, :project

  def initialize(name: "", task_string: "")
    @name = name
    @task_string = task_string
```

```
  end

  def build
    self.project = Project.new(name: name)
  end

end
```

When I create an action object, I separate initializing the action, executing the action, and saving the resulting items. I separate them not just because it allows for easier testing, but because inevitably I find that when I have an object like this, there will come a time when I want to create an object without saving the result. It's much easier to test features of the action object if I can do so without touching the database by saving the result.

Also, we're using Ruby 2.0 keyword arguments in the initializer, as a cheap type check to make sure that the arguments passed to the CreatesProject initializer are limited to the ones we want.

Next up, we test the string parsing features. If you are following along, you should write these one at a time.

test_first/02/gatherer/test/actions/creates_project_test.rb

```
test "handles an empty string" do
  creator = CreatesProject.new(name: "Test", task_string: "")
  tasks = creator.convert_string_to_tasks
  assert_equal 0, tasks.size
end

test "handles a single string" do
  creator = CreatesProject.new(name: "Test", task_string: "start things")
  tasks = creator.convert_string_to_tasks
  assert_equal 1, tasks.size
  assert_equal "start things", tasks.first.title
  assert_equal 1, tasks.first.size
end

test "handles a single string with a size" do
  creator = CreatesProject.new(name: "Test", task_string: "start things:3")
  tasks = creator.convert_string_to_tasks
  assert_equal 1, tasks.size
  assert_equal "start things", tasks.first.title
  assert_equal 3, tasks.first.size
end

test "handles multiple tasks" do
  creator = CreatesProject.new(
      name: "Test", task_string: "start things:3\nend things:2")
  tasks = creator.convert_string_to_tasks
```

```
  assert_equal 2, tasks.size
end

test "saves a project with tasks" do
  creator = CreatesProject.new(name: "Project Runway",
      task_string: "start things:3\nend things:2")
  creator.build.save
  assert_equal 2, creator.project.tasks.size
  refute creator.project.new_record?
end
```

You can see the progression – an empty string parses to an empty list of tasks, a single element has a default size, then the size is set, and then multiple tasks separated by a \n line separator. All those tests call a convert_string_to_tasks method that allows us to test string parsing separate from any other feature of the class. After all that, we then do a mini-integration test where we explicitly call build and save to ensure that the task creation is picked up as part of the regular API process.

It is important to call save because sometimes Rails associations behave better when all the objects have been saved, since Rails uses ID numbers to track associated objects, and IDs are only assigned to saved objects. In tests, I will often call the Rails save! method, which throws an exception (and therefore fails the test) immediately if the object is invalid. Failing the test as quickly after an error happens is a good idea—if an object fails to save and causes a problem several lines later, that problem is harder to track down.

And back to the code:

test_first/02/gatherer/app/actions/creates_project.rb

```
def build
  self.project = Project.new(name: name)
  project.tasks = convert_string_to_tasks
  project
end


def convert_string_to_tasks
  task_string.split("\n").map do |task_string|
    title, size = task_string.split(":")
    size = 1 if (size.blank? || size.to_i.zero?)
    Task.new(title: title, size: size)
  end
end

def save
  project.save
end
```

```
def create
  build
  save
end
```

The only thing I might refactor here would be taking the inside of the loop and breaking it out into it's own method that parses a single string – I might also then refactor the relevant tests to use that method.

Who Controls the Controller?

All our new tests pass, so let's take stock and see where we are. We still have one test failure—our end to end test still doesn't like that the create action can't be found in the ProjectsController. We now have all the pieces we need to write that action.

Since we've put our business logic in the action object, the controller doesn't have much, but it does have some. Specifically, the controller sends data to the action object, then also sends data onward to the view layer. Finally, although we haven't stressed the point, the controller needs to do something in case the action object errors or does something else unexpected. Notice that we've separated responsibilities here — almost nothing that the controller does is dependent on the logic of creating and saving projects.

We're going to write a simple controller test that is the path of least resistance given the testing tools and libraries that we've discussed so far. Once we show the tests and make them pass, we'll discuss how the tests might be improved. Later in the book, as we introduce more testing tools, we'll revisit controller testing.

Rails provides a subclass of ActiveSupport::Test with a few extra features for testing controllers. It's called ActionController::TestCase. We'll create a subclass of ActionController::TestCase to write our first controller test.

test_first/02/gatherer/test/controllers/projects_controller_test.rb

```
Line 1 require 'test_helper'
  -
  - class ProjectsControllerTest < ActionController::TestCase
  -
  5   test "the project method creates a project" do
  -     post :create, project: {name: "Runway", tasks: "start something:2"}
  -     assert_redirected_to projects_path
  -     assert_equal "Runway", assigns[:action].project.name
  -   end
 10
  - end
```

This test simulates a call to a controller method, and then allows us to make assertions about what the controller does. Unlike the end-to-end test that we wrote at the start of this chapter, the controller test does not go through the entire Rails stack. Instead, it skips Rails routing, and calls the controller method directly.

Each line of this test has a feature that is unique to ActionController::TestCase. On line 6, we use a method named post. This method name has a striking resemblance to the HTTP request type POST, and in fact is used to trigger a controller action directly. In this test, the post method takes two arguments. The first one is a symbol naming the action to be invoked, and the second is a Hash representing the request parameters.

In addition to post, you also have controller test methods representing the other HTTP request types, such as get, put, and delete. However, the controller test does not verify that the routing engine actually matches the request type with the action – Rails will happily invoke the action even if the request type would not route to that action from the browser.

The Rails testing engine will run the controller action, setting the params hash to the parameters that you specify in the test. Rails will parse the associated view, but will not execute it unless you specifically make a view assertion. In other words, this test will fail with a parse error if the view contains an incorrect Ruby string like 1 +. However if the view contains semantically valid Ruby and you do not specifically make assertions against the view, the test will pass even if that code references values that don't exist. So the view could contain semantically valid nonsense like banana + cheese, and your test will be fine as long as you do not make an assertion against the specifics of the view.

On line 7, we make an assertion about what happens after the controller method completes, specifically that it will redirect to projects_path, or /projects. We could also use the method assert_template to specify that a particular template is invoked as a result of the controller action, which is useful in cases where you are doing something other than the Rails default.

Finally, line 8 specifies the controller's contract with the view, which we are claiming is to assign an instance variable @action to the actual action object. We're specifying that the action object actually creates a project by testing the name of that project—we don't need to test any more because further functionality is covered by the tests that we already wrote against the action object. The assigns variable on this line is a hash created by the test which matches the instance variables created by the controller. So the controller creates @action, and the test can reference that value with assigns[:action].

And that's our controller test. The passing code in the controller looks like this:

test_first/02/gatherer/app/controllers/projects_controller.rb

```
def create
  @action = CreatesProject.new(
    name: params[:project][:name],
    task_string: params[:project][:tasks])
  @action.create
  redirect_to projects_path
end
```

There is nothing complicated in the controller action itself, we create an action object, invoke it, and redirect.

Testing for failure

Failure, of course, is always an option, so we need to test for it. I prefer to do failure path testing in unit tests rather than end-to-end tests. Success requires the entire system to work in concert, failure response can usually be isolated to one component.

We haven't yet introduced a library that will allow us to fake failure – that requires a mock object package. But in this case, it's not hard for us to create a real failure by adding a validation that we can then not fulfill.

In the app/models/project.rb file, just add validates :name, presence: true. We're going to want all our projects to have names, so this validation seems perfectly reasonable.

Now, we trigger a failure by trying to create a project without a name.

test_first/03/gatherer/test/controllers/projects_controller_test.rb

```
Line 1 test "on failure we go back to the form" do
   2   post :create, project: {name: "", tasks: ""}
   3   assert_template :new
   4   refute_nil assigns(:project)
   5 end
```

We've seen most of this test before. On line 2, we use the same post method to invoke the controller action, but this time with a pair of empty strings in our form. On the next line, we use the assert_template method alluded to above to specify non-default behavior, specifically that we're redisplaying the form by rendering the new action template. Finally, since the new action needs an @project value for it to display, we use refute_nil to assert that the instance variable is actually assigned.

Making this code work requires us to use the fact that the CreatesProject action returns a boolean from the save action that is true if the action succeeds (more complex actions would probably maintain their own status as an instance variable.)

test_first/03/gatherer/app/controllers/projects_controller.rb

```
def create
  @action = CreatesProject.new(
    name: params[:project][:name],
    task_string: params[:project][:tasks])
  success = @action.create
  if success
    redirect_to projects_path
  else
    @project = @action.project
    render :new
  end
end
```

We've added the logic to switch the behavior of the controller based on the result of the action. (Some flavors of action objects would place the success and failure responses in the action class rather than the controller).

I don't think there's any more controller logic to add at this point, and I don't have a refactor for the controller code itself, so I think this round is over.

We've been able to cover the controller logic in just these two short tests because we placed the business logic in the action object. If we hadn't, all those tests we wrote for CreatesProject would be part of these controller test suite. As controller tests, they would run slower. More importantly, the tests would potentially be separated from the code where the expected failure would occur, making them less likely to drive design and less likely to be useful in troubleshooting.

That said, there are ways we can possibly make these tests better. In the style of outside-in testing that were demonstrating here, controller tests like these are being squeezed in two directions. If the controller test does a lot of verification of output, it runs the risk of merely duplicating the original end-to-end test. On the other hand, if the controller winds up interacting with the model layer, it can easily find itself duplicating the model test. Overlapping tests leads to a slower test suite and again, makes it harder to pinpoint problems when tests fail. Also, it was lucky for us that we were able to trigger a failure in our controller with a relatively simple validation. Sometimes we won't have that luxury.

We want to make the controller test completely isolated from the action object that it interacts with. The key insight is that the controller test only needs to test the behavior of the controller itself – the fact that the controller calls the action object with the correct parameters and uses the values as expected. Whether the action object works correctly, or even whether the action object exists is technically the action object test's problem. When testing the controller, the controller's behavior is what's important, not the action object's state.

There are a couple of reasons to isolate the controller test from the rest of the system. One reason is pure speed – calling the action object from the controller test would potentially require creating and saving ActiveRecords. This is relatively slow. And while the difference between a 0.2 second test and a 0.02 second test may not seem like much, after you write a couple hundred of them the difference becomes pretty salient. The second reason is isolation as an end it itself. The more tests that fail based on a single bug or point of regression, the harder it is to isolate and diagnose the actual problem.

Prescription 6	Test isolation makes it easier to understand test failures by limiting the scope of potential locations where the failure might have occurred.

If we're sold on the idea that we want to test the controller's behavior separate from whatever it might call, the next question is how. What we want is to allow the controller to send all the messages it needs to, but to somehow have those messages not actually do anything. Or, to put it another way, instead of having the controller interact with a real action object, we need to interact with a fake one.

These fake objects are most commonly called *mock objects*, and they are valuable in a variety of situations where an object you want to interact with is expensive or risky to deal with. We'll talk quite a bit about when to use mock objects in Chapter 8, *Using Mock Objects*, on page 105. For now, I just want you to be vaguely dissatisfied with this test.

A Test With A View

Meanwhile, we still have this end-to-end test to wrap up. Let's show that test again:

test_first/03/gatherer/test/integration/add_project_test.rb

```
require "test_helper"

class AddProjectTest < Capybara::Rails::TestCase
```

```
  test "a user can add a a project and give it tasks" do
    visit new_project_path
    fill_in "Name", with: "Project Runway"
    fill_in "Tasks", with: "Task 1:3\nTask2:5"
    click_on("Create Project")
    visit projects_path
    assert_content("Project Runway")
    assert_content("8")
  end

end
```

So far, we've gotten this test to pass up to the visit projects_path line. This line triggers a visit to the path /projects, which is routed to the index method of the ProjectController. Since we don't have one of those, our current error is The action 'index' could not be found for ProjectsController

So we'll need the index action.

test_first/03/gatherer/app/controllers/projects_controller.rb

```
def index
  @projects = Project.all
end
```

And now we have an action without a view, resulting in the error message that starts: ActionView::MissingTemplate: Missing template projects/index,.... To get past this error, we create a blank view file at app/views/projects/index.html.erb.

Now our error is on the following line of the test:

```
AddProjectTest#test_a_user_can_add_a_a_project_and_give_it_tasks
[gatherer/test/integration/add_project_test.rb:10]:
Expected to include "Project Runway".
```

At this point, the Capypara test is evaluating the actual HTML response from our application. We're asking the test to assert_content, which is defined by the minitest-capybara-rails gem and which passes if the response either literally contains a string argument or matches a regular expression argument. Specifically, we're asserting that the output contains the string Project Runway, which is the name of the newly created project and the string 8, which is its size.

This is a very weak test. There are all kinds of ways, for example, that the number 8 could appear in our response HTML. (To pick one unlikely scenario, our user could be journalist and author Jennifer 8. Lee http://en.wikipedia.org/wiki/Jennifer_8._Lee.) The struggle when view testing is to find a balance between a test that validates something meaningful about the output and one that isn't so tied to the actual markup that it will break when a designer looks at the page cross-eyed.

Let's make the test pass first, then explore how we can make the test a little stronger. Passing the test requires a straightforward Rails view:

test_first/03/gatherer/app/views/projects/index.html.erb

```
<h1>All Projects</h1>
<table>
  <thead>
    <tr>
      <td>Project Name</td>
      <td>Total Project Size</td>
    </tr>
  </thead>
  <tbody>
    <% @projects.each do |project| %>
      <tr>
        <td><%= project.name %></td>
        <td><%= project.total_size %></td>
      </tr>
    <% end %>
  </tbody>
</table>
```

Obviously, we're not concerned with making this pretty, in a production application, presumably the markup would be more complex.

When this executes, our newly entered project gets its own table row, complete with its name and size.

And our end-to-end test finally passes.

Which puts us in a refactoring phase. We didn't write much code in this step, but I'd like to take the opportunity to refactor the last couple lines of the end to end test, using the capybara-rails assert_selector method. This is a really common work pattern for me. Sometimes I have trouble seeing the shape of a view before I write it, so I write a very loose test to get me there, then tighten the test once I see what pieces of the view will exist for me to hook on to.

The assert_selector method takes as its argument a jQuery style selector, with the usual # representing a DOM id, and . representing a DOM class. The assertion passes if the actual page contains a DOM element that matches the selector. You can also specify a text: option that means the matching DOM element must also contain particular text (or match a particular regular expression).

With assert_selector, we can rewrite the test as follows:

test_first/04/gatherer/test/integration/add_project_test.rb

```
require "test_helper"
```

```
class AddProjectTest < Capybara::Rails::TestCase
  test "a user can add a a project and give it tasks" do
    visit new_project_path
    fill_in "Name", with: "Project Runway"
    fill_in "Tasks", with: "Task 1:3\nTask2:5"
    click_on("Create Project")
    visit projects_path
    @project = Project.find_by_name("Project Runway")
    assert_selector("#project_#{@project.id} .name", text: "Project Runway")
    assert_selector("#project_#{@project.id} .total-size", text: "8")
  end

end
```

In our final two lines, we are still testing for the same text, but now we are forcing it to appear in a specific part of the page. We want each bit of text to be associated with a DOM id representing the project item – using the Rails-blessed pattern <class>_<id>, and then a DOM class representing type. This gives us a stronger test – no longer would a random 8 somewhere on the page cause a pass – now the 8 specifically has to be associated with the size of this project.

However, the test isn't completely brittle – nothing specifies, for example, that the elements are table rows and cells. So if we go off and redesign this page using more modern markup, as long as the size element is subordinate to the project element, the test will still pass.

The view needs only minor changes to make this test pass.

test_first/04/gatherer/app/views/projects/index.html.erb

```
<h1>All Projects</h1>
<table>
  <thead>
    <tr>
      <td>Project Name</td>
      <td>Total Project Size</td>
    </tr>
  </thead>
  <tbody>
    <% @projects.each do |project| %>
      <tr class="project-row" id="<%= dom_id(project) %>">
        <td class="name"><%= project.name %></td>
        <td class="total-size"><%= project.total_size %></td>
      </tr>
    <% end %>
  </tbody>
</table>
```

We've only added a few DOM id's and DOM classes.

And with that, the test passes again, and I think we've got this feature in the books.

What Have We Done? And What's Next?

We've written an entire, albeit small, piece of Rails functionality starting with an end-to-end integration test, and moving down to unit tests to make each individual part of the feature work. And it only took us one chapter. It goes a lot faster when you don't stop to explain every line of code.

From here, we'll proceed in a couple of different directions. The next few chapters will go more in depth. We'll look at model testing, controller testing, and view testing covering the libraries discussed in this chapter in more detail. There will also be discussion of some related topics like placing data in tests, testing for security, and testing JavaScript. After that, we'll tackle some wider topics. How to test legacy code, how to keep your tests from becoming legacy code, how to test external services, and the like.

First, though, let's step back for a second and talk about what makes automated testing effective.

CHAPTER 5

What Makes Great Tests

As the Rails community has matured, Rails developers are much more likely to be working with codebases and test suites that contain five years worth of work. As a result, there has been a lot of discussion about design strategies to manage complexity over time.

There hasn't been nearly as much discussion over what practices make tests and test suites continue to be valuable over time. As applications get bigger, as suite runs get longer, as complexity grows, how can you write tests that will be useful in the future, and not act as an impediment to future development.

The Big One

The best, most general, piece of advice I can give about the style and structure of automated tests is this:

> **Prescription 7** Your tests are also code. Specifically, your tests are code that does not have tests.

Your code is verified by your tests, but your tests are verified by nothing.

Having your tests be as clear and manageable as possible is the only way to keep your tests honest.

The Big Two

There are two important criteria that any programming tool or practice must meet:

- Does doing or using X make it easier for me to add the code I need in the short-term?

- Does doing or using X make it easier for me to continue adding code to this project over time?

There are all kinds of gems in the Ruby and Rails ecosystem that help with the first goal (including Rails itself). Testing is normally thought of as working toward the second goal. That's true, but often people assume that the only contribution testing makes toward long term application health is verification and preventing regressions. In fact, over the long term, test-driven development tends to pay off as good tests lead toward modular designs.

This means a valuable test saves time and effort over the long term, while a poor test costs time and effort. I've focused on five qualities that tend to make a test save time and effort. The absence of these qualities, on the other hand, is often a sign that the test could be a problem in the future.

The More Detailed Five: SWIFT Tests

I have five criteria that I like to use to evaluate test quality. I have even managed to turn them into an acronym that is only slightly contrived: SWIFT.

- Straightforward
- Well-defined
- Independent
- Fast
- Truthful

Let's explore those in more detail.

Straightforward

A test is *straightforward* if its purpose is immediately understandable.

Straightforwardness in testing goes beyond just having clear code. A straightforward test is also clear about how it fits into the larger test suite. Every test should have a point: it should test something different from the other tests, and that purpose should be easy to discern from reading the test.

Here is a test that is not straightforward:

```
## Don't do this
test "the sum should be 37" do
  assert_equal(37, User.all_total_points)
end
```

Where does the 37 come from? It's part of the global setup. If you were to peek into the user fixture file of this fake example, you'd see that somehow

the totals of the total points of all the users in that file add up to 37. The test passes. Yay?

There are two relevant problems with this test:

- The 37, which is a magic literal that apparently comes from nowhere
- The name of the test is utterly opaque about whether this is a test for the main-line case, a test for a common error condition, or a test that exists only because the coder was bored and thought it would be fun.

Combine these problems, and it quickly becomes next to impossible to fix the test a few months later when a change to the User class or the fixture file breaks it.

Naming tests is a critical to being straightforward. Creating data with more locally and more explicitly also helps. With most factory tools (see *Factories*, on page 89, default values are pre-set, so the description of an object created in the test can be limited to only defining the attributes that are actually important to the behavior being tested. Showing those attributes in the test is an important clue toward the programmer intent. Rewriting the above test with a little more information might result in this:

```
test "total points should round to the nearest integer" do
  User.create(:points => 32.1)
  User.create(:points => 5.3)
  assert_equal(37, User.all_total_points)
end
```

It's not poetry, but at the very least, an interested reader now knows where that pesky 37 comes from and where the test fits in the grand scheme of things. The reader might then have a better chance of fixing the test if something breaks. The test is also more independent, since it no longer relies on global fixtures—making it less likely to break.

Long tests or long setup tend to muddy the water and make it hard to identify the critical parts of the test. The same principles that guide refactoring and cleaning up code apply to tests. This is especially true of the rule that states "A method should only do one thing," which here means splitting up test setups into semantically meaningful parts, as well as keeping each test focused on one particular goal.

On the other hand, if you can't write short tests, consider the possibility that it is the code's fault, and you need to do some redesign. If it's hard to set up a short test, that often indicates the code has too many internal dependencies.

There's an old programming adage that since debugging is more complicated than coding, if you've written code that is as complicated as you can make it, then you are by definition not skilled enough to debug it. Because tests don't have their own tests, this adage suggests that you should keep your tests simple, so as to give yourself some cognitive room to understand them.

In particular, this guideline argues against using clever tricks to reduce duplication among multiple tests that share a similar structure. If you find yourself starting to metaprogram to generate multiple tests in your suite, you're probably going to find that complexity working against you at some point. You never want to be in a position to have to decide whether a bug is in your test or in the code. And when—not if—you do find a bug in your test suite, it's an easier place to be if the test code is simple.

We'll talk more about clarity issues throughout the book. In particular, the issue will come up when we discuss factories vs. fixtures as ways of adding test data in Chapter 7, *Adding Data To Tests*, on page 81, and when we talk about RSpec in Chapter 10, *Driving Behavior With RSpec*, on page 149.

Well-defined

A test is *well-defined* if running the same test repeatedly gives the same result. If your tests are not well defined, the symptom will be intermittent, seemingly random, test failure.

Two classic causes of repeatability problems are time and date testing and random numbers. In both cases, the issue is that your test data changes from test to test. Dates and time have a nasty habit of monotonically increasing, while random data tends to stubbornly insist on being random. Similarly, tests that depend on third-party service, or even test code that makes Ajax calls back to your own application, can also vary from test-run to test run, causing intermittent failures.

Dates and times tend to lead to intermittent failures when certain magic time boundaries are crossed. You can also get tests that fail at particular times of day or fail when run in certain time zones. Random numbers, in contrast, make it somewhat difficult to test both the randomness of the number and that the randomly generated number is used properly in whatever calculation requires it.

The test plan is similar for dates, randomness, and external services—really, it applies to any constantly changing dataset. You test changing data with a combination of encapsulation and mocking. You encapsulate the data by creating a service object that wraps around the changing functionality. By

mediating access to the changing functionality, you make it easier to stub or mock the output values. Stubbing the values provides the consistency you need for testing.

You might, for example, create a RandomService class that wraps Ruby's rand() method:

```
class RandomStream
  def next
    rand()
  end
end
```

This example is a little oversimplified—normally, you'd be encapsulating Random as part of some larger service inside your application. With your own wrapper class, you can provide more specific methods tuned to your use case, something like `def random_phone_number`. First you unit test the steam class to verify that the class works as expected. Then any class that uses `RandomStream` can be provided mock random values to allow for easier and more stable testing.

The exact mix of encapsulation and mocking varies. Timecop is a Ruby gem that stubs the time and date classes with no encapsulation. This allows you to specify an exact value for the current time for the purpose of a test. That said, nearly every time I talk about Timecop in a public forum, someone points out that creating a time service is a superior solution.

This pattern for wrapping a potentially variable external service will be discussed in more detail in Chapter 13, *Testing External Services*, on page 231. Mock objects will be covered in Chapter 8, *Using Mock Objects*, on page 105, and we'll talk more about debugging intermittent test failures in *the (as yet) unwritten chp.debugging*I don't know how to generate a cross reference to chp.debugging.

Independent

A test is *independent* if it does not depend on any other tests or external data to run. An independent test suite gives the same results no matter what order the tests are run and also tends to limit the scope of test failures to only those tests that cover a buggy method.

In contrast, a very dependent test suite could trigger failures throughout your tests from a single change in one part of an application. A clear sign that your tests are not independent is if you have test failures that happen only when the test suite is run in a particular order—in fully independent tests, the

order in which they are run should not matter. Another sign is a single line of code breaking multiple tests.

The biggest impediment to independence in the test suite itself is the use of global data. Rails fixtures are not the only possible cause of global data in a Rails test suite, but they are a common cause. Somewhat less common in a Rails context is using a tool or third-party library in a setup and not tearing it down.

Outside the test suite, if the application code is not well encapsulated, it may be difficult or impossible to make the tests fully independent of one another.

Fast

It's easy to overlook the importance of pure speed in the long-term maintenance of a test suite or a TDD practice. In the beginning, it doesn't make much difference. When you only have a few methods under test, the difference between a second per test and a tenth of a second per test is almost imperceptible.

The difference between a one-minute suite and a six second suite is easier to discern.

From there, the sky's the limit. I worked in one Rails shop where nobody really knew how long the tests ran in development, because they farmed the test suite out to a server farm that was more powerful than most production web servers I've seen. This is bad.

Slow test suites hurt you in a variety of ways.

There are startup costs. In the sample TDD session we went through in Chapter 3, *Test-Driven Development Basics*, on page 13 and Chapter 4, *Test-Driven Rails*, on page 37 we went back and forth to run the tests a lot. In practice, I went back and forth even more frequently. Over the course of writing that tutorial, I ran the tests dozens of times. Imagine what happens if it takes even 10 seconds to start a test run. Or a minute, which is not out of the question for a larger Rails app. I've worked on JRuby based applications that took well over a minute to start.

TDD is about flow in the moment, and the ability to go back and forth between running tests and writing code without breaking focus is crucial to being able to use TDD as a design tool. If you can check Twitter while your tests are running, you just aren't going to get the full value of the TDD process.

There are a number of reasons why tests get slow, but the most important in a Rails context are:

- Startup time
- Dependencies within the code that require a lot of objects to be created in order to invoke the method under test.
- Extensive use of the database during a test

Not only do large trees of objects slow down the test at runtime, but setting up large amounts of data makes writing the tests slower more labor intensive. And if writing the tests becomes burdensome, you aren't going to do it.

Speeding tests up often means isolating application logic from the Rails stack so that logic can be tested without loading the entire Rails stack, or without retrieving test data from the database. As with a lot of good testing practice, this isolation results in more robust code that is easier to change moving forward.

Since test speed is so important for successful TDD, we will be discussing ways to write fast tests in many places throughout the book. In particular, the discussion of creating data in Chapter 7, *Adding Data To Tests*, on page 81, and the discussion of testing environments in *the (as yet) unwritten chp.environment*I don't know how to generate a cross reference to chp.environment will be concerned with creating fast tests.

Truthful

A *truthful* test accurately reflects the underlying code – it passes when the underlying code works, and fails when it does not. This is easier said than done.

A frequent cause of brittle tests is targeting assertions at surface features that might change even if the underlying logic stays the same. The classic example along these lines is view testing, in which you base the assertion on the actual creative text on the page that will frequently change, even though the basic logic stays the same.

```
test "the view should show the project section" do
  get :dashboard
  assert_select("h2", :text => "My Projects")
end
```

It seems a perfectly valid test right up until somebody decides that "My Projects" is a lame header and decides to go with "My Happy Fun-Time Projects." And breaks your test. You are often better served by testing something that slightly insulated from surface changes, like a DOM id.

```
test "the view should show the project section" do
  get :dashboard
```

```
  assert_select("h2#projects")
end
```

The basic issue here is not limited to view testing. There are areas of model testing in which testing to a surface feature might be brittle in the face of trivial changes to the model (as opposed to tests that are brittle in the face of changes to the test data itself, which we've already discussed).

The other side of robustness is not just a test that fails when the logic is good but a test that stubbornly passes even if the underlying code is bad—a tautology, in other words.

Speaking of tautologies, mock objects have their own special robustness issues. It's easy to create a tautology by using a mock object. It's also easy to create a brittle test by virtue of the fact that a mock object often creates a hard expectation of exactly what methods will be called on the mock object. If you add an unexpected method call to the code being tested, then you can get mock object failures simply because an unexpected method has been called. I've had changes to a login filter cause hundreds of test failures because mock users going through the login filter bounced off the new call.

CHAPTER 6

Testing Models

A standard Rails application uses a design pattern called *MVC*, which stands for Model, View, Controller. Each of the three sections in the MVC pattern is a separate layer of code, with it's own responsibilities and which communicates with the other layers as infrequently as possible. In Rails, the model layer contains both business logic and persistence logic, with the persistence logic being handled by ActiveRecord. Typically, all of your ActiveRecord objects will be part of the model layer, but not everything in the model layer is an ActiveRecord object. The model can include various services, value objects, or other classes which encapsulate logic and use ActiveRecord objects for storage.

We're going to start our tour of testing the Rails stack with the model layer because model tests have the fewest dependencies on Rails-specific features and are often the easiest place to start testing your application. Standard Rails model tests are very nearly just vanilla Minitest. Features specific to Rails include the ability to set up initial data, additional assertions, and a block syntax for describing setup and teardown.

We'll talk about testing ActiveRecord features like associations and models. And we'll talk about separating logic from persistence, and why that can be a valuable practice both for testing and for application development.

What Can We Do In A Model Test?

Rails model tests are subclasses of `ActiveSupport::TestCase`, which is a subclass of the core Ruby `Minitest::Unit::TestCase`. Model tests also include a couple of modules from the Rails core mixed in to provide additional functionality. All told, the following functionality is added to Ruby unit tests to make them Rails model tests:

- The ability to load data from fixtures before tests.
- The declarative `test "test name" do` syntax that we've already been using.
- For every Minitest `refute` method, Rails throws in a similar `assert_not`, as in `assert_not_equal`. Weirdly, the method name for `refute_match`, however, is `assert_no_match`. My understanding is that somebody on the Rails core team really doesn't like the `refute` syntax.
- The assertions `assert_difference`, `assert_no_difference`, `assert_blank`, and `assert_presence`. (If you are playing along at home with a copy of the first version of the book, the `assert_valid(foo)` test mentioned there has been deprecated in favor of `assert foo.valid?`)
- Support for asserting the existence of Rails deprecations, unlikely to be helpful unless you are actually working on Rails itself.
- Logging support to put the test class and test name in the Rails `test.log` before each test.
- Support for multiple `setup` and `teardown` blocks, defined with the method name, similar to Rails callbacks or filters.

The `test/test_helper.rb` file should be required by all tests that load Rails. This loads the Rails test environment, and checks for any pending data migrations. The helper file is a good place to put common setup for all your tests. The standard local `test_helper.rb` file requires the `rails/test_helper.rb` file, which is part of the Rails framework. This global file requires the Rails test classes, loads fixtures, and does some other housekeeping to tie your tests into the Rails environment.

What Should I Test in a Model Test?

Models. Next question?

Okay, Funny Man, What Makes a Good Set of Model Tests?

There are a lot of different, sometimes conflicting, goals for tests. It's hard to know where to start your TDD process, how many tests to write, and when you are done. The classic description of the process: write a simple test, make it pass, refactor, doesn't provide a lot of affirmative guidance or direction for the larger process of adding a feature.

A Meta-TDD Process

Here's a meta-process that reflects how I write a new business logic feature (I handle view logic a little differently). This process is more of a guideline than a hard-and-fast checklist. As the logic that we are working on gets more

complex, and the less we know about the implementation when we start, the closer we should stick to this process and the smaller the steps we take.

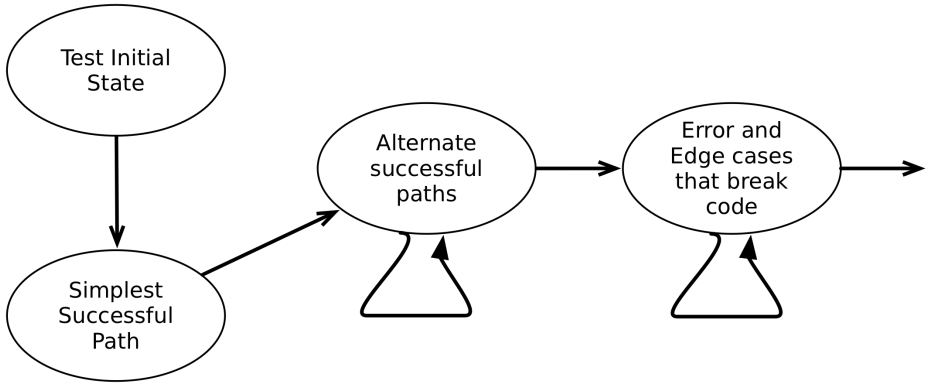


Figure 2—A Meta-TDD Process

Often, the best place to start is with a test that describes an initial state of the system, without invoking logic. This is especially useful if you are test-driving a new class, so the first test just sets up the class and verifies the initial state of instance variables. If you are working in an existing class, this step may not be necessary.

Next, determine the main cases that drive the new logic. Sometimes, there will be only one primary case, like "calculate the total price of all a user's purchases to date." Sometimes there will be many: "calculate tax on a purchase" might have lots of cases based on the user's location or the specific items being purchased.

Take that list and write tests for the main cases one at a time. I do not recommend actually writing multiple failing tests at a time – in my experience that's confusing. It can be helpful to use comments or pending tests to at least note what the future tests will be. Ideally these tests are small – if they need a lot of setup, there's a good chance you should be testing a smaller unit of code.

Sometimes you'll pass the first test by putting in a solution that is deliberately specific to the test, like doing nothing but returning the integer that happens to pass the test (Gary Bernhardt of Destroy All Software calls this technique "sliming"). This can be helpful in an initial keeping your code close to the tests, again, especially when the algorithm is complex. If you find yourself writing tests that already pass given the current state of the code, that often means you are writing too much code in each pass.

The goal in these main case tests is to first make the test pass quickly without worrying too much about niceties of implementation. Once each test passes, look for opportunities to refactor – about which more in the next section.

When the main cases are done, what you try to do then is think of ways to break the existing code. Sometimes you'll notice something as you are writing code to pass a previous test, like "hey, I wonder what would happen if this argument was nil?". Write a test that describes what the output should be and make it pass. Refactoring gets increasingly important here, because special cases and error conditions tend to be what makes code complex, and managing that complexity becomes really important to future versions of the code. The advantage of waiting to do special cases at the end is that you already have tests to cover the normal cases, so you can use that to check your new code each step of the way.

When you can no longer think of a way to break your code, there's a good chance you are done with this feature and are ready to move on to the next one. If you haven't been doing so, run the entire test suite at this point to make sure you didn't inadvertently break something else. Then take one further look at the code for larger-scale refactoring.

Refactoring Models

In a Test-Driven Design process, most of the design part takes place during the refactoring step. A lot of this design happens under the guise of cleanup – looking at parts of the code that seem overly complicated or poorly structured and figuring out how best to rearrange them.

Just because the refactoring step includes cleanup doesn't mean that it's something that you can skip when you are in a hurry. You'll often see people talk about refactoring or cleaning up code as a luxury that can easily be thrown overboard in a crunch. Don't do that. Refactoring is where you think about your code and how best to structure it.

Prescription 8	Refactoring is where design happens in a TDD process, and it's easiest to do in small steps. Skip at your peril.

At the most abstract level, you are looking for three things: complexity to break up, duplication to combine, and abstractions waiting to be born.

Break up complexity

Complexity will often initially manifest itself as long methods or long lines of code—often, if you are doing a "quick to green" first pass of the code to make

the tests pass, there will be a line of code with a lot of method chaining or something.

It's almost always a good idea to break up long methods or long lines by extracting part of the code into its own method. In addition to simplifying the original method, you have the opportunity to give the method you are creating a name that is meaningful in the domain and which can work to make your code self-documenting.

There are three signs in particular that are almost automatic candidates for extraction:

Prescription 9	Try to extract methods when you see compound booleans, local variables, or inline comments

- Any compound boolean logic expression goes in its own method. It's much harder for people to understand what a compound expression does than you might think. Give people reading your code a fighting chance by hiding the expression behind a name that expresses the intent of the code, like `valid_name?` or `has_purchased_before?`.
- Local variables are relatively easy to break out into methods with the same name as the variable – in Ruby, code that uses the variable doesn't need to change if the variable becomes a method with no arguments. Having a lot of local variables is a huge drag on complex refactorings – you'll be surprised at how much more flexible your code feels if you minimize the number of local variables in methods. (I first encountered this idea, along with a lot of other great refactoring ideas, in *Refactoring: Improving the Design of Existing Code [FBBO99]*.)
- In long methods, you'll sometimes see a single line comment breaking up the method by describing what the next part does. This nearly always is better extracted out to a separate method with a name based on the contents of the comment. Instead of one twenty-five line method, you wind up with a five-line method that calls five other five-line methods, each of which does one thing, and each of which has a name that is meaningful in the context of the application domain.

Combine duplication

There are three kinds of duplication that you need to look out for – duplication of facts, duplication of logic, and duplication of structure.

Duplication of fact is usually easy to see and easy to fix. A common case would be a "magic number" used by multiple parts of the code, like a conversion factor, or a maximum value. Often a status variable has only a few valid values, and the list of those values is duplicated.

```
validates :size, numericality: {less_than: 5}

def possible_sizes
  (1 .. 5)
end
```

The remedy for duplication of fact is also usually simple – make the value a constant or a method with a constant return value.

```
MAX_POINT_COUNT = 5

validates :size, numericality: {less_than: MAX_POINT_COUNT}

def possible_sizes
  (1 .. MAX_POINT_COUNT)
 end
```

or, alternately

```
VALID_POINT_RANGE = 1 .. 5
validates :size, inclusion: {in: VALID_POINT_RANGE}
```

That said, at some point, the extra character count for a constant is ridiculous and Java-like in the worst way. For string and symbol constants, if the constant value is effectively identical to the symbol (as in `ACTIVE_STATUS = :active`), I'll often just leave the duplication. I'm not saying I recommend that, just saying I do it.

Another thing I'll do is make the constant value an instance method with a static return value rather than a Ruby constant, as in:

```
def max_point_count
  5
end
```

I do this because then `max_point_count` has the same lookup semantics as any other instance value, and it also often reads better to make a value owned by the instance rather than the class. It's also easier to change if the constant turns out to be less than constant in the future.

Duplication of logic is similar to duplication of fact, but instead of looking for simple values, we're looking for longer structures. This will often include compound boolean statements used as guards in multiple methods (in our task manager example, this might be something about whether a task has

been completed), or simple calculations that are used in multiple places (such as converting task size to time based on the project's rate of completion).

In this example, the same boolean test is applied twice, and it's easy to imagine it being used many more times.

```
class User

  def maximum_posts
    if status == :trusted then 10 else 5 end
  end

  def urls_in_replies
    if status == :trusted then 3 else 0 end
  end

end
```

One thing to do here is move the duplicated logic into its own method and call the method from each location. (In this case def trusted?). I recommend being aggressive about this – you'll sometimes see advice that you just notice duplication on the second instance, and refactor on the third instance. In my experience, that just means you wind up with twice as much duplication as you should have. But, see the next section for other ideas about reused booleans.

The one thing you do need to keep in mind is that not every piece of logic that is spelled the same in Ruby is actually duplication. It's possible for early forms of two pieces of logic to look similar, but eventually evolve in separate directions. A great example of this is Rails controller scaffolding. Every RESTful controller starts with the same boilerplate code for the seven RESTful actions. And there have been innumerable attempts to remove that duplication by creating a common abstraction. Most of those attempts eventually wind up in a tangle of special case logic because each controller is actually going to eventually need to have different features.

Finding missing abstractions

Duplication of structure often means there's a missing abstraction, which in Ruby generally means that you can move some code into a new class.

A symptom to look for is a subset of instance attributes that are always used together. Especially notice if you have the same set of attributes being passed to multiple methods. Use of a common set of variables often indicates that you want a new class with those friendly attributes as the instance attributes.

Another common symptom is a group of methods that all share a prefix or a suffix, like, logger_init, logger_print, logger_read. Often this means you need a class corresponding to the common word.

In ActiveRecord, one side effect of discovering friendly attributes is the creation of value objects, which are immutable instances that represent parts of your data. For example, a start_date and end_date are often used together, and could easily be combined into a DateRange class.

Or how often do you write something like this?

```
class User < ActiveRecord::Base
  def full_name
    "#{first_name} #{last_name}"
   end

   def sort_name
     "#{last_name}, #{first_name}"
   end
 end
```

You could try this:

```
class Name
  attr_reader :first_name, :last_name

  def initialize(first_name, last_name)
    @first_name, @last_name = first_name, last_name
  end

  def full_name
    "#{first_name} #{last_name}"
   end

   def sort_name
     "#{last_name}, #{first_name}"
   end
 end

 class User < ActiveRecord::Base
   delegate :full_name, :sort_name, to: :name
   def name
     Name.new(first_name, last_name)
   end
 end
```

If you have existing tests on User in that situation, those tests should continue to pass.

Normally, I don't change tests when I refactor – the goal is to test functionality, not implementation. I will sometimes make an exception to this rule if I move the code to a new class, especially if I expect the new class to be shared often.

When you break out related attributes into their own class, as in this Name example, you'll often find it's much easier to add complexity when you have a dedicated place for that logic. When you need middle names or titles, it's easier to manage that in a separate class than it would be if you had a half-implementation of names in multiple classes.

You'll also find that these small classes are easy to test because not only does Name no longer have a dependency on the database, it doesn't have a dependency on anything. Without dependencies, it's easy to set up and write fast tests for name logic.

```
test "sort name" do
  name = Name.new("Noel", "Rappin")
  assert_equal("Rappin, Noel", name.sort_name)
end
```

The test for just the name class is quick to write and also to run. The easier it is to write tests, the more tests you'll write.

Another kind of structural abstraction you should be on the lookout for is repeated if statements or other conditionals that are switching on the same values. A common example of this is continually checking for a nil value. Another is frequent checking against a status variable. Our task tracker, for example, might have a lot of methods that do this:

```
if status == :completed
  calculate_completed_time
else
  calculate_incompleted_time
end
```

Obviously, every program needs the occasional if statement. But if you find yourself continually checking the state of an object to determine what to do next, you should at least consider the possibility that you actually have a new set of classes. For example, the above snippet implies the existence of something like CompleteTask and IncompleteTask. (Or possibly completeness and incompleteness only affect part of the class functionality, so you get something like CompleteTaskCalculator and IncompleteTaskCalculator)

Once you have separated out functionality into separate classes, an Object-Oriented program is supposed to switch based on class using message passing and polymorphism:

```
def calculator
  if complete?
    CompleteTaskCalculator.new(self)
  else
    IncompleteTaskCalculator.new(self)
  end
end

def calculate
  calculator.calculate_time
end
```

In the second example, we still have an if statement about the logic between complete and incomplete tasks, but only one. Any further difference between complete and incomplete tasks is handled in the difference between the two calculator classes. If you have many places where you are testing for completion, then this form can be clearer.

Again, these classes typically don't depend on the database making them easy to write fast tests for, and again you'll often find these classes attracting behavior – once they exist, it's easier to see what behavior belongs there.

A Note on Assertions Per Test

You will often find that a common setup results in multiple assertions in a test. This is particularly true of integration tests. For example, when we created a new project we asserted that the project existed, and also asserted that it had a specific relationship with newly created tests.

There are two contrasting styles for writing tests with multiple assertions. In one style, the setup and all the assertions are part of the same test. If we were trying to test changes when we mark a task complete, then having all the assertions in the same test might look like this.

```
test "mark a task complete" do
  task = tasks(:incomplete)
  task.mark_complete
  assert task.completed?
  assert task.blocked?
  assert_equal(Date.today.to_s(:db),task.end_date)
  assert_equal("completed", task.most_recent_log.end_state)
end
```

In contrast, you could put each assertion in a separate test, and put the common setup in a setup block. Using the same set of assertions in separate tests, looks like this:

```
setup do
  @task = tasks(:incomplete).complete!
end

test "mark a task complete and it should not be blocked" do
  assert !@task.blocked?
end

test "mark a task complete and thetask should be completed" do
  assert @task.completed?
end

test "mark a task complete and set the end date" do
  assert_equal(Date.today.to_s(:db), @task.end_date)
end

test "mark a task complete and sent the log" do
  assert_equal("completed", @task.most_recent_log.end_state)
end
```

Before we talk about the trade-offs, just a couple of quick logistical points. The one-assertion per test version needs to make @task an instance variable so that it can be shared between the setup and the test. The one-assertion per test version seems a little verbose here, but both Minitest and RSpec have mechanisms that streamline the typing somewhat.

The trade-off is pretty plain: the one assertion per test style has the advantage that each assertion can fail independently—when all the assertions are in a single test, the test bails on the first failure. In the all-in-one test, if assert task.completed? fails, you won't even get to the check for assert !task.blocked. If all the assertions are in separate tests, everything runs independently, but it's harder to determine how tests are related. There are two significant downsides to the one assertion style. A significant speed difference, since the single-assertion-per-test version will run the common setup multiple times. The single assertion style can also become difficult to read, especially if the setup and test wind up with some distance between them.

A compromise I often make is to write my first pass at TDD in the one-assertion per test style, which forces me to work in baby steps, and gives me a more accurate picture of what tests are failing. When I'm confident in the correctness of the code, I consolidate related assertions, giving me the speed benefit moving forward.

That said, when you are writing tests are actually different in that they cover different branches of the method under test, they should be different test methods. You should never change local variables inside a test method just to test a different branch in the same test. So you would want to keep these tests separate:

```
test "full names" do
  user = User.create(:first_name => "Fred", :last_name => "Flintstone")
  assert_equal("Fred Flintstone", user.full name)
end

test "full names with a middle initial" do
  user = User.create(:first_name => "Fred", :last_name => "Flintstone"
:middle_initial => "D")
  assert_equal("Fred D. Flintstone", user.full name)
end

test "full name where there's no first name" do
  user = User.create(:last_name => "Flintstone")
  assert_equal("Flintstone", user.full name)
end
```

Continually changing the value of user to put all those assertions in the same branch makes for a test that is very hard to read and understand.

Testing What Rails Gives You

Rails provides built-in functionality for associations and validations, which leads to questions about how to effectively and usefully test those features in your application.

The answer in both cases is similar, and goes back to the basic principle that we're testing functionality and not implementation. While I do not normally write tests just to show the existence of a particular association or validation, I will write tests that show those features in action. For associations, this means showing the association in use. For validations, it means testing the overall logic of what makes an instance valid.

The testing gem Shoulda, https://github.com/thoughtbot/shoulda, defines matchers that specifically test for the existence of validations and associations, like so:

```
class TaskTest < ActiveSupport::TestCase
  should belong_to(:project)
  should belong_to(:user)
  should ensure_length_of(:name)
end
```

Tests like that are not particularly valuable for a TDD process because they are not about the design of new features. If you are doing the TDD process, you shouldn't start from the idea that your Task belongs to a Project. Rather, as you describe features, the relationship is implied from the feature tests that you are writing. More operationally, this means that in a good TDD process, any condition in the code that would cause a direct test like those Shoulda matchers to fail would also cause another test to fail. In which case, what's the point of the Shoulda matcher?

To briefly and half-heartedly argue the other side, you often don't need to go through a whole TDD process to know that a relationship or a validation should exist, and these tests don't cost very much to write. And to rebut myself, part of doing the TDD process is to force you to examine the things you think you know and prove that they are really necessary. In particular, if I see a lot of those should belong_to lines of code in a test suite, rightly or wrongly, I'm going to be worried about the test suite's effectiveness.

Validations are a little different, since whether a piece of data is valid is often a legitimate and complex piece of business logic and the Rails helpers only cover a set of common cases. I think you can make validations in your code work efficiently with the TDD process if you focus on the functional part – "this object is valid", vs. "this object is not", rather than the "I'm using a Rails numerical validator" part. Also, consider only using an actual Rails database-blocking validation as a last resort when you sincerely believe that it's worth potentially raising an exception rather than having this data in your database.

What this means is that I would typically test for the functional effect of an invalid object rather than the implementation fact of the existence of a Rails validation. The functional effect is often along the lines of an object or objects not saving to the database. If we wanted to augment our project tracker project creation to require that all the tasks for the new project have a size, then we might try something like this, very similar to the "don't create a project without a name test" we already wrote:

```
test "don't create a task without a size" do
  creator = CreatesProject.new(name: "Test", task_string: "size:3\nno_size")
  creator.create
  assert_equal ["size"], creator.project.tasks.map(&:title)
end
```

In this test, we're validating that the second task, with the clever name "no size", does not get added to the project. This test works whether the size limitation is implemented as a Rails validation or as some kind of filter that the CreatesProject class manages. Again, testing the behavior, not the implementa-

tion. This strategy works for all kinds of Rails validations, including uniqueness (create two objects and validate that the second one doesn't save). When I create a custom validator, though, either as a method or as a separate class, those go through the same TDD process as any other method.

Testing ActiveRecord Finders

ActiveRecord provides a rich set of methods that are wrappers around SQL statements sent to your database. These methods are collectively referred to as *finders*. One great feature of ActiveRecord finders is that they can be composed, allowing you to express a compound statement like "bring me the most recently completed 5 large tasks" as `Task.where(status: completed).order("completed_at DESC").where("size > 3").limit(5)`.

You can even compose the finders if you extract them to their own methods in pieces:

```
class Task < ActiveRecord::Base

  def self.completed
    where(status: :completed)
  end

  def self.large
    where("size > 3")
  end

  def self.most_recent
    order("completed_at DESC")
  end

  def self.recent_done_and_large
    completed.large.most_recent.limit(5)
  end
end
```

Being able to compose this logic is awesome. But finder methods occupy an awkward place between methods you might write and Rails core features, leading to the question of how best to test them.

Here are some guidelines.

Be aggressive about extracting compound finder statements to their own method, in much the same way and for much the same reason as I recommended for compound Boolean logic. The methods are easier to understand and reuse if they are bound together behind a method name that defines the intent of the method. We'll also see when we talk about mock objects that

having finders called behind other methods makes it much easier to avoid touching the database when you don't need to.

If a finder is extracted during refactoring, and its functionality is already covered by an existing test, you may not need a new test to cover it. Like any other method extracted in refactoring, you aren't adding logic. Again, though, if the finder method winds up in a different class then that covered by the existing test, I might transfer the test logic.

If you are test-driving the finder method directly, you have two issues in tension. On the one hand, you need to create enough objects to feel confident that the finder method is being tested. On the other, ActiveRecord finder methods are tests that actually need to touch the database, which makes the test slow, so you want to create as few objects as possible. Don't shy away from creating ActiveRecord objects when you are legitimately testing database retrieval behavior, but don't create more objects than you need to, either.

If you are testing a method that finds objects based on criteria, start with a test that creates two objects. That's one object you expect to be found by the method and one that you do not, which allows you to cover the logic from both sides.

```
test "it finds completed tasks" do
  complete = Task.create(completed_at: 1.day.ago, title: "Completed")
  incomplete = Task.create(completed_at: nil, title: "Not Completed")
  assert_equal(["Completed"], Task.complete.map(&:title))
end
```

The test creates two objects and asserts that only one of them is found by the Task.complete method. The last line of the test does something that is perhaps a little idiosyncratic, but which I've found useful. Specifically, it converts the list of ActiveRecord objects (Task.complete) to a list of strings (map.(&:title)).

More generally, what I'm doing is converting a complex object to a simple one for the purposes of testing. The reason I do this is increased readability, to some extent in the test, but much more so in the test output. If this test fails as written, the output will look something like this:

```
  1) Failure:
TaskTest#test_it_finds_completed_tasks [/gatherer/test/models/task_test.rb:37]:
Expected: ["Completed"]
  Actual: []
```

Whereas if I had not converted the last line, the error would be more like:

```
TaskTest#test_it_finds_completed_tasks [/gatherer/test/models/task_test.rb:37]:
--- expected
```

```
+++ actual
@@ -1 +1 @@
-[#<Task id: 980190963, project_id: nil, title: "Completed",
size: nil, completed_at: "2013-12-14 21:47:22",
created_at: "2013-12-15 21:47:22",
updated_at: "2013-12-15 21:47:22">]
+[]
```

I submit that the first error message makes it easier to determine what's going on.

Once you have written your initial two-object test write another test if and only if you can think of another pair of objects that would fail given the current code. This would be the case if your finder had compound logic. If we were writing a method to find tasks that were both completed and large, we might start with a test that has one object with both criteria and one with neither and then write a second test with an object with both criteria and an object that only has one.

What we are trying to avoid is a combinatorial explosion where we create 16 objects to test a finder with four elements. Going two at a time, and only creating new pairs if there's still a potential failure keeps each test small and easy to understand.

If you are testing for sort logic, you have to work around the fact that the order in which you add data to the database in setup is the order in which you get the objects back when you don't explicitly specify an order (because the default order is by ID ascending). If you put your data in in the same order as the expected output, then the test passes even before you do anything. Which is bad.

If I need to test sort behavior, I create three objects, expected middle value first, excepted high value second, expected low value third. This pattern is out of order no matter which way we go, so a test for sorting will correctly fail until the logic is in place.

Modeling Data

These guidelines should give you some direction as you test-drive new business logic in your Rails application. Every test you write, however, depends on some data to run. Getting useful data into the test cleanly and quickly turns out to be kind of complicated. In the next chapter, we will discuss several ways of managing test data.

CHAPTER 7

Adding Data To Tests

Creating test data sounds like it should be the easiest thing ever. We've already got ActiveRecord#create, right? Not quite. In order to be useful, the data that you generate for your tests needs to support the goals of testing. You should be able to create the data quickly and easily, both in the amount of typing it takes to create data and the speed the test runs. The data should be the same every time you generate it, and should be specific to a set of tests, and the data should be an accurate representation of the objects that will be used when the code actually runs outside of tests.

Nothing against ActiveRecord, but if it is the only way you use to get data into your tests, you're going to have some problems. These problems include tests with a lot of extraneous details, slow tests, and tests that are brittle against future changes to the definition of your model.

In this chapter, we'll discuss two techniques that are in wide use in the Rails community for creating data. In the Rails framework itself, *fixtures* are used to rapidly create global data. As we'll see, fixtures solve some of the problems of creating test data, but cause different ones. Specifically, fixtures are fast and easy to use, but are global to all tests.

Over time, the Rails community has created a set of tools called *factory* tools, which use some variant on the Factory design pattern to create data. Factories have an overlapping, but slightly different set of strengths than fixtures, they are also easy to create, but can be slow to run.

Next chapter, we'll explore a completely different way to think about the inputs to your test when we talk about *mock objects*.

As with many testing decisions, there's no one answer that works for all situations, but a variety of tools with different strengths that can be used well or used poorly.

What's the Problem?

What's the big deal if I want to use normal, ordinary ActiveRecord#create in my tests? I use it in my code. What could go wrong?

Since you asked…

We'll start with a simple test involving two users.

```
test "I can tell which user is older" do
  eldest = User.create(date_of_birth: '1971-01-22')
  youngest = User.create(date_of_birth: '1973-08-31')
  assert_equal eldest, User.find_eldest
  assert_equal youngest, User.find_youngest
end
```

That test is deliberately simple, so as not to distract from the data creation issue. The only weird thing here is that we are testing a hypothetical finder method find_eldest that actually goes into the database, so it's necessary for the test that the objects actually make it all the way into the database.

You make the test pass and forget about the test, which silently continues to pass every time you run your test suite.

And then…

You add authentication. And even though this test has nothing to do with authentication, it fails. The first assertion will fail, instead of returning eldest, it will return nil.

The problem is that adding authentication adds two new validations, a requirement that a user must have an email address and a password. Our test data no longer matches those requirements, so the objects aren't saved to the database, so they can't be found by the finder methods, hence the nil.

With a heavy sigh, we add the required fields to the test:

```
test "I can tell which user is older" do
  eldest = User.create!(date_of_birth: '1971-01-22',
    email: "eldest@example.com", password: "password")
        youngest = User.create!(date_of_birth: '1973-08-31'
    email: "youngest@example.com", password: "password")
        assert_equal eldest, User.find_eldest
  assert_equal youngest, User.find_youngest
end
```

Okay… that's not horrible. It's not great, but life marches on. We've switched to create! so now at least any further validation will fail at the point of creation, which makes diagnosing the failure much easier.

Some time later, marketing insists on full demographic data for all your users, including height, zip code, and handedness. (Handedness? Sure, let's say your company makes golf clubs.) And the database guy insists that means that all demographic categories must be required in the database. Now the test looks like this:

```
test "I can tell which user is older" do
  eldest = User.create!(date_of_birth: '1971-01-22',
    email: "eldest@example.com", password: "password",
    height: 185, zip code: "60642", handedness: "left")
  youngest = User.create!(date_of_birth: '1973-08-31'
    email: "youngest@example.com", password: "password",
    height: 178, zip code: "60642", handedness: "ambidextrous")
  assert_equal eldest, User.find_eldest
  assert_equal youngest, User.find_youngest
end
```

This is starting to get out of hand. Now, not only do you need to type three lines of text in a test just to create a single user, it's also nearly impossible to pick out of this data the one attribute — date of birth — that is actually relevant for the test.

Not only that, but this problem happens every time we create a user in a test and every time we add a new validation to users. In other words, all the time.

What we'd like is a way to specify a known valid object in such a way that there is at most one place to update when new validations get created. Fixtures and factories are two different mechanisms for solving this problem.

Fixtures

Rails has always made it very easy to manage a database just for test data, which is automatically cleared between each test. (While there's no denying this is tremendously useful, it has also lulled all of us into feeling that a test that touches the database – a huge third-party dependency – is somehow a *unit* test.) One of the most valuable ways in which Ruby on Rails has supported automated testing is through the use of easily created data that is accessible to all the tests in your system, no matter when or where you write those tests, using *fixtures* specified in a YAML file. It's sometimes hard for an experienced Rails programmer to remember just how exciting the YAML fixtures used to seem. You can just set up data once? In an easy format? And it's always there? Amazing.

Over time, the infatuation with fixtures has dimmed a bit, but fixtures are still a quick and easy way to get data into your tests.

What's a fixture?

Generically, a fixture is any baseline state known to exist at the beginning of a test. The existence of a fixed state makes it possible to write tests that make assumptions based on that particular set of data. In Rails, the term "fixture" refers to a specific mechanism to easily define a set of objects that will exist globally for all tests. These fixtures are defined in a set of YAML files that are automatically written to the database and converted to ActiveRecord objects at the beginning of each test run.

Under normal circumstances, each ActiveRecord model in your application will have an associated fixture file. The fixture file is in YAML format, a data-description format often used as an easier-to-type alternative to XML. (YAML stands for Yet Another Markup Language. A full description of the syntax is available at http://www.yaml.org.) The details of YAML syntax are both way outside the scope of this book and largely irrelevant to fixtures. YAML contains a number of advanced features that don't concern us here.

Each model in your system has a fixture file named after the plural version of the model. So, if we wanted fixtures for our Projects and Task models, they would go in test/fixtures/projects.yml and test/fixtures/tasks.yml, respectively. If you use Rails generators to create your model, then a fixture file is created for you with some boilerplate values for each attribute.

Each entry in a fixture file starts with an identifier, with the attributes for that entry subordinate to the identifier. Here's a sample for Project:

```
runway:
  name: Project Runway
  due_date: 2013-12-18

book:
  name: Write the book
  due_date: 2014-04-14
```

YAML syntax is somewhat reminiscent of Python, both in the colon used to separate key/value pairs and in the use of indentation to mark the bounds of each entry. The fact that the line book: is outdented two spaces indicates to the YAML parser that a new entry has begun. Strings do not need to be enclosed in quotation marks, unless it's a string that the YAML parser would find ambiguous, such as if the string value also contains a colon and a space. It doesn't hurt to add the quotation marks if you find it more readable.

A multiline string can be specified by putting a pipe character (|) on the line with the attribute name. The multiline string can then be written over the

next set of lines; each line must be indented relative to the line with the attribute name. Once again, outdenting indicates the end of the string.

```
runway:
  name: Project Runway
  due_date: 2013-12-18
  description: |
    The awesomest project ever.
    It's really, really great.
```

The Rails fixture creation process uses information in your database to coerce the values to the proper type. I write dates in SQL format (yyyy-mm-dd), though any format readable by Ruby's Date.parse() will work.

The identifier that introduces each record is then used to access the individual fixture entry within your tests. Assuming that this is the Project class, you'd be able to retrieve these entries throughout your test suite as projects(:runway) and projects(:book), respectively. Unless you like trying to figure out what's special about projects(:project_10), I recommend meaningful entry names, especially for entries that expose special cases: projects(:project_with_no_due_date).

The YAML data is converted to a database record directly, without using ActiveRecord#new or ActiveRecord#create. (To be clear, when you use the data in your tests, the objects you get are ActiveRecord models — only the original creation of the data to the database bypasses ActiveRecord.) This means you can't use arbitrary methods of the model as attributes in the fixture the way you can in a create call.

Fixture attributes have to be either actual database columns or ActiveRecord associations explicitly defined in the model. Removing a database column from your model and forgetting to take it out of the fixtures results in your test suite erroring out when loaded. The fixture loading mechanism also bypasses any validations you have created on your ActiveRecord, meaning that there is no way to guarantee the validity of fixture data on load, short of explicitly testing each fixture yourself.

You do not need to specify the id for a fixture (although you can if you want). If you do not specify an id explicitly, the id is generated for you based on the YAML identifier name of the entry. If you allow Rails to generate these ids, then you get a side benefit: an easy way of specifying relationships between fixture objects. If your two model classes have an explicitly defined ActiveRecord relationship, then you can use the fixture identifier of one object to define the relationship in the other object. In this snippet from a potential tasks.yml,

I'm defining the task as having a relationship with the project defined as projects(:book)

```
chapter:
  title: Write a chapter
  project: book
```

If the relationship is has_many, the multiple values in the relationship can be specified as a comma-delimited list. This is true even if the two objects are in a has_and_belongs_to_many relationship via a join table, although a has_many :through relationship does need to have the join model entry explicitly specified.

Fixture files are also interpreted as ERb files, which means you can have dynamic attributes like this:

```
runway:
  name: Project Runway
  due_date: <%= 1.month.from_now %>
```

Or you can specify multiple entries dynamically, like this:

```
<% 10.times do |i| %>
task_<%=i%>:
  name: "Task <%= i %>"
<% end %>
```

In the second case, notice that the identifier still needs to be at the leftmost column; you can't indent the inside of the block the way that normal Ruby style would suggest. Also, don't do this, it gets super-confusing really quickly. If you find yourself needing dynamic data functionality like this, you are probably better off with a factory tool.

Loading Fixture Data

By default, all your defined fixtures are loaded into the database once at the beginning of your test run. Rails starts a database transaction at the beginning of each test. And the end of each individual test, the transaction is rolled back, restoring the initial state very quickly.

You can control fixture loading with a few parameters which are set in the test/test_helper.rb file. The most important is the fixtures :all method call, which is the line that ensures that all your fixture files are loaded in all your tests. The existence of this line actually being written in the file is a little bit of framework archeology. The original default in Rails was to only load fixtures for the class under test. You can still specify specific fixtures to be loaded by passing the model names as symbols to the fixtures method, though I'm not sure there's a good reason for doing so these days.

The transactional behavior of fixtures is a problem if you are actually trying to test transactional behavior in your application. In that case the fixture transaction will overwhelm the transaction you are trying to test. If you need less aggressive transaction behavior, you can go into the test/test_helper.rb file and add the line self.use_transactional_fixtures = false. That will change the value for all tests, but you can also override the value on a class-by-class basis by including the assignment (set to false) in your individual class. There's no way to change this behavior to be fine-grained enough to use the nontransactional behavior for only a single method. Again, if you need to test transactional behavior, fixtures may not be your best bet.

Why Fixtures are Great

Fixtures are Fast

One reason to use fixtures in a modern Rails application is how fast they are. Fixtures only add overhead when the Rails framework is loaded, there's no real cost to having fixtures persist between tests. So there's no particular downside to having a lot of objects defined in fixture files.

Well, there's one downside, which is sometimes you might write a test that assumes the database table is blank, so you'd test something that's supposed to create an object, and then test that there's exactly one object in the database. The existence of fixture data will break that test, because you'll start with objects in the database. One workaround is to explicitly test the change between before and after, rather than assuming the before value is zero.

Fixture speed makes fixtures ideal for setting up reasonably complicated object relationship trees that you might be using in your tests. That said, if you are truly unit testing, it's likely you don't need complicated object relationship trees. (If you are acceptance testing, on the other hand… Hold that thought.)

Fixtures are Always There

You can count on fixtures always being available anywhere in your test suite. In a lot of unit testing situations that's not really a big deal, because you don't create a lot of data for each test.

However, some applications rely on the existence of some kind of mostly-static data that's stored in the database. Often this is some kind of meta-data, product types or user types. It's stored in the database to make it easy to modify, but most of the time it's basically static data. If your application

assumes that kind of data will always be there, setting that data up via fixtures will be faster and easier than re-creating the data for each test.

Prescription 10	Fixtures are particularly useful for global semi-static data that happens to be stored in the database.

Why Fixtures are a Pain

As great as fixtures are when you are starting out, using them long-term on complex projects exposes problems. Here are some things to keep an eye on.

Fixtures are Global

There is only one set of fixtures in a default Rails application. So, the temptation to keep adding new data points to the fixture set every time you need a corner case is pretty much overwhelming. The problem is that every time you add a user because you need to test what happens when a left-handed user sends a message to another user with a friend relationship who happens to live in Fiji, or whatever oddball scenario you need, every other test has to deal with that data point being part of the test data.

Fixtures are Spread Out

Fixtures live in their own directory, and each model has its own fixture file. That's fine, until you start needing to manage connections and a simple setup of a user commenting on a post related to a given article quickly spans across four different fixture files, with no easy way to trace the relationships. I'm a big fan of "small and plentiful" over "large and few" when it comes to code structure, but even I find fixtures too spread out.

Fixtures are Distant

If you are doing a complex test based on the specific fixture lineup, you'll often wind up with the end data being based on the fixture setup in such a way that, when reading the test, it's not clear exactly how the final value is derived. You need to go back to the fixture files to understand the calculation.

Fixtures are Brittle

Of course, once you add that left-handed user to your fixture set, you're guaranteed to break any test that depends on the exact makeup of the entire user population. Tests for searching and reporting are notorious culprits here. There aren't many more effective ways to kill your team's enthusiasm for testing like having to fix twenty-five tests on the other side of the project every time you add new sample data.

Sounds grim, right? It's not. Not only are fixtures perfectly suitable for simple projects, the Rails community has responded to the weaknesses of fixtures by creating factory tools that can replace fixtures in creating test data.

Factories

Generically, the *factory* pattern refers to a class or module in your application whose sole purpose in life is to safely and correctly create other objects in your application. Outside of tests, factories are frequently used to encapsulate complex object-creation logic. Inside of Rails tests, factories are used to provide templates for creating valid objects.

Rather than specifying all the test data exactly, the factory tool provides a blueprint for creating a sample instance of your model. When you need data for a specific test, you call a factory method, which gives you an object based on your blueprint. You can override the blueprint to specify any specific attribute values required to make your test work out. Calling the factory method is simple enough to make it feasible to set up a useful amount of data in each test.

The most common factory tool used for Rails testing is factory_girl, which lives at https://github.com/thoughtbot/factory_girl and https://github.com/thoughtbot/factory_girl_rails. The current version is 4.3.0.

Let's talk first about how to set up and use factory_girl, and once we have the basics down, we'll talk about how to use factory_girl effectively.

Installing factory_girl

To install factory_girl in a Rails project, just include the following in the Gemfile:

```
gem 'factory_girl_rails'
```

In a Rails project, factory files are automatically loaded if they are in test/factories.rb or test/factories/*.rb (in RSpec land, that would be spec/factories.rb or spec/factories/*.rb). Factories defined any other place need to be explicitly required into the program.

There's one optional configuration, which is to place the following line inside the class definition in test_helper.rb:

```
include FactoryGirl::Syntax::Methods
```

If enabled, you can use the factory_girl creation methods without the FactoryGirl prefix — this will make more sense in a little bit. Personally, I avoid using this line, I'm used to the more verbose syntax.

Basic Factory Definition

All the definitions of your factories go inside a call to the method FactoryGirl.define, which takes a block argument. Inside that block, factories can be declared to define default data. Each factory declaration takes its own block argument in which you can define default values on an attribute-by-attribute basis.

A very simple example for our task builder might look like this:

```
FactoryGirl.define do
  factory :project do
    name "Project Runway"
    due_date Date.parse("2014-01-12")
  end
end
```

Note the absence of equals signs — these are not assignments. Technically, they are method calls, so if it makes it more readable to write the lines like name("Project Runway"), go for it.

Factory_girl assumes that there is an ActiveRecord class with the same name as the one you give to the factory. When the factory is invoked, the resulting object is of that class.

If you want to have the factory refer to an arbitrary class, you can specify the class when the factory is defined:

```
FactoryGirl.define do
  factory :big_project, class: Project do
    name: "Big Project"
  end
end
```

In the previous factory, all the values are static and are determined when the factory file is loaded. If you want a dynamic value to be determined when an individual factory object is created, just pass a block instead of a value; the block will be evaluated when each new factory is called.

```
FactoryGirl.define do
  factory :project do
    name "Project Runway"
    due_date { Date.today - rand(50) }
  end
end
```

You can also refer to a previously assigned value later in the factory.

```
FactoryGirl.define do
  factory :project do
    name "Project Runway"
    due_date { Date.today - rand(50) }
    slug { "#{name.downcase.gsub!(" ", "_")}" }
  end
end
```

What's nice about this is that the factory will always use the value in the name attribute to calculate the URL, even if you pass the name in yourself rather than use the default value. So this factory could be used as:

```
test "factory girl slug block" do
  project = FactoryGirl.create(:project, name: "Book To Write")
  assert_equal("book_to_write", project.url)
end
```

If you used the include FactoryGirl::Syntax::Methods call alluded to above, then you could write the first line as just project = create(:project, name: "Book To Write"). I prefer the explicit reminder that FactoryGirl is being used.

Inside the factory, you can call any attribute in the model that has a setter method; in other words, unlike fixtures, any virtual attribute in the model (like the password attribute of a Devise User model) is fair game.

Basic Factory Creation

Factory_girl provides four different ways of turning a factory into a Ruby object. Given the project factory we were just looking at, the four ways are:

- build(:project), which returns a model instance that has not been saved to the database.
- create(:project), which returns a model instance and saves it to the database.
- attributes_for(:project), which returns a hash of all the attributes in the factory, suitable for passing to ActiveRecord#new or ActiveRecord#create. This method is most often useful for creating a hash that will be sent as params to a controller test.
- build_stubbed(:project), is almost magical. Like build, it returns an unsaved model object. Unlike build, it assigns a fake ActiveRecord id to the model and stubs out database interaction methods (like save) such that the test raises an exception if they are called.

All four of the build strategy methods allow you to pass key/value pairs to override the factory value for a given attribute:

```
project = FactoryGirl.build_stubbed(:project, name: "New Project")
```

```
assert_equal("New Project", project.name)
```

I consider it useful to explicitly list any attribute whose value is essential to the test when building the object from the factory. I will do this even if the attribute has the same value as the factory default, because I like having the explicit value in the test where it can easily be seen.

All the build strategy methods will also yield the new object to a block if you wanted to do some more custom processing.

```
project = FactoryGirl.build_stubbed(:project) do |p|
  p.tasks << FactoryGirl.build_stubbed(:task)
end
```

In practice, I don't use this form very much, though I think it would be helpful if you have additional creation logic on a new test object.

I have a very simple set of strategies to determine which of these methods to use.

- Only use attribute_for in the specialized case of needing a valid set of hash attributes, again, in my experience that's most likely in a controller test.
- Only use create if the object absolutely must be in the database. Typically, this is because the test code must be able to access it via an ActiveRecord finder. However, create is much slower than any of the other methods, so it's also worth thinking about whether there's a way to structure the code so that persistent data is not needed for the test.
- In all other cases, use build_stubbed, which does everything build does, plus more. Because a build_stubbed object has a Rails id, you can build up real Rails associations, and still not have to take the speed hit of saving to the database.

Prescription 11 Your go-to build strategy for factory_girl should be 'build_stubbed' unless there is a need for the object to be in the database during the test.

If you need to create a set of objects together, all the build strategies have two special forms, *_pair and *_list. The pair methods, like create_pair(:project) or build_stubbed_pair(:project) create exactly two objects of the given factory. You can still pass key/value pairs to the method, in which case the attribute overrides are applied to both the objects. The list methods create an arbitrary amount of items, denoted by an integer argument after the name of the factory, as in create_list(:project, 5), which creates 5 projects. As with the pair methods, key/values can be passed in and are applied to the entire list.

Associations and Factories

Factory_girl has a powerful set of features for adding associations to factories. We're going to talk about them because they are powerful, and you might see code that uses them. Then we are going to talk about why you should be careful to actually use them.

The simplest case is also a common one: the class being created has a belongs_to association with the same name as a factory. In that case, you just include that name in the factory.

```
FactoryGirl.define do
  factory :task do
    title: "To Something"
    size: 3
    project
  end
end
```

In this case, calling task = FactoryGirl.create(:task) would also implicitly call task.project = FactoryGirl.create(:project).

If you want to explicitly specify the project when calling the task factory, you can do so in just the same way you would for any other attribute, namely task = FactoryGirl.create(:task, project: Project.new). If the association is specified in the factory definition, but you don't want any value in the test, then you need to set the association to nil, as in: task = FactoryGirl.create(:task, project: nil).

If the association name doesn't match the factory name, or if you want to specify default attributes of the associated object, you can use the longer form of the association statement in the factory definition:

```
FactoryGirl.define do
  factory :task do
    title: "To Something"
    size: 3
    project
    association :doer, factory: :user, name: "Task Doer"
  end
end
```

The syntax is association, followed by the name of the association in the ActiveRecord model, and then a bunch of key/value pairs, with the factory key being used by FactoryGirl to determine which factory to use to create the associated object.

Where this gets tricky is in the build strategy used for the subordinate object. As you might expect, calling the parent factory—in this case Task—with create

causes the associated factory—in this case Project—to also be instantiated using create.

By default, however, even if you call the parent factory with build, the subordinate factory is still called with create. This is a side effect of how Rails manages associations. The associated object needs an id so that the parent object can link to it, and in Rails you only get an ActiveRecord id when you are saved to the database.

As a result even if you use the build build strategy specifically to avoid slow and unnecessary database interaction, if the factory has associations, you will still save objects to the database. Since those associated factories may themselves have associations, if you aren't careful you can end up creating a lot of objects to the database resulting in prohibitively slow tests.

It's exactly this characteristic of factory_girl that has made it unwelcome in some circles. Particularly if the people in those circles have to maintain a large, unwieldy test suite. I know of one Rails example where a test suite that had become so slow that the entire suite was only run on dedicated servers and took over 45 minutes. Factory association misuse wasn't the only reason for the slowness, but a significant number of the slowest individual tests in the system built up way, way more objects then they needed to because of factory_girl associations.

There are a few ways to avoid unnecessary database creation of associated objects.

If you use the longer form of specifying the association, you can explicitly specify the build strategy.

```
FactoryGirl.define do
  factory :task do
    title: "To Something"
    size: 3
    project
    association :doer, factory: :user, strategy: :build
  end
end
```

If you go this route, you may have problems because the associated object won't have an id. In this specific case, for example, the Task object will have it's user attribute set, but not its user_id. If your code is expecting the user_id to be set, for example because user_id is faster to access than user, this may cause problems in your tests.

Since the build_stubbed strategy assigns an ID to the objects being created, using build_stubbed sidesteps the whole issue. If a factory with associations is instantiated using build_stubbed, then by default all the associations are also invoked using build_stubbed. Which solves the problem, as long as you always use build_stubbed.

My preferred strategy in my own code is to not specify attributes in factories at all, and if I need associated objects in a specific test, explicitly add them to the test only at the point needed.

Why?

- The surest way to keep factory_girl from creating large trees of objects is not to define large trees of objects.
- Tests that require multiple degrees of associated objects often indicate improperly factored code. Making it a little harder to write associations in tests nudges me in the direction of code that can be tested without associations.

  Avoid defining associations automatically in factory_girl definitions, set them test by test, as needed. You'll wind up with more manageable test data.

The only downside is that there's a little more typing involved in some tests.

DRY Factories

Once you have more than a couple of factories in your application, you want to make sure that you can manage complexity and duplication. Factory_girl has a number of features to allow you to do just that.

Sequences

A common problem is the creation of multiple objects that require unique values. This most often happens with unique user attributes like a login or email address. To allow the easy creation of unique attributes, factory_girl allows you to define an attribute as part of a sequence of values.

The short version of the syntax looks like this:

```
FactoryGirl.define do
  factory :task do
    sequence(:title) { |n| "Task #{n}" }
  end
end
```

Calling sequence inside a factory takes one argument, which is the attribute whose values are being sequenced, and a block. When the factory is invoked, the block is called with a new value, and the return value of the block is set

to be the value of the attribute. The start value is 1 be default, but a second argument to sequence can be used to specify an arbitrary value, which is usually a number, but which can be any object that responds to next.

When a sequence is defined inside a factory, it can only be used in that one place, however sequences can also be defined outside of factories and reused:

```
FactoryGirl.define do
  sequence :email do |n|
    "user_#{n}@test.com"
  end

  factory :user do
    name "Fred Flintstone"
    email
  end
end
```

The use of email inside the factory is a shortcut which assumes that the sequence and the attribute have the same name. If so, the sequence is triggered and the next value becomes the value of the attributes. If the sequence and the attribute have different names, then you need to invoke the sequence explicitly by calling generate inside an attribute's block.

```
factory :task do
  title "Finish Chapter"
  user_email { generate(:email) }
end
```

Inherited Factories

Often, you'll need to create multiple factories from the same class — a classic example is the ability to create different kinds of users such as regular users versus administrators.

The most direct way to create slightly different factories in the same class is via factory_girl's inheritance feature. If you define a factory as having another factory as a parent, it takes all the attributes set in that parent but then allows you to override in the child factory. Effectively, factory_girl inheritance allows you to group common attributes so they can be reused:

```
FactoryGirl.define do
  factory :task do
    sequence(:title) { |n| "Task #{n}" }
  end

  factory :big_task, parent: :task do
    size 5
  end
```

```
  factory :small_task, parent: :task do
    size 1
  end
end
```

In the above set of factories, both big_task and small_task share the sequence being used to generate unique titles, but they each define their own size value.

In addition to explicitly setting the parent, you can achieve the same affect by nesting the child factories inside the parent definition, like so:

```
FactoryGirl.define do
  factory :task do
    sequence(:title) { |n| "Task #{n}" }

    factory :big_task do
      size 5
    end

    factory :small_task do
      size 1
    end
  end
end
```

If actually creating a parent factory and child factories seems a little backwards to you, factory_girl also allows you to group a set of common attributes into a single chunk, called a *trait*, which can then be used inside other factories. This makes sense if you have groups of attributes that have values that are basically orthogonal to each other.

Traits can be used as though they were single attributes. Creating a non-contrived example is tricky here without making our sample classes much more complex then they currently are — just realize that each trait could hold multiple attributes here.

```
FactoryGirl.define do
  factory :task do
    sequence(:title) { |n| "Task #{n}" }

    trait :small do
      size 1
    end

    trait :large do
      size 5
    end

    trait :soon do
```

```
      due_date { 1.day.from_now }
    end

    trait :later do
      due_date { 1.month.from_now }
    end

    factory :trivial do
      small
      later
    end

    factory :panic do
      large
      soon
    end
  end
end
```

The last couple of factories can be written slightly differently:

```
factory :trivial, traits: [:small, :later]
factory :panic, traits: [:large, :soon]
```

Again, having just one attribute per trait doesn't show the feature in its best light, but you have the basic trade off of verbose versus succinct. Traits take some extra definition, but give meaningful names to groups of attributes that you might reuse. The good side is the meaningful name, the downside is the extra typing and added complexity of the factory.

As factories get even more complex, factory_girl offers a few other techniques to manage them, including the ability to have post-creation callbacks, create custom build strategies, and have dummy attributes that are only used to control the factory creation. I suspect that these are only useful in somewhat specialized cases, so I'm not going to go into them in more detail, the factory_girl documentation at https://github.com/thoughtbot/factory_girl/blob/master/GETTING_STARTED.md can help you if you are curious.

Preventing Factory Abuse

The initial temptation when using fixtures is to continue to build large trees of objects — this is particularly true if you are converting a project that was using fixtures. The best way to use factories is to create as little data as possible for each test. Create only the smallest amount of data needed to expose the issue in each test. This practice speeds up the test, makes the issue easy to see rather than burying it among dozens of fixtures, and makes the correctness of the test itself easier to verify.

Dates and Times

Date and time logic has a well-deserved reputation as one of the most complex and irritating logic in any application. Testing calendar logic, including time-based reports, automatic logouts, and "1 day ago" text displays, can be a particular headache, but there are a couple of things you can do to simplify the time logic beast.

Part of the Problem

We've got a YAML file with some projects:

```
runway:
  name: Project Runway
  start_date: 2015-01-20

greenlight:
  name: Project Greenlight
  start_date: 2015-02-04

gutenberg:
  name: Project Gutenberg
  start_date: 2015-01-31
```

We'd like to test some time-based code, that might be used in a search or report result; this goes in test/unit/project_test.rb:

```
test "finding recently started projects" do
  actual = Project.find_recently_started(6.months)
  assert_equal(3, actual.size)
end
```

Here's code that makes the test pass, from app/models/project.rb:

```
def self.find_recently_started(time_span)
  old_time = Date.today - time_span
  all(conditions: ["start_date > ?", old_time.to_s(:db)])
end
```

On January 20, 2015, the test passes. And on the 21st it will pass, and the day after.

Six months later, though, about June 20th. We've probably long forgotten about this test, this sample data, and maybe even this entire project. Suddenly, the test fails. And we'll spend way too much time trying to figure out what happened, until we remember the date issue and realize that the January 20th sample has moved out of the six-month time span specified in the test.

Of course, changing all the dates just pushes the problem forward and gives us time to forget all about it again.

This issue may sound silly to some, but like many of the more ridiculous examples in the book, this is a mistake that happened to me and can end up costing a lot of time. I've seen tests that fail at a particular time of day (because time zones used inconsistently push part of the code into the next day). I've seen tests that pass in Chicago but fail in California. I've seen tests that fail on the first day of a new month and a new year. And, of course, tests that fail on the boundary in and out of daylight savings time. Most of this can be prevented.

When I was young and foolish and got paid to write Java, I solved this problem by adding an optional argument to just about every method that used the equivalent `Date.today`, allowing an optional time to be passed to the method and allowing an explicit date to be used for testing. This is a lot of work (although, interestingly, I think the Ruby community is coming back to allowing this kind of optional argument injection as a regular practice). Here are a few other options for dealing with date and time data.

Use Relative Dates

Using relative dates in your test data is often a way to work around date and time weirdness. This can be done with fixtures or factories, or just the objects you create in your tests.

Since Rails fixture files are evaluated as ERb files before loading, you can specify dynamic dates:

```
runway:
  name: Project Runway
  start_date: <%= 1.month.ago %>

greenlight:
  name: Project Greenlight
  start_date: <%= 1.week.ago %>

gutenberg:
  name: Project Gutenberg
  start_date: <%= 1.day.ago %>
```

With fixtures written like this, the previous test will always work, since the `start_date` of the projects will never fall out of the six-month range.

You can do something similar in factory_girl:

```
factory :project do
```

```
  name "Project Runway"
  start_date { 1.week.ago }
end
```

Although this technique works quite well for keeping test data a consistent relative distance from the test time, it's less helpful if you are actually trying to test based on the exact value of one of the dates. For example, if you are testing an output value or format. With the first, static set of fixture data, you could write the following:

```
test "that project dates are displayed in this goofy format" do
  assert_equal("2010 1 January", projects(:runway).goofy_start_date)
end
```

This test is a lot more difficult to write if you don't explicitly know the value of the project's start_date.

Stub Time

Another option for managing date logic in tests is to "freeze" time by using a stub explicitly specify what time Ruby reports why you ask for the current time. When I do this, I use the Timecop gem[1]. Timecop can be placed in your Bundler Gemfile with the traditional gem "timecop".

Timecop is essentially a super-specific mock object package: it stubs out Date.today, DateTime.now, and Time.now, allowing you to explicitly set the effective date for your tests. Using Timecop, the original test could be rewritten as follows:

```
test "finding recently started projects" do
  Timecop.freeze(Date.parse("2015-02-10"))
  actual = Project.find_recently_started(6.months)
  assert_equal(3, actual.size)
end
```

The Timecop.freeze method stubs the current date and time methods to the date passed as the argument: in this case, February 10, 2015. Time does not move for the duration of the test. A separate method, Timecop.travel, resets the time but lets the system time move forward from that point onward.

Why both options? It's because keeping time constant for the life of a test makes the test environment more consistent. But sometimes, it is necessary for time to move forward, so Timecop offers both options. Along those lines, it's sometimes useful to put the following line in a setup method:

```
Timecop.freeze(Date.today)
```

1. https://github.com/travisjeffery/timecop

And then the other half in the teardown method:

```
Timecop.return
```

This ensures that the current time doesn't change for the duration of each test. Again, with certain kinds of timing-related issues, that consistency eliminates a possible source of intermittent test failures or just plain confusion.

The argument to freeze or travel is an instance of Date, DateTime, Time, or a series of arguments of the form (year, month, day, hour=0, minute=0, second=0). Both methods also take blocks such that the fake time is good only for the duration of the block:

```
test "reports based on start date" do
  Timecop.freeze(Date.parse("2015-02-10")) do
    actual = Project.find_started_in_last(6.months)
    assert_equal(3, actual.size)
  end
end
```

The time travel methods can be in your setup or in an individual test. You can also change the time in the middle of a test to speed up an ongoing process:

```
test "is the project over" do
  p = Project.new(:start_date => Date.today,
      :end_date = Date.today + 8.weeks)
  assert(!p.complete?)
  Timecop.freeze(Date.today + 10.weeks)
  assert(p.complete?)
end
```

Timecop lets you keep explicit dates in your test data without causing problems later. The only downside is that if you have many tests setting time to different days, it can get somewhat confusing in the aggregate. It's easier if you use the same start date consistently. (On a solo project, you might use your birthday, for instance, but that's probably overly cute for a team project.) A more minor problem is that the line at the end of your test runs that says how long the test suite took will be hopelessly messed up because of the continued messing with Time.now.

Comparing Time

Ruby, not content with a simple date and time system, has three separate classes that manage date and time data. The Time class is a thin wrapper around the same Unix C library that pretty much every language exposes. (Motto: "Annoying programmers since 1983!") There are also the Ruby-specific

classes Date and DateTime, which are more flexible and have a more coherent API but are slower.

For Rails testing purposes, the relevant points are that ActiveRecord uses Date and DateTime, depending on the specifics of the underlying database column; comparing a Date to a DateTime instance will always fail (as will trying to add or subtract them), and most of the Rails ActiveSupport methods (such as 5.days.ago) return DateTime.

In testing, this can lead to a lot of annoying failures, especially when you have a Date column with no time information — which is recommended if the time is not important. In general, it's a good idea to compare dates and times by converting them using to_s(:db). It avoids the irritating question of object equality, and you get more readable tests and error messages. When the exact time of the time object is in question, try to force the issue by using the Rails ActiveSupport methods to_date, to_time, and to_datetime. At worst, this means something like 5.days.ago.to_ date.to_s(:db), which may read a touch on the awkward side but is a robust test with a decent error message on failure.

Setting Rails Timestamps

One trick worth mentioning when testing dates is explicitly setting the created_at attribute of your ActiveRecord model. Normally, created_at is a timestamp automatically generated by Rails, and it's often used for the kind of time-based reporting alluded to in the rest of this section. Since it's automatically created at the current time, you can get into some weird situations if other dates are specified in the past. Even without that complication, you may still need to explicitly set created_at to use the attribute to test time-based features.

You can set created_at in the fixture file, just like any other attribute, or it can be specified in ActiveRecord::create or ActiveRecord::new, specified in a factory blueprint, or just plain reset with an assignment or update method.

Setting updated_at is trickier. Under normal circumstances, if you try to explicitly set updated_at, Rails will just automatically reset it on save, which completely defeats the purpose. To change this behavior, set the class variable for your class, with code like Project.record_timestamps = false, using the name of your class as the receiver of the message, sometime before you save the object with modified update time. After the save, reset things to normal with Project.record_timestamps = true.

Summary

To sum up, Rails provides fixtures as an exceptionally simple way to create a set of test data that can be shared across multiple tests. However, fixtures are so simple that they tend to not be adaptable to more complex product needs. Factory tools, which take a little bit more initial setup, allow for more flexibility in use at some cost in test performance. The two structures don't have to be mutually exclusive. One pattern for combining them is to create exactly one complex scenario in fixtures for use in integration or complex controller tests and to use factories for unit tests or simpler controller tests.

Fixtures and factory tools allow you to get test data into your database in order to create a known baseline for testing. However, in some cases, you may not want to actually place data in the database. Using the database from a test may be undesirable for performance reasons, for philosophical reasons (some people don't consider it reasonable to touch the database in a "unit" test), or where logistical reasons make objects hard to create. In the next chapter, we'll explore mock objects, which allow tests to proceed by faking not the data but rather the actual method calls that produce the data.

CHAPTER 8

Using Mock Objects

We have a problem. We want to add credit card processing to our project application so that we can make money. Testing the credit card functionality presents immediate difficulties. For one thing, we don't want to accidentally make a credit card purchase during testing — that would be bad. But even if the purchase gateway provides a test sandbox, we still don't want to depend on it for our unit tests to run. That network call is slow, and we don't want our passing tests to depend on the status of a remote server.

We have a problem. We'd like to build our code in a very modular kind of way. In doing so, we'd like our tests to be as isolated as possible from dependencies on other parts of the code. For example, we might have controller logic that calls a model, but which we want to test without depending on the model. We want our controller test to work even if the model is broken — even if the model code does not yet exist.

The solution to both of these problems is a *mock object*, sometimes referred to as a *test double*.

A mock object is a "fake" object used in place of a "real" object for the purposes of automated testing. A mock might be used when the real object is unavailable or difficult to access from a test environment. Or, you might use a mock to create a specific application state that would be otherwise difficult to trigger in a test environment, like a database or network failure.

Mocks can also be used strategically to limit the execution of a test to the object and method specifically under test. Used in that manner, mocks drive a different style of testing, where the test is verifying the behavior of the system during the test, rather than the state of the system at the end of the test.

Mocks can be a bit of a contentious issue, with different people giving conflicting advice about the best way to use mocks when testing. I'd like to give you

enough information to make informed decisions about who to agree with. (Though I hope you'll agree with me…) We're going to start by discussing the mechanics of mock objects, using the Mocha library. Then we will discuss different ways to use mock objects to solve both of these testing issues. Later, Chapter 10, *Driving Behavior With RSpec*, on page 149 will cover the basics of the RSpec mocking library.

Mock Objects Defined

One complicating factor in dealing with mock objects is that pretty much everybody who creates a mock framework feels perfectly free to use slightly different naming conventions than everybody else. Here are the names that I use, which are—of course—also the correct ones. (Actually, I believe this naming structure is the creation of Gerard Meszaros in *xUnit Test Patterns [Mes07]*.)

The generic term for any object used as a stand-in for another object is test double, by analogy to "stunt double," and with the same connotation of a cheaper or more focused replacement for a more expensive real object. Colloquially, *mock object* is also used as the generic term but, confusingly, is also the name of a specific type of test double.

A *stub* is a fake object that returns a predetermined value for a method call without calling the actual method on an actual object. In the Mocha library, we create a stub as follows:

```
thing.stubs(:name).returns("Fred")
```

That line of code says that if you call `thing.name`, you'll get Fred as a result. Crucially, the actual `thing.name` method is not touched, so whatever value the "real" method would return is not relevant; the Fred response comes from the stub, not the actual object. If `thing.name` is not called in the test, nothing happens.

A *mock* is similar to a stub, but in addition to returning the fake value, a mock object also sets a testable expectation that the method being replaced will actually be called in the test. If the method is not called, the mock object triggers a test failure. You can write the following snippet to create a mock object instead of a stub, using `expects` instead of `stubs`:

```
thing.expects(:name).returns("Fred")
```

If you use the mock, then if you call `thing.name` in your test, you still get `Fred` and the actual `thing.name` method is still untouched. But if you don't call `thing.name` in the test, the test fails with what's Mocha calls an `ExpectationError`.

In other words, setting a stub on a method is passive and just says, "Ignore the real implementation of this method and return this value," while setting a mock on a method is aggressive and says, "This method will return this value, and you better call the method, or else!"

The reason you might set an expectation on whether a method is called is that once you've stubbed the method, it makes no sense to write an assertion on it like this one:

```
thing.stubs(:name).returns("Fred")
assert_equal("Fred", thing.name)
```

In this case, you're just testing that the stub works as advertised—this test can't fail. But if you do this:

```
thing.expects(:name).returns("Fred")
```

The above test requires the code to behave a certain way—namely, it has to call `thing.name` in order to pass the test.

There is a third test double pattern, called a *spy*. A spy is often declared like a stub, but allows you to specify a testable expectation later in the test. For example, you can use the Bourne library to add spies to Mocha using the following syntax.

```
thing.stubs(:name).returns("Fred")
assert_received(thing, :name)
```

In this test, the first line defines the stub, and the last line sets the expectation. Typically, we would place the body of the test in between. Using spies mitigates against a common criticism of mock object testing, which is that it can be difficult to look at a mock test and see exactly what behavior is being tested for. The spy explicitly declares the behavior that is expected. Spies are also more consistent with the Given/When/Then test structure we've used elsewhere, allowing the stub to be declared the the Given section, and the expectation to be set separately in the Then part of the test.

Installing Mocha

We're going to use the Mocha library for Ruby mocks (not to be confused with the JavaScript test framework of the same name). Mocha is used to test Rails itself, giving it quasi-official status, and it also is a more complete mock library that the library that is a part of Minitest, so it better enables us to discuss different types of mock testing. In Chapter 10, *Driving Behavior With RSpec*, on page 149, we will discuss RSpec's mock framework, which can also be used with Minitest.

Mocha needs to be installed after Minitest, which requires a slight indirection to ensure your Rails app does the right thing.

First, in the Gemfile, in the :test group, add the Mocha gem:

```
gem "mocha", require: false
```

The require: false ensures that Mocha loads in the correct order.

Then, inside the test/test_helper.rb file, you manually require Mocha, at any point in the file after the rails/test_help library is loaded:

```
require "mocha/mini_test"
```

At that point, we should be good to go.

Creating Stubs in Mocha

A stub is a replacement for all or part of an object that prevents a normal method call from happening and instead returns a value that is pre-set when the stub is created. In Mocha, as in many Ruby mock libraries, there are two kinds of fake objects. You can create entire objects that exist only to be stubs, which we'll call *full doubles*, or you can stub specific methods of existing objects, which we will call *partial doubles*.

A partial double is useful when you want to use a "real" ActiveRecord object, but have one or two dangerous or expensive methods you want to bypass. A full double is useful when you are testing that your code works with a specific API rather than a specific object—by passing in a generic object that only responds to certain methods, you make it hard for the code to assume anything about the structure of the objects being coordinated with.

Prescription 12	Use partial doubles when you want to ensure most of your real object behavior. Use full doubles when the behavior of the stubbed object doesn't matter, only its public interface.

In Mocha, you can create a full double that is just a set of stubbed methods by calling stub, which is available throughout your test cases. Since Ruby uses duck typing and therefore cares only whether objects respond to the messages sent to them, a stub object created in such a way can be inserted into your application as a replacement for a real object.

```
test "here's a sample stub" do
  stubby = stub(name: "Paul", weight: 100)
  assert_equal("Paul", stubby.name)
end
```

The stub method takes a hash argument, the keys being messages the stub will respond to, the values being the return values of those messages. The assertion in the second line of the snippet is true because the stub has been preset to respond to the name message with "Paul."

If you call the stub with a method that is not in the hash argument, Mocha will return an error. If that's not the behavior you want, Mocha provides the stub_everything method, which instead returns nil for methods not in the hash argument. Using stub_everything makes sense in the case where there are a large number of potential methods to be stubbed, but where the values make so little difference that specifying them reduces the readability of the test.

This test is a very bad way to use stubs; I've set up a nice little tautology, and I haven't actually learned anything about any larger system around this test.

Mocha has one other trick for full doubles. Often you want to verify that the double behaves exactly like an existing object, which is to say, that you want it to throw a NoMethodError if you call a method that is not defined on the object being impersonated. In Mocha you can trigger this behavior with responds_like and responds_like_instance_of, which you might use as follows:

```
stubby = stub(size: 3)
stubby.respond_like(Task.new)
```

Or

```
stubby = stub(size: 3)
stubby.respond_like_instance_of(Task)
```

In either case, if you called stubby with a method that is not defined for the Task class, say stubby.due_date, then Mocha will raise a NoMethodError. Internally, this uses the Ruby respond_to? method, so if you are using dynamically generated methods via define_method, method_missing, or something more esoteric, you need to ensure that your respond_to? method works in sync.

You might use a full double object to stand in for an entire object that is unavailable or prohibitively expensive to create or call in the test environment. In Ruby, though, you would more often take advantage of the way Ruby allows you to open up existing classes and objects for the purposes of adding or overriding methods. It's easy to take a "real" object and stub out only the methods that you need. This is extraordinarily useful when it comes to actual uses of stub objects.

In Mocha, this is managed with the stubs method, which is mixed in to any Ruby Object.

mocks/01/gatherer/test/models/project_test.rb

```
Line 1 test "lets stub an object" do
   2   project = Project.new(name: "Project Greenlight")
   3   project.stubs(:name)
   4   assert_nil(project.name)
   5 end
```

This test passes: line 3 sets up the stub, and the project.name call in line 4 is intercepted by the stub to return nil and never even gets to the actual project name.

Having a stub that always returns nil is a little pointless, so Mocha allows you to specify a return value for the stubbed method using the following syntax:

mocks/01/gatherer/test/models/project_test.rb

```
Line 1 test "lets stub an object again" do
   2   project = Project.new(:name => "Project Greenlight")
   3   project.stubs(:name).returns("Fred")
   4   assert_equal("Fred", project.name)
   5 end
```

Line 3 is doing the heavy lifting here, tying the return value Fred to the method :name. The stubs method returns a Mocha Expectation object, which is effectively a proxy to the real object, but which responds to a number of methods that let you annotate the stub. The returns method is one of those annotation messages which associates the return value with the method. Mocha also provides the shortcut, project.stubs(name: "Fred"), allowing you to specify messages and return values as key/value pairs.

Since classes in Ruby are really just objects themselves, you'd probably expect that you can stub classes just like stubbing instance objects. You'd be right:

mocks/01/gatherer/test/models/project_test.rb

```
Line 1 test "let's stub a class" do
   2   Project.stubs(:find).returns(Project.new(:name => "Project Greenlight"))
   3   project = Project.find(1)
   4   assert_equal("Project Greenlight", project.name)
   5 end
```

In this test, the class Project is being stubbed to return a specific project instance whenever find is called. In line 3, the find method is, in fact, called, and returns that object.

Let's pause here a second and examine what we've actually done in this test. We're using find to get an ActiveRecord object. And because we are stubbing find we're not touching the actual database. Using the database is, for testing purposes, slow. Very slow. And this is one strategy for avoiding the database. In the meantime, remember that this stub shouldn't be used to verify that

the find method works; it should be used by other tests that need the find method along the way to the other logic that is actually under test.

That said, in actual practice, you also should avoid stubbing the find method because it's part of an external library—you might be better off creating a model method that has a more meaningful and specific behavior and stubbing that method.

On a related note, if you wanted to create multiple partial stubs from the same class and have them all behave the same way, you can do so with the method any_instance, as in:

```
Project.any_instance.stubs(:save).returns(false)
```

You should use any_instance very sparingly, as it often implies that you don't really understand what the underlying code is doing or where objects might come from. However, when you are testing legacy code you may genuinely not know where the object comes from, and any_instance might be a lesser evil. Also, sometimes framework concerns in Rails make using any_instance easier than managing the set of objects that might be returned by some distant method. In either case, though, consider the possibility of refactoring the code or test to avoid any_instance.

Prescription 13	The use of the 'any_instance' stub modifier often means the underlying code being tested could be refactored with a more useful method to stub.

A very common use of stub objects is to simulate exception conditions. If you want your stubbed method to raise an exception, you can use the raises method, which takes an exception class and an optional message:

```
stubby.stubs(:user_count).raises(Exception, "oops")
```

Mock Expectations

A mock object retains the basic idea of the stub—returning a specified value without actually calling a live method—and adds the requirement that the specified method must actually be called during the test. In other words, a mock is like a stub with attitude, expecting—nay, demanding—that its parameters be matched in the test or else we get a test failure.

As with stubs, Mocha provides a way to create whole mocks that exist just as mocks, as well as a way to create partial mocks that add expectations to an existing object. The method for whole mock creation is mock:

```
test "a sample mock" do
  mocky = mock(name: "Paul", weight: 100)
  assert_equal("Paul", mocky.name)
end
```

This test fails:

```
  1) Failure:
ProjectTest#test_a_sample_mock […]:
not all expectations were satisfied
unsatisfied expectations:
- expected exactly once, not yet invoked:
  #<Mock:0x7fe302d2ab58>.weight(any_parameters)
satisfied expectations:
- expected exactly once, invoked once:
  #<Mock:0x7fe302d2ab58>.name(any_parameters)
```

It fails because the first line sets up two mock expectations, one for mocky.name and one for mocky.weight, but only one of those two mocked methods are called in the test. Hence, it's an unsatisfied expectation. To pass the test, add a call to mocky.weight:

```
test "a sample mock" do
  mocky = mock(name: "Paul", weight: 100)
  assert_equal("Paul", mocky.name)
  assert_equal(100, mocky.weight)
end
```

The method for adding a mock expectation to a message on an existing object is expects: (I know, you'd think they'd have used mocks. But they didn't).

mocks/01/gatherer/test/models/project_test.rb

```
test "lets mock an object" do
  mock_project = Project.new(:name => "Project Greenlight")
  mock_project.expects(:name).returns("Fred")
  assert_equal("Fred", mock_project.name)
end
```

All the modifiers we've seen so far like returns, raises, and any_instance, as well as ones we haven't seen such as with, can be added to a mock just like they can to a stub.

By default, themock and expects methods set a validation that the associated method is called exactly once during the test. If that does not meet your testing needs, Mocha has methods that let you specify the number of calls to the method. These methods are largely self-explanatory:

```
proj = Project.new
proj.expects(:name).once
proj.expects(:name).twice
```

```
proj.expects(:name).at_least_once
proj.expects(:name).at_most_once
proj.expects(:name).at_least(3)
proj.expects(:name).at_most(3)
proj.expects(:name).times(5)
proj.expects(:name).times(4..6)
proj.expects(:name).never
```

In practice, the default behavior is good for most usages, though never is sometimes useful to guarantee that a particular expensive method is not called. Note that a stub is completely equivalent to to a mock expectation defined with at_least(0), as in: proj.expects(:name).at_least(0).

Using Mocks To Simulate Rails Save

You can use mock objects to rectify a nagging annoyance in the standard Rails scaffolds (well, it annoys me). The Rails-generated tests for a scaffolded controller created with rails generate scaffold controller do not cover the failure conditions for create and update. I've always assumed, with no real justification, this oversight was because the easiest way to test these is with a mock package, and the Rails team didn't want to mandate one particular package.

We've already mandated a mock package. And we've already written a failure test for ProjectController#create (see *Testing for failure*, on page 50). But, if you'll recall, we had to think of a case where the save would fail. We won't always be able to do that so easily, but we can mock objects to ensure a failing save. I'll throw in the failing update test, even though we didn't put update in our controller in the earlier tutorial — it's still a test structure you might find useful.

Our controller test for create will take advantage of the fact that we're using an external action object.

Let's think about these tests from a Given/When/Then perspective.

- Given: The create test needs some way to simulate failure. In the earlier tutorial, we used a set of form values that were invalid. In this case, we'll use a stub. The update test needs an existing object, and a way to simulate failure on update.
- When: The actual save or update_attributes. For create, where save happen inside the CreatesProject action, we just need to simulate a failure of that action.
- Then: The temptation is to test that new objects are not created and existing objects not updated. Those values represent the ending state of the action. Since we are stubbing save and update_attributes, however, testing

the state is pointless—the object is not being saved because the real save method is prevented from being called by the stub. Instead, we're testing the controller, and we want to test the controller's response. Namely, that a failed create action goes back to the new form, and a failed update goes back to edit.

Here are the tests:

mocks/01/gatherer/test/controllers/projects_controller_test.rb

```
Line 1 test "fail create gracefully" do
   -   action_stub = stub(create: false, project: Project.new)
   -   CreatesProject.expects(:new).returns(action_stub)
   -   post :create, :project => {:name => 'Project Runway'}
     5 assert_template('new')
   - end
   -
   - test "fail update gracefully" do
   -   sample = Project.create!(name: "Test Project")
    10 sample.expects(:update_attributes).returns(false)
   -   Project.stubs(:find).returns(sample)
   -   patch :update, id: sample.id, project: {name: "Fred"}
   -   assert_template('edit')
   - end
```

The create tests starts on line 2 by defining a full double that will stand in for the CreatesProject action. All we need for that double to do is respond to the create method with false and respond to the project method with a dummy Project. On the next line, we stub CreatesProject#new to return the action_stub when the controller method tries to create the action. We then call the create normally. When the create method executes, the stub is returned in place of the action, the controller method interprets as failure the false result when create is called on the action, and the new template is invoked.

The update test is similar, except that the update test doesn't have a separate action object, it just calls the regular ActiveRecord update_action. The steps are similar, though. We create a dummy object, in this case a Project. (It's going into the database because doing so is the simplest, if not the fastest way to give the object an ID). We then set up the failure condition on line 10 by stubbing update_attributes on the new object to return false. And then we stub Project#find on line 11 so that the stubbed object we've created will be returned by the controller action. Otherwise, the controller action would go to the database and create a new instance of the same object. That new object would not have the update_attribute stub, and presumably would not fail when updated. We then call the update normally, the controller takes the stubbed object, interprets the failure of update_attribute, and invokes the edit template.

And, for reference, the scaffolded update method that passes this test.

mocks/01/gatherer/app/controllers/projects_controller.rb
```
def update
  @project = Project.find(params[:id])
  if @project.update_attributes(params[:project])
    redirect_to @project, notice: "'project was successfully updated.'"
  else
    render action: 'edit'
  end
end
```

You also need a blank file template file at app/views/projects/edit.html.erb, since it's referenced by the controller method.

There's a slight difference between the two tests, the update method is actually stubbing ActiveRecord classes that are not part of our application proper. This should give us pause. Normally, we'd start looking for ways to wrap that behavior in an extracted method so that we can stub the extracted method. However, in this case, the update_attributes is so simple that the extraction would make the overall code more complex, so it's probably not worth it.

Prescription 14	If you are stubbing methods that do not belong to your program, think about whether the code would be better if restructured to wrap the external behavior.

We'll expand on this technique in Chapter 13, *Testing External Services*, on page 231, where we'll show when to use test doubles when testing external services.

Using Mocks To Specify Behavior

In addition to merely replacing expensive method calls, mock objects enable a different style of testing, where you validate the behavior of the application rather than its ending state. In most of the tests we've seen throughout the book, the test validates the result of a computation: it's testing whether something is true at the end of an action. When using mocks, however, we have the opportunity to test the behavior of the process during the test, rather than the outcome.

Let's look at an example. Back in *Who Controls the Controller?*, on page 48, we wrote the following test of our ProjectsController:

mocks/01/gatherer/test/controllers/projects_controller_test.rb
```
test "the project method creates a project" do
  post :create, project: {name: "Runway", tasks: "start something:2"}
```

```
  assert_redirected_to projects_path
  assert_equal "Runway", assigns[:action].project.name
end
```

At the time I mumbled something about being careful not to duplicate test model functionality, and said we'd do something different after we'd covered some more tools. Guess what—we've covered a new tool, and now it's time for us to apply it to the controller test.

This test makes two assertions — it asserts that the successful creation redirects to the projects listing page and it asserts that an instance variable named @actions is correctly set with the proper project name. Actually, let's split that last assumption into two parts. The controller is a) setting a particular instance variable and b) giving it a value matching the incoming data.

Setting the instance variable is part of the controller's logic, because it's an expectation that the view template will have when it renders, it will expect to have a particular value at @actions. However, the specific value that goes there, that value isn't really the controller's responsibility. The controller is acting as a conduit, its job is to just get the value from some data source and pass it on. Using the controller test to verify the value is just misplaced.

What, then, is the controller's responsibility? The controller is a conduit. By testing that it sets @actions we're testing the output of the conduit. The input of the conduit, though, is verified not by the state of the variable at the end of the controller call, but by the way in which that value is obtained. In other words, the controller's responsibility is to meet a contract on both ends—it's responsible for setting a particular value to satisfy the view, and it's responsible for calling some other place in the system to acquire data.

Specifically, the controller calls CreatesProject.new and then calls create on the resulting action. Crucially, we leave the fact that CreatesProject accurately creates a project to the tests for that object (tests we also wrote at the time). So all we need to do here is specify that the controller actually calls the appropriate methods.

Enter mocks. We use mock objects to set an expectation for the behavior of the controller during the user action. This is going to look similar to our failure test from the last section, but I'm going to emphasize a different part of the test.

mocks/01/gatherer/test/controllers/projects_controller_test.rb

```
Line 1 test "the project method creates a project (mock version)" do
   2   fake_action = mock(create: true)
   3   CreatesProject.expects(:new)
   4       .with(name: "Runway", task_string: "start something:2")
```

```
5       .returns(fake_action)
6   post :create, project: {name: "Runway", tasks: "start something:2"}
7   assert_redirected_to projects_path
8   refute_nil assigns[:action]
9 end
```

Well, we're still making the same controller call to create and we're still testing the redirection. Everything else has changed.

On lines 2 and 3, we create our doubles. We need to do this in two steps because the controller both instantiates a CreatesProject object, and also calls create on it. At the end of the test, on line 8, we've changed the test for the @action, now all we are testing is that the variable is set to something non-nil. Which seems like a weaker test, but is actually a more accurate representation of the actual responsibilities of the controller.

The power of this test is in the mock expectations. We're validating all of the following:

- The controller calls CreatesProject.new
- That call passes a name and a task_string as key value pairs based on the incoming parameters.
- The controller calls create on the return value of CreatesProject.new.

That's a fairly detailed description of the responsibilities of the controller.

This test does not even attempt to validate features of other objects. In fact, this test will pass even if CreatesProject does not exist. That's great, in that a bug in CreatesProject will only trigger one test failure, making it easier to track down. That's terrifying in that it's possible for this to pass without the underlying code working if CreatesProject doesn't exist. We could add a line that fake_project.respond_like_instance_of(CreatesProject) if we wanted to protect against a mismatch between our test double and the actual project method names.

Because it is possible to have tests pass due only to mismatches between the API and the test double, generally I only do this kind of testing based on setting mock expectaion if there is a separate integration test tying all the small, focused, unit tests together.

A plus for this test is speed. Since it doesn't actually contact the model layer, it's probably going to be fast.

A downside is readability. It can be genuinely hard to look at a mocked test and determine exactly what is being validated. Using spies can help here, because spies force you to be explicit about the expectations you are claiming in the test.

Finally, an elaborate edifice of mocked methods runs the risk of causing the test to be dependent on very specific details of the method structure of the object being mocked. This can make the test brittle in the face of refactorings that might change the object's methods. Good API design and an awareness of this potential problem go a long way toward mitigating the issue.

I'm hesitant to put a "don't do this" example on the page, but we could easily have made this test far more brittle if we had started to worry about tasks being part of projects

```
test "don't do this" do
  fake_action = mock(create: true,
    project: stub(name: "Fred", tasks: [stub(title: "Start", size: 2)]))
    # and so on…
```

In this snippet, fake_action isn't merely concerned with reporting success, it also wants to stub the project and have the project stub the array of tasks. This is where test doubles become a pain. The above snippet is hard to set up, it's hard to read, and it's very brittle against changes to object internals. If you find yourself needing to write nested mocks like this, try to restructure your code to reduce dependencies.

Prescription 15	A stubbed method that returns a stub is usually okay. A stubbed method that returns a stub that itself contains a stub probably means your code is too dependent on the internals of other objects.

More Expectation Annotations

Many Happy Returns

There are a couple of advanced usages of returns that might be valuable now and again. If you have multiple return values specified, the stubbed method returns them one at a time:

mocks/01/gatherer/test/models/project_test.rb

```
test "stub with multiple returns" do
  stubby = Project.new
  stubby.stubs(:user_count).returns(1, 2)
  assert_equal(1, stubby.user_count)
  assert_equal(2, stubby.user_count)
  assert_equal(2, stubby.user_count)
end
```

The return values of the stubbed method walk through the values passed to returns. Note that the values don't cycle; the last value is repeated over and over again.

You can get the same effect with a little more syntactic sugar by using the then method. You can chain together as many of these as you want:

```
stubby.stubs(:user_count).returns(1).then.returns(2)
```

You can use then to chain regular returns and exceptional returns:

```
stubby.stubs(:user_count).returns(1).then.raises(Exception)
```

Mocks with arguments

You can tune a Mocha stub or mock to return different values based in the parameters passed to the method using the with method to filter incoming calls:

In its simplest form, shown earlier in the chapter, the with method takes one or more arguments. When the stubbed method is called, Mocha searches for a match between the arguments passed and the declared stubs and returns the value matching those arguments.

One thing to be careful of is that by setting expectations tied to specific input values, you are limiting the Mocha stub to only those input values. In other words, if we were to try Project.find(3) in this test, the test would fail with the following rather cryptic error message:

```
unexpected invocation: Project(id: integer, name: string,
   due_date: date, created_at: datetime,
   updated_at: datetime).find(3)
satisfied expectations:
- allowed any number of times, invoked once:
    Project(id: integer, name: string, due_date: date,
    created_at: datetime, updated_at: datetime).find(2)
- allowed any number of times, invoked once:
    Project(id: integer, name: string, due_date: date,
    created_at: datetime, updated_at: datetime).find(1)
```

What this message is saying is that Mocha doesn't know what to do find called with the argument 3. Somewhat verbosely, it does pass along the information that it knows what to do with find(1) and find(2) . In other words, we did something Mocha didn't expect, and Mocha doesn't like surprises.

Using with does not constrain the eventual return value, you can use both returns or raises after a with call.

You need to be careful here — using with makes Mocha powerful and flexible, but in general, testing with mock objects works best if they are weak and rigid. The use of a complicated mock object suggests the existence of an overly complex dependency in your code.

Many of these matchers make more sense when you are talking about mock expectations rather than just stubs. When you are actually validating the behavior of the object being mocked, then having a tighter filter on the incoming values you expect makes more sense.

That said, there are two occasionally useful things you can do with with. First off, it can take a block as an argument, in which case the value passed to the method is passed on to the block. If the block returns true, then the expectation is considered matched:

```
proj = Project.new()
proj.expects(:status).with { |value| value % 2 == 0 }
    .returns("Active")
proj.expects(:status).with { |value| value % 3 == 0 }
    .returns("Asleep")
```

If more than one block returns true, the last one declared wins. If none of the blocks return true, we get the same unexpected invocation error listed above.

The argument to with can also be one of a series of parameter matchers. Probably the most useful one is instance_of, which can be used to simulate behavior given different types of input.

```
proj = Project.new()
proj.expects(:tasks_before).with(instance_of(Date)).returns(3)
proj.expects(:tasks_before).with(instance_of(String)).raises(Exception)
```

The instance_of matcher or any other Mocha matcher can be negated with the Not method. (Yes, it's capitalized, presumably to avoid weird parse collisions with the keyword not.)

```
proj = Project.new()
proj.expects(:tasks_before).with(Not(instance_of(Date))).returns(3)
```

The only other one of these that I've ever found even remotely valuable in practice is the has_entry matcher which works against a hash:

```
proj.expects(:options).with(has_entry(verbose: true))
```

The stub in this snippet will match any hash argument that contains a verbose: true entry, no matter what the other contents of the hash might be. Using the has_entry matcher is occasionally valuable against, say an ActiveRecord or

controller method that expects a hash where we only care about one of the methods.

There's about a dozen more of these matchers, many of which seem to be, shall we say, somewhat lacking in real-world practical value. Rather than cluttering your head with a bunch of stuff you'll never use, I invite you to check out the Mocha docs at http://gofreerange.com/mocha/docs/ for a full listing.

Mock Tips

My opinion about the best way to use mock objects changes every few months. I'll try some mocks, it'll work well, I'll start using more mocks, they will start getting in the way, I'll back off, then I'll think, "let's try some mocks". This cycle has been going for about five years, and I have no reason to think it's going to change anytime soon.

That said, there are some guidelines that are always true.

The Ownership Rule:

"Don't Mock What You Don't Own". In other words, only use test doubles to replace methods that are actually part of your application, and not part of an external framework. (Note that we violated this rule in this chapter when we stubbed ActiveRecord methods like `update_attributes`.)

One reason to only mock methods you control is, well, that you control them. One danger in mocking methods is that your mock either doesn't receive or doesn't return a value that is actually a reasonable value from the method being replaced. If the method in question is belongs to a third-party framework, the chance that it will change without you knowing is increased, and thus the test becomes more brittle.

More importantly, is that mocking a method you don't own couples your test to the internal details of the third party framework. By implication, this means that the method being tested is also coupled to those internal details. Which is bad, not just if the third party tool changes, but also if you want to refactor your code, the dependency will make that change more complicated.

The solution, in most cases, is to create a method or class in your application that calls the third party tool and stub that method. (While also writing tests to ensure that the wrapper does the right thing) We'll see a larger example of this technique in Chapter 13, *Testing External Services*, on page 231. However, in the small, you can do this as easily as:

```
class Project
```

```
  def self.create_from_controller(params)
    create(params)
  end
end
```

And then in a test:

```
test "create a project" do
  Project.stubs(:create_from_controller).returns(Project.new)
end
```

I admit that, oversimplified like this, the technique seems like overkill. (All Object-Oriented techniques seem like overkill until you suddenly realize you needed them six months ago). If you create a method like this wrapper, however, you'll often find that it is functionality that is shared in multiple places, giving you one point of contact with the third party tool rather than several. You'll also find that these methods tend to attract what would otherwise be duplicated behavior. Both of these are good things.

When to Mock, When to Stub

If you are using your fake objects to take the place of real objects that are hard or impossible to create in a test environment, it's probably a good idea to use stubs rather than mocks. If you are actually using the fake value as an input to a different process, then you should test that process directly using the fake value rather than a mock. Adding the mock expectation just gives you another thing that can break, which in this use case is probably not related to what you are actually testing.

On the other side, if you are testing the relationship between different systems of your code, tend to use mocks to verify the behavior of one part of the code as it calls the other.

Mocks are particularly good at testing across boundaries between subsystems. For example, controller testing to isolate the controller test from the behavior of the model, essentially only testing that the controller makes a specific model call and using the model test to verify model behavior. Among the benefits of using mocks this way is you are encouraged to make the interface between your controllers and models as simple as possible. However, it does mean that the controller test knows more about your model than it otherwise might, which may make the model code harder to change.

You also need to be careful of mocking methods that have side effects or that call other methods that might be interesting. The mock totally bypasses the original method, which means no side effect and no calling the internal

method. Pro tip: saving to the database and outputting to the response stream are both side effects.

Be very nervous if you are specifying a value as a result of a mock and then asserting the existence of the very same value. One of the biggest potential problems with any test suite is false positives, and testing results with mocked values is a really efficient way to generate false positives.

Mocks are Design Canaries

Test-Driven Development in general is a sensitive indicator of the quality of your application code. As the code becomes more complex and tightly coupled, the tests will become harder and harder to write. Mock objects are a very sensitive indicator of code quality. As the code becomes more tightly coupled, tests with mocked method calls will become harder to write really really quickly.

When you are using a true mock to encapsulate a test and isolate it from methods that are not under test, try to limit the number of methods you are mocking in one test. The more mocks, the more vulnerable the test will be to changes in the actual code. A lot of mocks may indicate that your test is trying to do too much or might indicate a poor object-oriented design where one class is asking for too many details of a different class.

Coming Up Next

In this part of the book, we covered model testing. First, we talked about the services Rails provides for testing models, and we discussed fixtures and factories as mechanisms for creating consistent test data. With this chapter, we've started to transition from testing models to testing the user-facing parts of the application. Mock testing is useful for testing models, but it becomes especially useful when trying to shield the various layers of your application from each other.

In the next part, we'll be discussing testing the controller and view layers. Mock objects can be a very important part of controller testing; creating mock models allows the controller tests to proceed independently of the model test.

CHAPTER 9

Testing Controllers And Views

Rails application follow a Model View Controller, or MVC, pattern. The view layer has the responsibility of taking data and presenting it to the user, which in a server-side web application usually means generating HTML. Ideally, the view layer does this with minimal interaction with the model. The controller takes in information about the user request, contacts the appropriate parts of the model layer for data, and passes that information on to the model. A very simplified diagram is in *Simple MVC Architecture*

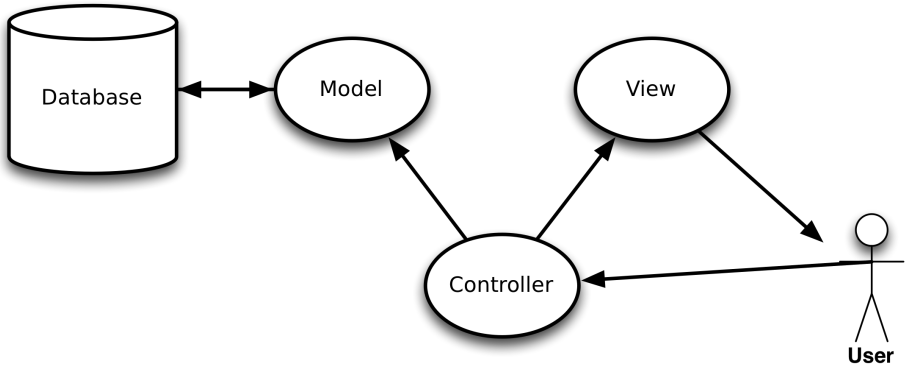


Figure 3—Simple MVC Architecture

Testing Rails controllers and Rails views are more challenging than testing Rails models. You can see from the diagram that controllers and views both interact with the external users, whereas models are more inherently isolated. In Rails, Controller and view instances are typically created by the framework itself, and are not easy to create in isolation during a test (as far as the Rails developer is concerned, the view instance is mostly just a template). Controller

and view calls often are more interesting for their side effects than for the value returned by them. Also, individual controller actions and view templates are often too large to be meaningfully unit tested.

While the Rails framework and third party testing tools allow us to interact with controller actions and and view templates in our test environment, the issues of isolation and size still exist. A discussion of how to best test views and controllers, therefore, often turns into a discussion about what code belongs in the controller and view and what should be extracted into a different object. The object extracted to, which is not a part of the Rails framework, is sometimes referred to as a PORO: Plain Old Ruby Object. This issue of how to best deconstruct or refactor controller and view code is somewhat contentious within the Rails community.

We've discussed the idea that the most useful tests are either testing an entire end-to-end process, or testing a single unit. Controller and view tests are easy to put in the middle ground, and are therefore notoriously brittle and hard to manage.

Testing Controllers

We've already written a few controller tests as part of our earlier testing walkthroughs. Let's take a look at one of them:

display/01/gatherer/test/controllers/projects_controller_test.rb

```
Line 1 require 'test_helper'
   2
   3 class ProjectsControllerTest < ActionController::TestCase
   4
   5   test "the project method creates a project" do
   6     post :create, project: {name: "Runway", tasks: "start something:2"}
   7     assert_redirected_to projects_path
   8     assert_equal "Runway", assigns[:action].project.name
   9   end
```

This test is simple but has most of the features of a basic controller test. Like many tests we have seen, controller tests have three parts. First, the controller test may create data needed to cover a particular logic path. We don't need any data for this test, but we will see examples of generating controller test data in our next examples. Secondly, on line 6 the code performs an action. Specifically, it simulates a `post` request to the controller's `create` action with one argument, the hash `{name: "Runway", tasks: "start something:2"}` which represents the parameters being passed to the action as part of the request.

Finally, on lines 7 and 8, our tests makes assertions about the behavior of the controller. Broadly, we care about two kinds of behavior. We care about

what template or other action the controller passes control to. The assert_redirect_to method is one of a few assertions added by Rails to specify that transfer of control. We may also care that the controller specifies particular instance variables for use by a view template. The assigns hash is one of a few objects that the test object manages to enable assertions to be made about those values.

What to test in a controller test

Ideally, your controllers are relatively simple. The complicated functionality is in a model or other object and is being tested in your unit tests for those objects. One of reasons why this is a best practice is that models are easier to test because they are generally easier to extract and use independently in a test framework.

Prescription 16	A controller test should test controller behavior. A controller test should not fail because of problems in the model.

A controller test that overlaps with model behavior is part of the awkward middle ground of testing that we're trying to avoid. If the controller test is actually going to the database, then the test is slower than it needs to be, and if a model failure can cascade into the controller tests, then it's harder than it needs to be to isolate the problem.

A controller test should have one or more of the following goals:

- Verifying that an normal, basic user request triggers expected model calls and passes the necessary data to the view.
- Verifying that an ill-formed or otherwise invalid user request is handled properly, for whatever definition of "properly" fits your app.
- Verifying security, such as requiring logins for pages as needed and testing that users who enter a URL for a resource they shouldn't be able to see is blocked or diverted. We will discuss this more in Chapter 12, *Testing For Security*, on page 207.

Simulating Requests in a Controller Test

Most of your controller tests in Rails will surround a simulated request. To make this simulation easier, Rails provides a controller test method for each HTTP verb: delete, get, head, patch, post, and put. Each of these methods works the same way (internally, they all dispatch to a common method that does all the work). A full call to one of these methods has 5 arguments, though you'll often just use the first three:

```
get :show, {id: @task.id}, {user_id: "3",
    current_project: @project.id.to_s}, {notice: "flash test"}
```

The method name, get, is the HTTP verb being simulated—sort of. While the controller test will set the HTTP verb if for some reason you query the Rails request object, it does not check the Rails routing table to see if that action is reachable using that HTTP verb. As a result, you can't test routing via a controller test. Rails does provide a mechanism for testing routes, which we'll cover in *Testing Routes*, on page 132

The first argument, in this case, :show, is the controller action being called. The second argument, {id: @task.id} is a hash that becomes the params hash in the controller action. In the controller action called from this test, you would expect params[:id] to equal @task.id. The Rails form name parsing trick is not used here—if you want to simulate a form upload, you use a multilevel hash directly like user: {name: "Noel", email: "noel@noelrappin.com"}, which implies params[:user][:name] == "Noel" in the controller.

Any value passed in this hash argument is converted to a string—specifically, to_param is called on it. So you can do something like id: 3, confident that it will be "3" in the controller. This, by the way, is a welcome change in recent versions of Rails, older versions did not do this conversion, which led to occasional heads pounding against walls.

If one of the arguments is an uploaded file, say from a multipart form, you can simulate that using the Rails helper fixure_file_upload(filename, mime_type), as in:

```
post :create, logo: fixture_file_upload('/test/data/logo.png', 'image/png')
```

If you are using a third party tool, such as Paperclip or Carrierwave to manage uploads, those tools typically have more specific testing helpers.

The fourth and fifth arguments to these controller methods are optional and rarely used. The fourth argument sets key/value pairs for the session object, which is useful in testing multi-step processes that use the session for continuity. The fifth argument represents the Rails flash object, which is useful, well, basically never, but if for some reason the incoming flash is important for your logic, there it is.

You may occasionally want to do something fancier to the simulated request. In a controller test, you have access to the request object as @request, and access to the controller object as @controller. (As we'll see in *Evaluating Controller Results*, on page 129, you also have the @response object). You can get at the HTTP headers using the hash @request.headers.

There is one more controller action method, xml_http_request (also aliased to xhr). This simulates a classic Ajax call to the controller, and has a slightly different signature:

```
test "my ajax call" do
  xhr :post, :create, :task => {:id => "3"}
end
```

The method name is xhr, the first argument is the HTTP verb associated with the xhr call, and the remaining arguments are the arguments to all the other controller-calling methods in the same order: action, params, session, flash. The xhr call sets the appropriate headers such that Rails controller will appropriately be able to consider the request an Ajax request (meaning that .js format blocks will be triggered), then simulates the call based on its arguments.

Evaluating Controller Results

A controller test has three things that you might want to validate after the controller action:

- Did it return the expected HTTP status code? Rails provides the assert_response method for this purpose.
- Did it pass control to the expected template or redirected controller action. Here, we have assert_template and assert_redirected_to.
- Did it set the values that the view will expect. For this, we have the special hash objects assigns, cookies, flash, and session.

Often, you'll combine more than one of these in the same test:

```
test "successful index request with no filters" do
  get :index
  assert_response :success
  assert_template :index
end
```

Asserting Controller Response Type

Let's talk about these three types of assertions in more detail:

We can use assert_response to verify the HTTP response code sent back to the browser from Rails. Normally we use this assertion to ensure that our controller correctly distinguishes between success and redirect or error cases.

The value passed to assert_response is usually one of four special symbols:

Symbol	HTTP Code Equivalent
:success	200
:redirect	300 - 399
:missing	404
:error	500 - 599

If you need to test for a more specific response, you can pass either the exact HTTP code number or one of the associated symbols defined by Rails, which you can find at http://guides.rubyonrails.org/layouts_and_rendering.html. The most common case I've had for specific HTTP codes is the need to distinguish between 301 permanent redirects and other redirects.

Asserting Which View is Rendered

The assert_template method is used to determine whether the controller is passing control to the expected view template. The method has a simple form, and then some optional complexity. In the simple form, assert_template is passed a template name which is specified exactly as it would be in the controller using render :action—the template name can be a string or a symbol. If the argument is just a single string or symbol, then it is checked against the name of the main template that rendered the action.

Normally, I will not use assert_template when the controller action is just using the implicit Rails default and ceding to a view of the same name. I will use assert_template when I expect the controller will need to explicitly pass control to a specific template, with the most common case being a create action that is successful and renders a show template.

There are two optional arguments to assert_template. The layout option specifies that the template in question was rendered against a particular Rails layout (you can explicitly assert the no-layout case by passing layout: false or layout: nil). The common use case here is perhaps using different layouts for mobile or desktop browsers based on request headers.

If you want to make sure that a specific partial is rendered along the way, use the :partial option, as in assert_template partial: '_user_data_row'. In this case, you are testing whether the specified partial is called when the controller action is rendered. The partial name in the test method must include the leading underscore. Adding the :count option verifies that the specified partial was called a specific number of times, which is potentially useful for a partial that is rendered inside a loop. This is perilously close to view testing, so I wouldn't recommend testing for partials in general. I have had cases where the partial

to be rendered was determined dynamically in the view, so testing that the logic is correct was useful in that case.

When you expect the controller to redirect, assert_redirected_to can be used to assert the exact nature of the redirect. The argument to assert_redirected_to is pretty much anything that Rails can convert to a URL, although the behavior of the method is slightly different based on what the argument actually is. The code for assert_redirected_to explicitly includes assert_response :redirect, so you don't need to duplicate that assertion.

If the argument to assert_redirected_to is the name of a URL, either because it's a string, a method for a Rails named route, or an object that has a Rails RESTful route, then the assertion passes if and only if the redirecting URL exactly matches the asserted URL. For testing purposes, Rails will assume that the local hostname of the application is http://www.example.com. If that's too exact a test for your taste, you can pass a hash to assert_redirected_to, which specifies the :controller, :action, and any parameters. If the argument is a hash, then assert_redirected_to only checks the exact key/value pairs in the hash, other parts of the URL are not checked.

Rails controller tests do not—repeat, do not—follow the redirect. Any data validation tests you write apply only to the method before the redirect occurs. If you need your test to follow the redirection for some reason, you are cordially invited to try something in an integration test; see Chapter 11, *Integration Testing with Capybara and Cucumber*, on page 179.

Asserting Controller Data

Rails allows you to verify the data generated by the controller action being tested with the four hashes mentioned earlier: assigns, session, cookies, and flash. Of these, assigns, which is a hash of instance variables created in the controller, is the most commonly used. A typical use might look like this, with a common use of assigns, and a frankly contrived use of session:

```
test "should show task" do
  get :show, id: @task_1.id
  assert_response :success
  assert_equal @task_1.id, assigns[:task].id
  assert_equal "task/show", session[:previous_page]
end
```

The cookies and flash special variables are used similarly, though I don't write tests for the flash very often. The cookie hash only has key/value pairs for cookies. If you want to test other cookie attributes, you need to access them via the @request object.

Testing Routes

Although the basics of Rails routing is simple, the desire to customize Rails response to URLs can lead to confusion about exactly what your application is going to do when converting between an URL and a Rails action, and vice versa. Rails provides a way to specify route behavior in a test.

Routing tests are not typically part of my TDD process—usually my integration test implicitly covers the routing. That said, sometimes routing gets complicated and has some logic of its own (especially if you are trying to replicate an existing URL scheme), so it's nice to have this as part of your test suite when needed.

The primary method that Rails uses for route testing is assert_routing. Here's a sample test that includes all seven default RESTful routes for the project resource:

display/01/gatherer/test/controllers/projects_controller_test.rb

```
  test "routing" do
    assert_routing "/projects", controller: "projects", action: "index"
    assert_routing({path: "/projects", method: "post"},
        controller: "projects", action: "create")
    assert_routing "/projects/new", controller: "projects", action: "new"
    assert_routing "/projects/1", controller: "projects",
        action: "show", id: "1"
    assert_routing "/projects/1/edit", controller: "projects",
        action: "edit", id: "1"
    assert_routing({path: "/projects/1", method: "patch"},
        controller: "projects", action: "update", id: "1")
    assert_routing({path: "/projects/1", method: "delete"},
        controller: "projects", action: "destroy", id: "1")
  end

end
```

The first argument to assert_routing represents the path. As you can see from the above examples, this argument is either a string representing the URL, or a hash with a :path key representing the URL and a :method key representing the HTTP method being invoked. If no method is specified, the default is GET.

The second argument is a hash representing any the elements of the route after it is processed by the Rails router, meaning you would expect this argument to specify a :controller, an :action, and any other symbols which are defined by the route. This second argument does not contain any elements which are expected to be part of the query string as opposed to the base URL itself.

The third argument is defined as defaults. As far as I can tell, it's essentially merged into the second argument. (Actually, the documentation says this parameter is unused, though its clearly referenced in the source code.) It seems to be safe to leave it as an empty hash, if you need the fourth argument. That fourth argument is where you specify key/value pairs for any part of the route that you expect to be in the query string.

The assert_routing method validates the routing in both directions. It checks that when you send the path through the routing engine, you get the controller, action, and other variables specified. It also checks that the set of controller, action, and other variables sent through the router results in the path string (which is why you might need to specify query string elements).

It's not completely clear to me why a route might pass in one direction and not the other. You can, however, run the individual checks using assert_generates to go from a string to a hash, and assert_recognizes to go from the hash to the string.

View Tests

The Rails framework allows you to make assertions from a controller test about the nature of the view that is rendered. I used to write this type of test frequently, but over time, I've found that I've tended to handle my views either by:

- Integration tests, which start with an actual request rather than a simulated call to a controller action, and which typically validate their results via examining the view.
- Regular unit tests of either helper methods or presenter objects which have been extracted out of the view layer and which can be unit tested directly.

In between, there are the Rails framework view tests, which are somewhat faster than integration tests, but somewhat less focused than unit tests.

In addition to the ability to test the markup generated by a view, the Rails framework provides the ability to test mail functionality, and helper methods.

Testing Helper Methods

Helper methods are the storage attic of Rails applications. Helper modules are designed to contain reusable bits of view logic. This might include view-specific representations of data, or conditional logic that governs how content is displayed. In practice, helper modules tend to get filled with all kinds of

clutter that doesn't seem to belong anywhere else. Because they are a little tricky to set up for testing, helper methods often aren't tested even when they contain significant amounts of logic.

Rails provides the ActionView::TestCase class, which is a subclass of ActiveSupport::TestCase specifically designed to load enough of the Rails controller structure to enable helpers to be called and tested. Since we created our models using Rails scaffolding, empty helper files already exist. In test/helpers/project_helper_test.rb, we have the following:

```
require 'test_helper'

class ProjectsHelperTest < ActionView::TestCase
end
```

Let's say we want to change our project view so that projects that are behind schedule show up differently. We could do that in a helper. My normal practice is to add a CSS class to the output for both the regular and behind schedule cases, to give the design maximum freedom to display as desired.

Here's a test for that helper.

display/01/gatherer/test/helpers/projects_helper_test.rb

```
Line 1 require 'test_helper'
   -
   - class ProjectsHelperTest < ActionView::TestCase
   -
   5   test "project name with status info" do
   -     project = Project.new(name: "Project Runway")
   -     project.stubs(:on_schedule?).returns(true)
   -     actual = name_with_status(project)
   -     expected = "<span class='on_schedule'>Project Runway</span>"
  10     assert_dom_equal expected, actual
   -   end
   - end
```

In this test, we're creating a new project using a standard ActiveRecord new method—we could use FactoryGirl, I suppose. Rather than define a bunch of tasks that would mean that the new project is on schedule, on line 7 we just stub the on_schedule? method to return true. This has the advantage of being faster than creating a bunch of objects, and also, I think, being clearer as to the exact state of the project being tested.

We use assert_dom_equal to compare the string we expect to the string we actually get from the helper. The assert_dom_equal assertion compares that two strings both resolve to equivalent DOM structures, which means that it tests that attributes have identical values, but don't necessarily need to be in the same

order. It's nice to have, because it spares us some fiddling around with HTML strings in our test.

That test will fail because we haven't defined the name_with_status helper. Let's define one:

display/01/gatherer/app/helpers/projects_helper.rb

```
Line 1 module ProjectsHelper
   2
   3   def name_with_status(project)
   4     content_tag(:span, project.name, class: 'on_schedule')
   5   end
   6 end
```

Tests pass, and now let's add a second test for the remaining case. This test will look familiar.

display/02/gatherer/test/helpers/projects_helper_test.rb

```
test "project name with status info behind sechdule" do
  project = Project.new(name: "Project Runway")
  project.stubs(:on_schedule?).returns(false)
  actual = name_with_status(project)
  expected = "<span class='behind_schedule'>Project Runway</span>"
  assert_dom_equal expected, actual
end
```

Which passes with the following...

display/02/gatherer/app/helpers/projects_helper.rb

```
module ProjectsHelper

  def name_with_status(project)
    dom_class = project.on_schedule? ? 'on_schedule' : 'behind_schedule'
    content_tag(:span, project.name, class: dom_class)
  end
end
```

One gotcha that you need to worry about when view testing is using Rails-internal view methods like url_for. Although all core Rails helpers are automatically loaded into the ActionView test environment, there are one or two that have significant dependencies on the real controller object and therefore fail with opaque error messages during helper testing. The most notable of these is url_for. One workaround is to override url_for by defining it in your own test case (the method signature is def url_for(options = {})). The return value is up to you; I find a simple stub response is often good enough.

Sometimes, helper methods take a block, which is expected to be ERb text. One common use of this kind of helper is access control, in which the logic in the helper determines whether the code in the block is invoked. Alternately,

blocks are very helpful as wrapper code for HTML that might surround many different kinds of text—a rounded rectangle effect, for example.

Here's a simple example of a helper that takes a block:

```
def if_logged_in
  yield if logged_in?
end
```

Which would be invoked like so:

```
<% if_logged_in do %>
  <%= link_to "logout", logout_path %>
<% end %>
```

To test the if_logged_in helper, we take advantage of the fact that the yield statement is the last statement of the helper, and therefore is the return value. We also take advantage of the fact that Ruby will let us pass any arbitrary string into the block, giving us tests that look like this:

```
test "does not display if not logged_in" do
  assert !logged_in?
  assert_nil(if_logged_in { "logged in" })
end

test "displays if logged in" do
  login_as users(:quentin)
  assert logged_in?
  assert_equal("logged in", if_logged_in { "logged in" })
end
```

The first test asserts that the block is not invoked so the helper returns nil, while the second asserts that the block is invoked, basically just returning the value passed into the block.

You have to be a little careful here, because these tests are just testing the return value of the helper method, not what is actually sent to the output stream. The output stream part is a side-effect of the process, but it is stored in a variable called output_buffer, which you can access via testing. So you could write the above tests as:

```
test "does not display if not logged_in" do
  assert !logged_in?
  if_logged_in { "logged in" }
  assert_nil(output_buffer)
end

test "displays if logged in" do
  login_as users(:quentin)
  assert logged_in?
```

```
  if_logged_in { "logged in" }
  assert_equal("logged in", output_buffer)
end
```

Testing View Markup

We've tested a helper for project status, but when we go to the browser, the new status DOM elements don't show up. This is because we haven't actually placed our new helper in the view template itself. Naturally, we would like our dazzling two-line helper to be incorporated into the view.

From a TDD perspective, we have a few options:

- Write no further tests, and just insert the helper into the view template. Technically, we're not adding logic, so we can kind of squeak by with this one. I don't mean to be glib here—often, no extra test is the right choice when the test is a) expensive, b) trivial in the larger scheme of things and c) easy to visually inspect.
- Write an integration test using Capybara, as we saw in Chapter 4, *Test-Driven Rails*, on page 37 and will see again in Chapter 11, *Integration Testing with Capybara and Cucumber*, on page 179. If we've been using outside-in development, we may already have an integration test in place.
- Write a Rails view test. The Rails view test has the advantage of being somewhat faster than the integration test, and we may be able to piggyback it on existing controller tests.

As it happens, we don't have a controller test for the index call, so lets build one that also verifies the inclusion of the view helper we just wrote.

Let's take a second to plan this test. What does this test need to do?

- Given: We need just two projects, one that is on schedule, one that's not on schedule. That allows us to to verify both halves of the helper. The projects we create need to be visible to the controller method, meaning we either need to put the data in the database or do some clever mocking (fixtures could be used too, but I don't want to create fixtures). Let's start with the database, it's simpler for the moment.
- When: We just need to hit the index action of the controller.
- Then: Our on schedule project has the on-schedule DOM class, and our behind schedule class has the behind-schedule one.

In code, that becomes:

display/02/gatherer/test/controllers/projects_controller_test.rb

```
test "the index method displays all projects correctly" do
  on_schedule = Project.create!(due_date: 1.year.from_now,
```

```
      name: "On Schedule",
      tasks: [Task.create!(completed_at: 1.day.ago, size: 1)])
  behind_schedule = Project.create!(due_date: 1.day.from_now,
      name: "Behind Schedule",
      tasks: [Task.create!(size: 1)])
  get :index
  assert_select("#project_#{on_schedule.id} .on_schedule")
  assert_select("#project_#{behind_schedule.id} .behind_schedule")
end
```

The test matches the Given/When/Then structure. It's a little bit awkward to create the two projects quickly—we are taking advantage of a quirk that a project with no unfinished tasks is always on schedule. (FactoryGirl would only make this slightly easier, in that we probably wouldn't have to specify a name, unless we created specific :on_schedule and :behind_schedule factories.)

The new parts here are the assert_select calls in the last two lines. The assert_select method is defined by Rails and is typically used to make assertions about the existence of a selector pattern in the output rendered by a controller action.

In this case, we're testing for the existence of a selector pattern with a DOM id #project_#{on_schedule.id} with a subordinate object containing the DOM class.on_schedule and a similar selector pattern with the DOM id #project_#{behind_schedule.id} containing an item with the DOM class .behind_schedule. By default, the assert_select method uses the HTTP response of the controller action being tested as the target of the search.

The selector syntax that assert_select uses is very similar to jQuery and other DOM selection tools, though assert_select uses its own HTML parser. As in jQuery, the use of two separate selectors means that we expect to match an instance of the first selector which contains an instance of the second. For example, we're looking for an HTML element with the DOM ID of the form project_12, where 12 is the id of the on_schedule project. We also need for that outer HTML element to contain an inner HTML element with a DOM class on_schedule.

As it happens, the pattern project_12 is exactly what the Rails dom_id helper uses, and we'd previously put that in the tr element of each project in the index listing. So we are looking for an on_schedule class inside that view.

This test will fail with an error message which looks something like this, showing that the test is looking for a pattern that is not being found.

```
Expected at least 1 element matching
"#project_1 .on_schedule", found 0..
Expected 0 to be >= 1
```

A minor change in our index template gets the test to pass—we change the project name cell to use the name_with_status helper:

display/02/gatherer/app/views/projects/index.html.erb

```
<h1>All Projects</h1>
<table>
  <thead>
    <tr>
      <td>Project Name</td>
      <td>Total Project Size</td>
    </tr>
  </thead>
  <tbody>
    <% @projects.each do |project| %>
      <tr class="project-row" id="<%= dom_id(project) %>">
        <td class="name"><%= name_with_status(project) %></td>
        <td class="total-size"><%= project.total_size %></td>
      </tr>
    <% end %>
  </tbody>
</table>
```

Now, the tr field with the appropriate DOM id has a span element from the helper that contains the expected DOM class.

There are a number of ways we can augment assert_select. The selector argument to assert_select can be just an element, as in div, or an element plus a class or ID decoration, as in div.hidden. In the latter case, a matching element must have both the HTML tag and the DOM class or ID. As with other DOM selectors, . indicates a DOM class and # indicates a DOM id. You can also use brackets to indicate arbitrary HTML attributes, as in input[name='email'] . The selector is an instance of the ActionView class HTML::Selector, and a complete listing of the syntax can be found at http://api.rubyonrails.org/classes/HTML/Selector.html.

Prescription 17 When testing for view elements, try to test for DOM classes that you control, rather than text or element names that might be subject to design changes.

You can make a more specific match by including an optional text: argument. The value associated with the text is a string or a regular expression. The assertion passes if there is at least one HTML tag that matches the selector and has inner content that matches the argument. String arguments must match the content exactly. Regular expression arguments must =~ match the contents. In other words, we could make our assertions more specific like this:

```
assert_select("#project_#{on_schedule.id} .on_schedule", text: "On Schedule")
```

In the above case, the test still passes because On Schedule is the project name and is included in the .on_schedule span tag.

Additionally, you could pass a count: option, which takes either an integer or a range. If there is a count:, then the assert_select only passes if the number of matching elements either equals the number or is in the range. Therefore, we can verify that there is exactly one on schedule element on the page with the following:

```
assert_select("#project_#{on_schedule.id} .on_schedule", count: 1)
```

It's often useful to be a able to say that an element is not on the page—an edit button for a non-administrator, for example. You can do that by making the second argument to assert_select false, as in assert_select(selector, false).

The assert_select syntax has some other features that are basically identical to jQuery and Capybara, but which are rarely used. One trick I do want to mention is the ability to pass a block to assert_select and then have assert_select work on an arbitrary HTML element rather than the entire page output.

If you pass a block to assert_select, then asset_select invokes the block with a single argument containing an array of all HTML elements that match your selector, each element is of the Rails class HTML::Node. You can then do whatever you want with that array. A common choice is to iterate over the array and run further assertions, perhaps using assert_select. This is a different way to test for nested elements. Our original assert_select, which was "#project_#{on_schedule.id} .on_schedule" could instead be written as:

```
assert_select("#project_#{on_schedule.id}") do |matches|
  matches.each do |element|
    assert_select(element, ".on_schedule")
  end
end
```

This may be more explicit in some cases, or may allow for more complex logic for the internal selector.

The inner assert_select in that snippet is doing something we haven't seen. The first argument, element is one of the matching HTML::Node objects. When assert_select is passed an HTML::Node, then it searches that HTML for the matching selector, rather than the default behavior, which is to use the current HTTP response object.

That HTML::Node first argument is handy for the block syntax, but we can also take advantage of this to use assert_select against arbitrary strings that just

happen to be valid HTML with a parent root element. All we need is a helper method, which we place in our test_helper.rb file:

display/02/gatherer/test/test_helper.rb
```
def assert_select_string(string, *selectors, &block)
  doc_root = HTML::Document.new(string).root
  assert_select(doc_root, *selectors, &block)
end
```

We're using Rails HTML parsing classes, which aren't normally part of your balanced Rails breakfast, but allows us to parse a string into an HTML document and take the root element. That root element can then be used as the basis for an assert_select search.

This can come in handy when testing helpers—our helper test from earlier in the chapter, which validated that a particular DOM class was attached to the project name, could be rewritten like so:

display/02/gatherer/test/helpers/projects_helper_test.rb
```
  test "project name using assert_select" do
    project = Project.new(name: "Project Runway")
    project.stubs(:on_schedule?).returns(false)
    assert_select_string(name_with_status(project), "span.behind_schedule")
  end
end
```

To make this test work, we call assert_select_string, using the actual text returned by the helper method as the background for the search, and the expected DOM class as the selector argument. Since our expected string does have a span tag with the DOM class .behind_schedule, the test passes.

Presenters

Testing helpers is handy and all that, but if you have a lot of logic in your helpers, I recommend starting to move the logic into presenter objects. This is especially true if you have a series of helpers that take the same argument.

There's nothing necessarily complicated about using Presenters in Rails, I often roll my own using Ruby's SimpleDelegator class. If you want a little more structure, you can use the draper gem, which you can find at https://github.com/drapergem/draper.

We can take the project helper and convert it to a project presenter. This version of the code uses SimpleDelegator and includes a method for converting a list of projects into a list of presenters. In a break from our usual convention here, I'll show you the code first.

display/03/gatherer/app/presenters/project_presenter.rb

```
Line 1 class ProjectPresenter < SimpleDelegator
-
-   def self.from_project_list(*projects)
-     projects.flatten.map { |project| ProjectPresenter.new(project) }
5   end
-
-   def initialize(project)
-     super
-   end
10
-   def name_with_status
-     dom_class = on_schedule? ? 'on_schedule' : 'behind_schedule'
-     "<span class='#{dom_class}'>#{name}</span>"
-   end
15
- end
```

The main action here starts on line 7, with our initializer. All we need to do is call super, and SimpleDelegator will take care of the rest. Taking care of the rest means that if SimpleDelegator gets a message it doesn't understand, it automatically delegates it to the object passed to the constructor. In practice, this delegation means that we can treat the presenter as though it was an instance of the original object, plus any new methods we choose to add as decoration.

The name_with_status method in the presenter is simpler in one way and more complex in another. Since calls to methods like on_schedule? or name are now automatically delegated, there's no need to explicitly have the project as the message receiver, so we can just use on_schedule? rather than project.on_schedule?. However, since we are no longer inside a Rails helper, we no longer have access to the content_tag method we used to build the HTML output. Instead, we build the output as a string. (There are other options, including explicitly including the module that content_tag is a part of, but building the string is simplest in this case.)

Finally, at the top of the class, we have a method that takes in a list of Project instances and converts them to presenters. The *projects argument in conjunction with projects.flatten allows the method to be called with either an explicit array, ProjectPresenter.from_project_list([p1, p2]), or an implicit arbitrary list of projects, ProjectPresenter.from_project_list(p1, p2). If we were using presenters more frequently, this is the kind of method that would easily be abstracted to something generic, rather than being rewritten for each presenter class.

When we test the presenter, we're out of the realm of testing helpers, so we don't need ActionView::TestCase any more. In fact… well, let me show you:

display/03/gatherer/test/presenters/project_presenter_test.rb
```
require 'test_helper'

class ProjectPresenterTest < Minitest::Test

  def setup
    @project = Project.new(name: "Project Runway")
    @presenter = ProjectPresenter.new(@project)
  end

  def test_project_name_with_status_info
    @project.stubs(:on_schedule?).returns(true)
    expected = "<span class='on_schedule'>Project Runway</span>"
    assert_equal(expected, @presenter.name_with_status)
  end

  def test_project_name_with_status_info_behind_sechdule
    @project.stubs(:on_schedule?).returns(false)
    expected = "<span class='behind_schedule'>Project Runway</span>"
    assert_equal(expected, @presenter.name_with_status)
  end
end
```

Do you see what we've done here? Since our presenter class has no dependencies on Rails, we can write a test class that also has no dependency on Rails. Not only does this test not use ActionView::TestCase, it also doesn't use ActiveSupport::TestCase, and we've gone back to Minitest::Test as the parent class for the test.

We've given up a couple of things. We no longer have the test "string" do syntax, so we've gone back to the def test_name_with_underscores. (In practice, I like the test "string" syntax enough that I'd include it in a module.) We've also lost assert_dom_equal, we we are back to using the regular assert_equal.

But we've gained something big.

Since this test has no dependencies on Rails, we don't need the Rails environment—we could remove the require 'test_helper at the beginning of the file, and there would be no connection between this test and Rails.

Which means we could execute the test without running Rails. Which is great, because it's potentially much faster not to load Rails than to load Rails.

Hold that thought, we'll be coming back to it *the (as yet) unwritten chp.environment* **I don't know how to generate a cross reference to chp.environment** when we talk about Fast Tests.

Testing Mailers

Testing Rails mailers involves two separate bits of functionality: specifying whether the email gets sent as a result of some action, and specifying the contents of that email. Specifying whether the email gets sent often starts as part of a controller or integration test, while specifying the content has a lot in common with view testing. The somewhat indirect nature of the Rails ActionMailer makes testing email somewhat less obvious than it might be, but it's not that hard. We'll also look at a third party library that makes email testing easier.

Let's say, for sample purposes, that we want to send an email in our project system when a task is marked complete. We don't have a user model yet (we'll talk about users in this system when we get to Chapter 12, *Testing For Security*, on page 207), so we'll assume for the moment that all the emails go to some kind of common audit or monitoring address (insert `nsa.gov` joke of your choosing).

At this point, we have one of those weird, unique to book examples problems. Specifically, we haven't written very much of our task tracker site, basically just a project index and new project page. We don't have a list of tasks on a single project, let alone a way to mark a task completed. This would be a problem if we were to start with an external integration test using Capybara, that would need to interact with actual web pages.

Rather than walk you through writing a bunch of basically boilerplate web pages or having those pages magically insert themselves in the code repository, we're just going to focus on the controller action that handles marking a task complete. If we write tests for just that controller action, we don't need the rest of the application to exist. Everything I write here about dealing with mailers in a controller test also applies to a Capybara integration test.

With that handwaving out of the way, let's write our controller test. We take a moment to think about what we need:

- Given: We'll need one task that starts off incomplete
- When: A controller action. In a RESTful Rails interface, that action would be `TasksController#update`. Let's go with that. (Later, in *the (as yet) unwritten chp.javascript* **I don't know how to generate a cross reference to chp.javascript**, we'll make this an Ajax action.) The controller action has a `completed: true` parameter.
- Then: The task updates and an email is sent.

It occurs to me that there's a simple case to start with, where completed is not set and no email is sent. Writing that test first will let us write the structure of the method.

> **Prescription 18** When testing a boolean condition, make sure to write a test for both halves of the condition.

display/04/gatherer/test/controllers/tasks_controller_test.rb

```
Line 1 require 'test_helper'
-
- class TasksControllerTest < ActionController::TestCase
-
5  setup do
-    ActionMailer::Base.deliveries.clear
-  end
-
-  test "on update with no completion, no email is sent" do
10   task = Task.create!(title: "Write section on testing mailers",
-        size: 2)
-    patch :update, id: task.id, task: {size: 3}
-    assert_equal 0, ActionMailer::Base.deliveries.size
-  end
15
- end
```

Most of this is a standard controller test for update logic, but there are two lines specific to the code we are testing. On line 6, we're using the setup block to clear the ActionMailer::Base.deliveries object. Doing so ensures that the data structure holding the mailings is emptied. Otherwise, emails from other tests will linger and make your test results invalid.

We also need to ensure that in the config/environments/test.rb file we have the line config.action_mailer.delivery_method = :test, this should be done by default, and ensures that mail delivery in tests saves them to a data object whose behavior we can examine.

In the actual test, on line 13, we look at the mailer object, ActionMailer::Base.deliveries and confirm that no emails have been sent.

This test passes with this boilerplate controller method (plus a blank template in app/views/tasks/edit.html.erb):

display/04/gatherer/app/controllers/tasks_controller.rb

```
class TasksController < ApplicationController

  def update
    @task = Task.find(params[:id])
    if @task.update_attributes(params[:task].permit(:size))
```

```
      redirect_to @task, notice: "'project was successfully updated.'"
    else
      render action: 'edit'
    end
  end

  def show
    @task = Task.find(params[:id])
  end
end
```

Now, the test with the complete task that specifies email behavior.

display/05/gatherer/test/controllers/tasks_controller_test.rb

```
Line 1 test "on update with completion, send an email" do
  -   task = Task.create!(title: "Write section on testing mailers", size: 2)
  -   patch :update, id: task.id, task: {size: 3, completed: true}
  -   task.reload
    5 refute_nil task.completed_at
  -   assert_equal 1, ActionMailer::Base.deliveries.size
  -   email = ActionMailer::Base.deliveries.first
  -   assert_equal "A task has been completed", email.subject
  -   assert_equal ["monitor@tasks.com"], email.to
   10 assert_match /Write section on testing mailers/, email.body.to_s
  - end
```

Again, we simulate the call to the update method, this time with a pseudo-attribute completed, which we can assume indicates a checkbox of some kind. After that, on line 4, we reload the task object to take in the changes from the controller update, and verify that the completed_at attribute has, in fact, been updated.

Then we get to the mailer. We verify that one email has been sent, and then on line 7, we look at the actual email object, ActionMailer::Base.deliveries.first, and query it for it's subject, the list of addresses it's going to, and check that the body has the title of the task inside. A full description of this object can be found at http://guides.rubyonrails.org/action_mailer_basics.html, generally, the accessors have the names you would expected.

This task fails. The first fail is the check for the completed_at time, and the mailer will fail too.

The passing controller logic uses the completed param to trigger whether the task is updated and the mailer sent:

display/05/gatherer/app/controllers/tasks_controller.rb

```
class TasksController < ApplicationController

  def update
```

```
    @task = Task.find(params[:id])
    completed = params[:task].delete(:completed)
    params[:task][:completed_at] = Time.current if completed
    if @task.update_attributes(params[:task].permit(:size, :completed_at))
      TaskMailer.task_completed_email(@task).deliver if completed
      redirect_to @task, notice: "'project was successfully updated.'"
    else
      render action: 'edit'
    end
  end

  def show
    @task = Task.find(params[:id])
  end
end
```

Now all we need is the actual mailer. We can build a mailer from the command line using a Rails generator:

```
$ rails generate mailer TaskMailer
create  app/mailers/task_mailer.rb
      invoke  erb
      create    app/views/task_mailer
      invoke  test_unit
      create    test/mailers/task_mailer_test.rb
      create    test/mailers/previews/task_mailer_preview.rb
```

The mailer itself is straightforward, we need to take the task object, and set some mail variables.

display/05/gatherer/app/mailers/task_mailer.rb

```
class TaskMailer < ActionMailer::Base
  default from: "from@example.com"

  def task_completed_email(task)
    @task = task
    mail(to: "monitor@tasks.com", subject: "A task has been completed")
  end
end
```

And we need a template. We'll keep this super-simple:

display/05/gatherer/app/views/task_mailer/task_completed_email.text.erb

```
The task <%= @task.title %> was completed at <%= @task.completed_at.to_s %>

Thanks,

The Management
```

And the tests pass.

Outside of core Rails, The email-spec library provides a number of very useful helpers. For the most part, they are ways of performing the tests we've already examined with a slightly cleaner syntax. The library also provides the ability to follow a link in an email back to the site, which is very helpful for acceptance testing of user interactions that include email. The library's home is http://github.com/bmabey/email-spec.

Managing Controller and View Tests

Testing controllers and views is a very tricky part of Rails testing. Unlike model testing, which tends to be isolated to the particular model, or integration testing, which explicitly covers the entire stack, controller and view testing have boundaries that are more blurred. Controlling those boundaries is the difference between tests that run quickly and fail only when the logic being tested is incorrect, and tests that are slower and dependent on logic outside the test.

Ideally, controller tests are written so that they have minimal interaction with the model. There are costs to be balanced here. A controller action that has minimal contact with the model, and can therefore have that interaction stubbed, will often run faster and have fewer points of failure. On the other hand, the stubbing and additional classes that may be needed to mediate a controller and model interaction may feel overly complex, especially for boilerplate actions.

These days, I tend to be aggressive about moving controller logic that interacts with the model to some kind of action object that does not have Rails dependencies. The controller logic and controller testing then tends to be limited to correctly dispatching successful and failed actions. That said, many Rails developers, notably including David Heinemeier Hansson, find adding an extra layer of objects to be overkill and think that worry about slow tests is misplaced. I recommend you try both ways and see which one best suits your problem.

Focused view tests are possible in Rails, but overlap heavily with helper tests, logic placed in presenter objects, and integration testing.

CHAPTER 10

Driving Behavior With RSpec

RSpec is an alternative Ruby testing framework which encourages a Behavior-Driven Development (BDD) style of testing. What is the difference between BDD and TDD besides the first letter? Calling your process BDD signals a subtle but significant shift in how you approach a testing practice. Minitest and TDD use terms influenced by the idea of after the fact testing, like "assert", and "test". RSpec and BDD use terms, like "describe", "expect", and "specify", which suggest describing the behavior of code not yet written.

BDD emphasizes setting expectations about the code not yet written, rather than assertions about code in the past. Remembering our discussion of mock-object testing in Chapter 8, *Using Mock Objects*, on page 105, you might assume that BDD practice implies heavy use of mock objects. You'd be right, though more in terms of historical influence than current RSpec practice.

BDD and RSpec were created in part to combat the idea that the original TDD terminology, by emphasizing testing terms already in use, gave people the idea that the sole point of the process was to verify program correctness. Suggesting that TDD is about correctness may seem a minor point, except that the argument then had a tendency to morph into two somewhat more pernicious corollaries.

If the whole point of Test-Driven Development is to verify correctness, the argument goes…

- Then if the process isn't 100 percent perfect at proving correctness, it must be worthless.
- Then it doesn't matter when we write the tests; we can write them after we write the code.

The goal of TDD or BDD is to improve the quality of your code in many ways; verifying correctness is only one of them. And it makes a big difference when

you write the tests, because the process is much more effective when the tests come first. That's why Kent Beck, when originally naming the process explicitly called out *test-first*.

What's a Spec?

What do you call the things you write in an RSpec file? If you are used to TDD and Test::Unit, the temptation to call them tests can be overwhelming. However, as we've discussed, the BDD planning behind RSpec suggests it's better not to think of your RSpec code as tests, which are things happen after the fact. So, what are they?

The RSpec docs and code refer to the elements of RSpec as . Maybe I'm not going to the right parties, but I've never heard an actual person use that term. (I have, however, heard people use to refer to RSpec's version of contexts.) The term I hear most often is simply , as in: "I need to write some specs for that feature." I've tried to use that term in this section. But I suspect I'll slip up somewhere. Bear with me.

The important idea is that names like Test-Driven Development tended to lead people to the conclusion that the TDD process was mainly about verifying existing behavior. This led to a lot of frankly irritating online debates with people talking past each other because of different interpretations of what "testing" meant in "test-driven".

The term Behavior-Driven Development was coined to help win those arguments. Well, an argument like that can't really be won, but it the idea was to clarify terms. Where the word "test" implies verifying already written code, terms like "behavior" and "specify" imply describing the workings of code that has yet to be written. Ideally, then, discussing your process in BDD terms makes you more aware of the reason why you are using the process in the first place. Eventually, RSpec was created as a tool for implementing BDD in Ruby.

Getting Started With RSpec

RSpec is a larger framework than Minitest, so we'll focus on the most important parts for working with Rails here. We'll focus on how to write RSpec specs, and how RSpec encourages a slightly different style of testing. This chapter can only scratch the surface of what's possible with RSpec; for more details, check out *The RSpec Book [CADH09]*, which goes into more details, although it's now a full version out of date.

Installing RSpec

We're going to be talking about RSpec 3, which has some significant syntactical differences from previous versions of RSpec. We're going to largely ignore those differences to focus on only the new syntax.

To add RSpec to a Rails project, add the rspec-rails gem to your Gemfile:

```
gem 'rspec-rails', '3.0.1'
```

Installing the bundle causes many dependent gems to be added to the project, including rspec-core, rspec-expectations, and rspec-mocks.

Once the bundle is installed, we need to generate some installation files:

```
$ rails generate rspec:install
      create  .rspec
      create  spec
      create  spec/spec_helper.rb
      create  spec/rails_helper.rb
```

We now have

- The .rspec file, where RSpec run options go. The default currently sets three options, --color, which sets terminal output in color, --warnings, which turns on all Ruby warnings, and --require spec_helper, which ensures that the spec_helper file is always required. I *highly* recommend that you remove --warnings from this file, it results in a lot of irrelevant output that you can't control from third party libraries.
- The spec directory, which is where your specs go. RSpec does not automatically create subdirectories like controller and model on installation. Those are created by the generators when you use Rails generators build specs in those directories for the first time.
- Thespec_helper.rb and rails_helper.rb files. These are the RSpec analogy to test_helper.rb. The spec_helper.rb file contains general RSpec settings, while the rails_helper.rb file, which requires spec_helper contains settings dependent on the Rails environment being loaded. The idea is to make it easier to write specs that do not load Rails.

In addition, the rspec-rails gem does a couple of other things when loaded in the Rails project:

- RSpec adds a Rake file that resets the default Rake task to run the RSpec spec files instead of Minitest and also defines a number of subtasks such as spec:models to run part of the specs at once.
- RSpec sets itself up as the test framework for the purposes of future Rails generators. Later, when you set up, say, a generated model or resource,

RSpec's generators are automatically invoked to create appropriate spec files.

You run RSpec from the command line. I prefer to use the rspec command line option directly, though there are also rake tasks. You can use the .rspec file to add additional options that are attached to the command line whenever it is invoked—these options can also go in the spec_helper.rb file. RSpec has a variety of options for specifying subsets of tests to run, which we cover in *the (as yet) unwritten chp.environment*I don't know how to generate a cross reference to chp.environment.

RSpec in 10 Minutes

We're going to go through the Minitest tests that we wrote in Chapter 3, *Test-Driven Development Basics*, on page 13 and Chapter 4, *Test-Driven Rails*, on page 37 and convert them to RSpec. Here is a first pass at the initial tests that we wrote for our Project class. We're converting them into basic RSpec, which we will refine somewhat in a few steps. Since the code for all these specs is already written, we're just going to show RSpec.

rspec/01/gatherer/spec/models/project_spec.rb

```
require 'rails_helper'

describe Project do

  describe "with a new project" do
    before(:each) do
      @project = Project.new
    end

    it "knows that a project with no tasks is done" do
      expect(@project.done?).to be_truthy
    end

    it "knows that a project with an incomplete task is done" do
      task = Task.new
      @project.tasks << task
      expect(@project.done?).to be_falsy
    end
  end

end
```

You'll notice two major differences between RSpec and Minitest almost immediately. In Minitest, we need to define our tests inside a class that is a subclass of Minitest::Test. RSpec assumes that anything inside a file that ends in _spec.rb is RSpec, so there are no class declarations needed. Similarly, in

Minitest, individual tests are methods, while the RSpec file above does not declare any methods. RSpec is largely made up of calls to the describe and it methods, which each take blocks. When specs are run, RSpec causes those blocks to be executed in a particular order and in a particular context.

Shared setup with describe

RSpec makes it easy to have multiple contexts in a spec file that share all or part of the setup. Each of those contexts is declared by using the describe method or the context method, which is an exact alias. Each of these methods takes one argument, which is a string or a class constant, and then a block. If the argument is a class then the specifications in the block are expected to relate to that class. RSpec will use that information to make inferences about the code being exercised. (Stylistically, context is often used to when you have an inner block with multiple sibling blocks that test different situations: for example, an administrative user vs. a regular user.)

As we see in this example, describe calls can be nested. To see why we might want to do nest describe blocks, let's rewrite those two specs slightly:

rspec/02/gatherer/spec/models/project_spec.rb

```
require 'rails_helper'

describe Project do

  describe "with a new project" do
    before(:each) do
      @project = Project.new
    end

    it "knows that a project with no tasks is done" do
      expect(@project.done?).to be_truthy
    end

    it "knows that a project with an incomplete task is done" do
      task = Task.new
      @project.tasks << task
      expect(@project.done?).to be_falsy
    end

    describe "with a project that has one task" do
      before(:each) do
        @project.tasks << Task.new
      end

      it "knows a project is only done if all its tests are done" do
        expect(@project.done?).to be_falsy
        @project.tasks.first.mark_completed
```

```
        expect(@project.done?).to be_truthy
      end
    end
  end

end
```

In this version of the specs, we've added a third spec, and we've moved the common setup of creating a project to a before method. RSpec uses the before method for setup common to a group of specs. Teardown behavior is accomplished via before and after method. Both before and after take a single argument. If the argument is :each (which is also the default), the block is executed before or after each specification, as opposed to the much less often used :all, which indicates the block is executed once before or after all specifications in the common group are executed. (RSpec 3, in an effort to make the semantics clearer, defines before(:example) as an alias for before(:each) and before(:context) as an alias of before(:all)).

In moving common code to the before block, we've had to make @project an instance variable to ensure that it can be shared between the before block and the individual specs. In a page or so, we'll see a way to have initialization behavior and not have to make the variables instance variables. Support methods can be defined inside a describe block using normal Ruby def <methodname> syntax and can be used by any spec within the block.

When describe blocks are nested, all the before(:each) blocks are executed before each spec is executed. Outermost before blocks go first, innermost second. We've taken advantage of this behavior—our outermost describe block creates an @project variable, while the innermost one adds a task to the project. This kind of nesting is handy, allowing us to share the truly common parts of a test setup across a wide range of specs, while limiting more specialized setup only to the code which needs it.

The "It" Factor

In RSpec, you define your actual specs with the it method, which takes an optional string that documents the spec, and then a block which is the body of the spec. Unlike Minitest, the string argument is not used internally to identify the spec—you can have multiple specs with the same description string. RSpec also defines specify as an alias for it. Normally, it is used when the method takes a string argument and is used to give the method a readable natural-language name. For single-line tests in which a string description is unnecessary, specify is used to make the single line read more clearly, such as:

```
specify { expect(user.name).to eq("fred") }
```

RSpec has a handy mechanism for noting that a particular specification has not yet been implemented in the code. You can define an it method without a block. In this case, the test will appear in the RSpec output as “pending”.

```
it "bends steel in its bare hands"
```

You can temporarily mark a method as pending by changing it to xit. If you want to mark an entire describe block as pending, you can temporarily change the name to describe.

Or, you can use the method pending in the spec:

```
it "bends steel in its bare hands" do
  pending "not implemented yet
end
```

This test stops when the pending method is reached and returns its result as “pending.” (If the test fails before the pending is reached, the failure is treated normally.)

In RSpec 3, all pending specs are actually run, if there is code in the block part of the definition. The code is executed, with any failure there is treated as a “pending” result, rather than a failure result. However, if the code in the block passes, you’ll get an error that effectively means, “You said this was pending, but lo and behold, it works. Maybe it’s not actually pending anymore; please remove the pending block.”

If you want the spec to just not run, without testing for whether it works or not, the syntax above works, but use skip instead of pending.

Great “Expect”ations

RSpec allows you to make verifiable assumptions about state in your application through the use of expectations, which are activated using the expect method. Where Minitest would do something like

```
assert_equal(expected, actual)
```

RSpec syntax, in RSpec 3, orders the elements differently:

```
expect(actual).to eq(expected)
```

RSpec shifts the tone from an assertion, potentially implying already implemented behavior, to an expectation implying future behavior. The RSpec version, arguably, reads more smoothly, (though some strenuously dispute this). We’ll see some other tricks RSpec uses to make matchers read like

natural language. I find it difficult to remember the order the expected and actual parameters are supposed to have in the assert_equal version, but I have very little trouble remembering the order in the RSpec version.

It's worth breaking down RSpec expectations to see a little bit about how they work internally. Let's look at one of our expectations from our earlier specs.

```
expect(@project.done?).to be_truthy
```

Tracing this expectation takes a few steps. First is the expect call itself, expect(@project.done?). The expect method is defined by RSpec and takes in any object and returns a special RSpec proxy object called an ExpectationTarget. The ExpectationTarget holds on to the object that was the argument to expect, and itself responds to two messages, to, and not_to. (Okay, technically three messages, since to_not exists as an alias.) Both to and not_to are ordinary Ruby methods that expect as an argument an RSpec Matcher. There's nothing special about an RSpec matcher, at base it's just an object that responds to a matches? method. There are several pre-defined matchers, and you can write your own.

In our case be_truthy is a method defined by RSpec to return the BeTruthy matcher. You could get the same behavior with expect(@project.done?).to(RSpec::BuiltIn::BeTruthy.new), but you probably would agree that the idiomatic version reads better.

The ExpectationTarget is now holding on to two objects, the object being matched, in our case @project.done?, and the matcher be_truthy. When the spec is executed, RSpec calls the matches? method on the matcher with the object being matched as an argument. If the expectation uses to, then the expectation passes if matches? is true. If it uses not_to, then the expectation passes if matches? is false. The following diagram traces the execution of the matcher:

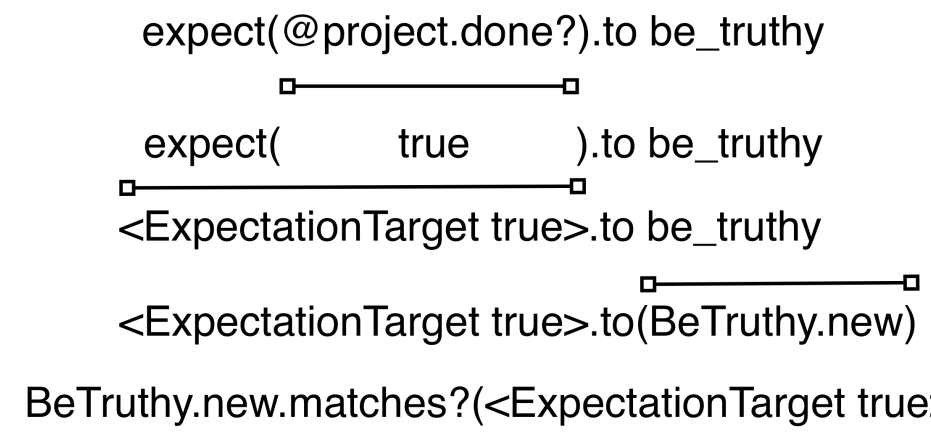


Figure 4—RSpec Matcher Execution

RSpec predefines a number of matchers, here's a list of the most useful ones, a full list can be found at https://relishapp.com/rspec/rspec-expectations/v/3-0/docs/built-in-matchers

```
expect(actual).to all(matcher)
expect(actual).to be_truthy
expect(actual).to be_falsy
expect(actual).to be_nil
expect(actual).to be_between(min, max)
expect(actual).to be_within(delta).of(actual)
expect(actual).to change(receiver, message, &block)
expect(actual).to contain_exactly(expected)
expect(actual).to eq(actual)
expect(actual).to include(*expected)
expect(actual).to match(regex)
expect(actual).to raise_error(exception)
expect(actual).to satisfy { block }
```

Most of these mean what they appear to say. The `change` matcher passes if the value of `receiver.message` changes when the block is evaluated. The `contain_exactly` matcher is true if the expected array and the actual array contain the same elements, regardless of order. The `satisfy` matcher passes if the block evaluates to `true`. Any of these can be negated by using `not_to` instead of `to`.

RSpec 3 also allows you to chain multiple matchers using `and` and `or`, as in `expect(actual).to include("a").and match(/.*3.*/)`, or `expect(actual).to eq(3).or eq(5)`.

You can also pass matchers as arguments to other matchers, or compose matchers to handle an data structures using match. The following two snippets are equivalent

```
expect(actual[0]).to eq(5)
expect(actual[1]).to eq(7)

expect(actual).to match([an_object_eq_to(5), an_object_eq_to(7)
```

In the bottom snippet, the elements of the array actual are individually matched against an_object_eq_to(5) and an_object_eq_to(5), which are aliases to eq(5) and eq(7). Most of the built-in matchers have aliases to make them more natural language like when used as arguments, see http://rubydoc.info/github/rspec/rspec-expectations/RSpec/Matchers for a full list. The all matcher also takes one of these matcher arguments and applies it to every element in an array, expect(actual).to all(be_truthy).

One of my favorite bits of RSpec is an implicit matcher that RSpec creates by name-mangling if you give it a matcher it doesn't recognize. Any matcher of the form be_whatever or be_a_whatever assumes an associated whatever? method—with question mark—on the actual object and calls it. The matcher passes if the predicate method passes. (Or, if the matcher was called via not_to, if the predicate method fails.)

In the specs we've been looking at, we invoked expectations like expect(@project.done?).to be_truthy. Since done? is a predicate method, we can rewrite those specs to be a little more direct.

rspec/03/gatherer/spec/models/project_spec.rb

```
it "knows that a project with no tasks is done" do
  expect(@project).to be_done
end

it "knows that a project with an incomplete task is done" do
  task = Task.new
  @project.tasks << task
  expect(@project).not_to be_done
end
```

Now, we can write expect(@project).to be_done, which almost reads as simple, clear, natural language. Often, it's easier to add a predicate method to your object than it is to create a custom matcher in RSpec.

A lot of complexity goes into making the language clear. In addition to allowing be_a and be_an, if the predicate method is in present tense, like matches?, you can write the expectation as be_a_match. In this case, RSpec will look for match?, and then form matches? if it can't find match?.

Similarly, if the predicate method starts with has, RSpec allows your matcher to start with have for readability so your tests don't look like they have been written by LOLCats, allowing expect(actual).to have_key(:id) rather than expect(actual).to has_key(:id).

I love this language inflection because I feel like it allows me to write tests that are succinct and clear. Others find the internal complexity of this feature to be too high a price to pay for the natural language tests (some people also really don't like the natural language syntax).

RSpec, with parenthesis

RSpec is supposed to be a Domain Specific Language (DSL) for testing, and as a result it has both a lot of unusual metaprogramming and a slightly different set of conventions about parenthesis and blocks than normal Ruby. Because RSpec looks, at first glance, different that typical Ruby, I often find it useful for people coming to RSpec for the first time to fully parenthesize the method calls, so as to be better able to follow the code flow. For our first test, the fully parenthesized version looks like this:

```
describe(Project) do
  describe("with a new project") do
    it("knows that a project with no tasks is done") do
      project = Project.new
      expect(project).to(be_done)
    end
  end
end
```

The parenthesis makes it clearer that describe and it are just ordinary methods, and that to is a method that takes the matcher be_done as an argument. If you find a piece of RSpec code confusing, try to add the parenthesis back until it becomes clearer.

Mocking RSpec

Mock objects have always been a big part of the RSpec style, and RSpec defines its own mock object framework, similar to Mocha, which we saw in Chapter 8, *Using Mock Objects*, on page 105. (At least one early reviewer felt that everybody should use RSpec's mocks over Mocha even in Minitest…) In particular, RSpec users often strive to use mock objects to make each individual spec independent to other specs such that any individual flaw in the application breaks exactly one spec. Like many design goals, this one is probably honored more in the breach, but the basic idea of isolating a spec

to a particular method, class, or layer is very important in using RSpec effectively.

If you prefer a different mock object package, the default spec_helper file for Rails contains some lines that allow you to change which mock framework you are using. I don't recommend doing so, you should stick with RSpec's mocks, which are designed to be used within RSpec and which have a very full feature set.

Like Mocha, RSpec allows you to create bare mock objects. In RSpec, this is done with the double method:

```
mock = double("name")
```

Without any further description, the mock object does not know about any methods and will return an expectation error if it is called. You can get it to silently return nil when called by calling the method as_null_object. This is often done in the original declaration:

```
mock = double("an object").as_null_object
```

If you want to specify methods on an RSpec mock object, you have a more and less verbose option. The following sets of code are identical:

```
mock1 = double(name: "Fred")

mock2 = double
allow(mock2).to receive(name: "Fred")

mock3 = double
allow(mock3).to receive(name).and_return("Fred")
```

In all three cases, the object created can receive name, and will return Fred. I prefer the third and most verbose form, in part because it's a little more flexible against changes in how the test might work over time.

That's the equivalent to a Mocha stub, but what if we want to create a doubled object that specifies behavior, and will fail if the method is not called. In RSpec, we accomplish that strict behavior by replacing allow with expect:

```
mock = double
expect(mock).to receive(:name).and_return("Fred")
```

In this case, not only will mock.name return Fred, but if mock.name is not called during the test, then the test will fail.

As with Mocha, you can define partial mocks on existing objects:

rspec/04/gatherer/spec/models/project_spec.rb

```
describe "with stubs" do
```

```
  it "can stub an instance and return a value" do
    project = Project.new(name: "Project Greenlight")
    allow(project).to receive(:name).and_return("Fred")
    expect(project.name).to eq("Fred")
  end

  it "can mock an instance and expect a value" do
    project = Project.new(name: "Project Greenlight")
    expect(project).to receive(:name).and_return("Fred")
    project.name
  end

  it "can spy on an instance" do
    project = Project.new(name: "Project Greenlight")
    allow(project).to receive(:name).and_return("Fred")
    project.name
    expect(project).to have_received(:name)
  end
end
```

As with pure mocks, the allow method creates a stub, while the expects method creates a stub but also fails if the stub is not invoked during the test.

The last example is interesting, though, and is the RSpec implementation of a test spy. In that example, we set up the double (allow(project).to receive(:name).and_return("Fred")), then run our code without setting a mock expectation. After the code is run, we then set the expectation, expect(project).to have_received(:name). This form is often easier to read, since it corresponds to the Given/When/Then structure common to tests, and it reduces the "there is a mysterious expectation in this code that I can't see" problem that can happen when you have tests with mock expectations.

If you need to specify multiple messages in one shot, you can use receive_messages, which takes a list of key/value pairs:

```
allow(mock).to receive_messages(name: "Fred", role: "admin")
```

You can also collapse a series of messages into a single stub using receive_message_chain. The following two lines are equivalent:

```
allow(user).to receive(:friend).and_return(double(name: double(first: "Noel")))

allow(user).to receive(:friend, :name, :first).and return("Noel")
```

RSpec has another nice trick up its sleeve:

```
user_mock = instance_double("User")
allow(user_mock).to_receive(:name).and_return("Fred")
```

The instance_double here adds an additional, and important specification to the mock object. This test assumes that you are trying to double a User object and will fail if User does not define name. (Technically, it will fail if an instance of user does not respond_to?(:name) . As you may recall from our earlier discussion, one of the occupational hazards of mock objects is having the mock be out of sync with the actual API of the real object.

The first argument to instance_double is the string name of the class being mimicked. After that, you can treat it just like double, including key/value pair arguments, further chained message calls, or later including the double in a spy. In addition to instance_double, RSpec defines class_double, which also takes a class name as a string, but verifies against class methods, and object_double which takes an instance and verifies against the methods that instance responds to.

One catch—in order for this verification to work, you must include this block in your spec_helper.rb configuration:

```
config.mock_with :rspec do |mocks|
  mocks.verify_doubled_constant_names = true
  mocks.verify_partial_doubles = true
end
```

The verify_doubled_constant_names is what we've been talking about with instance_double and its siblings. And verify_partial_doubles is the same thing applied to partial doubles. In other words, given the project specs we already saw:

```
it "can mock an instance and expect a value" do
  project = Project.new(name: "Project Greenlight")
  expect(project).to receive(:name).and_return("Fred")
  project.name
end
```

In this case, even though we haven't explicitly declared project to be a double, RSpec will still verify that the stubbed method, name, actually exists in the project instance, and if it does not, the spec will fail.

You should have these validations turned on all the time, to get this extra level of safety in using your mocks. If you are doing any kind of testing that uses mock objects to bridge a gap between layers of your application—as we saw in controller testing and as we will see in testing external services—then having this verification prevents an important problem where the object API changes, but the test, dependent on a mock object, soldiers on and passes anyway.

The argument to assert and expect can be an instance or a class. RSpec has its own syntax for making it so that any instance of a class responds to the same stubbed method:

```
allow_any_instance_of(Project).to receive(:save).and_return(false)
```

The method allow_any_instance_of takes a class argument, and then applies the chained methods of that statement to any instance of the class created subsequently. Although this is occasionally useful, especially when working directly with the Rails framework, in general, the use of allow_any_instance_of could mean that different sections of your code are too tightly connected, and you might want to separate object creation from object usage.

As with Mocha, the assert and expect method chains allow for additional methods to filter the incoming arguments and make more specific claims about application behavior. The most common is with:

```
allow(task).to receive(:mark_completed).with(Date.today).and_return(true)
```

The argument to the with method is typically the value or values that you expect to be passed to the method being stubbed. If you specify a with value, then any non-matching value passed to the method results in a test failure, even if you are using a stub and not a true mock. In addition to actual objects, RSpec allows the argument to with to be one of a series of argument matchers, allowing the argument to be, say, any instance of a particular type. I don't find these tremendously useful, but you can find full documentation at https://relishapp.com/rspec/rspec-mocks/v/3-0/docs/argument-matchers.

The default expectation for RSpec is that a mocked method will be called exactly once, an expectation you can make explicit by including the method once in the mock method chain. Again, as with Mocha, other methods such as twice, at_least, at_most, and exactly(x).times can change the default behavior:

```
allow(user).to receive(:friend_count).exactly(3).times.and_return(7)
```

Let RSpec Eat Cake

There is one more RSpec feature that I commonly use when writing RSpec that we haven't seen yet. RSpec's let statement makes setting up data a little cleaner than a before statement.

rspec/05/gatherer/spec/models/project_spec.rb

```
describe "with a new project" do
  let(:project) { Project.new }
  let(:task) { Task.new }

  it "knows that a project with no tasks is done" do
```

```
    expect(project).to be_done
  end

  it "knows that a project with an incomplete task is done" do
    project.tasks << task
    expect(project).not_to be_done
  end
```

Using let, you can make a variable available within the current describe, without having to place it inside the before block, and without having to make it an instance variable. Each let method call takes a symbol argument and a block. The symbol can then be called as if it was a local variable: the first call to the symbol invokes the block and caches the result; subsequent calls return the same result without reinvoking the block.

In this example, we use let twice, first to describe a project and then to describe a task. We can then use project and task in the body of the specs. Even though task isn't used in the first spec, that's perfectly fine—RSpec only lazily invokes the let block when the variable is used.

In essence, a let() call is syntactic sugar for defining a method and memoizing the result, like this:

```
def me
  @me ||= User.new(name: "Noel")
end
```

The main gotcha here is that the let block isn't executed unless it's invoked. That's often a good thing, since your test won't spend time creating unused objects. You can get in trouble sometimes if you expect that the object already exists. For a contrived problem case, note that this example will fail, since the two let blocks are never invoked:

```
describe "user behavior"
  let(:me) { User.new(:name => "Noel") }
  let(:you) { User.new(:name => "Erin") }
  specify { User.count.should == 2 }
end
```

Luckily, RSpec does provides a mechanism in these cases where an item must be present even though it is never invoked by name. It's called let! with a "bang" for "Block Always Needs Gathering", which I just made up and only makes a tiny amount of sense, but you probably won't forget it.

I use let all the time. I like that it separates the definition of each variable, I like that it encourages concise initializations, and the word let allows me to pretend I'm writing in Scheme for a brief moment.

RSpec On Rails

RSpec is not a library created for Rails. It can be and is used on many Ruby projects that are not Rails. The rspec-rails gem, however, augments RSpec with a few tools to make RSpec more valuable for us Rails developers. These features include:

- Rails generators that plug into the Railties system and create RSpec files when you generate new models, controllers, or the like.
- Support for Rails features like fixtures.
- Custom, Rails-specific, example group classes that are automatically associated with the specific RSpec directory that the file is in. So, files in the spec/controllers directory automatically have RSpec controller functionality.
- Stub support for ActiveRecord objects
- Matchers that mimic Rails Minitest assertions.

We've covered installing RSpec for Rails—put it in the Gemfile, run the generator, and go. So let's take a tour of RSpec's Rails features, including the RSpec version of some of the Minitest tests we've written.

RSpec defines a number of different custom example groups where Rails-specfic behavior is added. Historically, RSpec determined which group a spec belonged to by it's location in the file system. Files in spec/models were model specs, spec/controllers, controller specs and so on.

In RSpec 3, this behavior is covered by a configuration option, and the older behavior is the default, enabled by the config line config.infer_spec_type_from_file_location!, which is part of the new default RSpec 3 Rails rails_helper.rb. If you remove this configuration option, you can specify the type of a describe block using RSpec metadata, as in

```
describe ProjectsController, type: :controller do
end
```

RSpec and Models

The RSpec specs we've seen so far are model specs, what with us running them against ActiveRecord models. RSpec expects model spec files to have an outer describe block whose argument is the model class.

RSpec provides a couple of custom matchers for ActiveRecord:

```
expect(Project).to have(:no).records
expect(Project).to have(1).record
expect(Project.where(name: "Project Runway").to have(1).record
```

```
project.name == ""
expect(project).to have(1).error_on(:name)
```

The have(x).records matcher tests how many rows there are in the database for that model, it's a shortcut for expect(Project.all.size).to be(x). And, yes, RSpec does define :no to be an alias of zero and allows you to use singular or plural record as needed.

The error_on (naturally, errors_on also works) matcher tests for ActiveRecord validation errors attached to the given attribute.

RSpec provides two special methods to create mocked versions of ActiveRecord objects, stub_model and mock_model. As of RSpec 3.0, these methods are defined in a separate gem, called rspec-activemodel-mocks, which you must include in your Gemfile separately. The two methods share a goal, but are implemented differently. The mock_model method takes two arguments—the name or string name of an ActiveRecord subclass (technically, any class that includes the ActiveRecord::Naming module will work), and a key/value pair of methods and values:

```
mock_model("Project", name: "Runway")
```

The return value of mock_model is an RSpec double, with a number of ActiveRecord-ish methods pre-defined, including a fake id, valid? equal to true, persisted? accurately reflecting whether there's an id. Since you can pass a string name, the ActiveRecord model does not have to exist for the test to work, making mock_model a potentially valuable choice for integration tests involving code that is still in the works. If you override save or update_attributes to be false, the mocked model will stub errors.empty? with a value consistent to the save failing.

On the other hand, stub_model, also takes two arguments, but the first one must be a class constant name, not a string. The second argument is still a set of key/value pairs representing stubbed values:

```
stub_model(Project, name: "Runway")
```

The return value of stub_model is an real ActiveRecord object, with database-pointing methods such as save blocked so that the model can not touch the database. Unless passed in the second argument, an id is generated for the object by RSpec. If you don't want the stubbed model to act as though it has already been saved, then send as_new_record to it. The value returned by stub_model will behave more like the actual ActiveRecord, but the record must

already exist for it to work. It's especially useful in view testing where RSpec allows you to bypass controllers and test view templates in isolation.

Controllers in RSpec

In Minitest, as we saw in Chapter 9, *Testing Controllers And Views*, on page 125, controllers and views are tested together. RSpec splits the two apart, with files in the spec/controllers directory specifying controller behavior without calling the views and with separate files in the spec/views directory for view testing. The idea is to keep things as separate and independent as possible. The controller's job is to marshal together objects to pass to the view and that behavior can be specified independent of whether the view properly puts it on the screen. There's also a speed benefit in not running the display engine where it isn't needed.

In RSpec, files in spec/controllers or of type: :controller should have an outer describe block whose class is one of the Rails controllers in your application. That controller will be the target of actions called from the specs inside that block.

RSpec controller groups define a few matchers that effectively wrap the assertions defined by ActionController::TestCase.

```
expect(response).to redirect_to(something_redirectable)
expect(response).to render_template(template)
```

Instance variables assigned by the controller are accessible via the assigns method, as in expect(assigns(:project)).not_to be_nil. You can also get at the session and flash similarly.

In an RSpec controller spec, the same get, post, and other action methods work the same way as they do in ActionController::TestCase. The only difference is that, in RSpec, the test ends when the controller method is done, the view is not executed (although the view template must actually exist). If, for some reason, you actually want to run the views in a controller test—which I do not recommend—calling the method render_views inside a describe block turns view rendering on for the duration of that block.

What does this all look like in practice? Here are the specs for ProjectsController#create copied from Minitest to RSpec—including a version that uses mocks and one that doesn't.

rspec/05/gatherer/spec/controllers/projects_controller_spec.rb

```
require 'rails_helper'

describe ProjectsController do
```

```
  describe "#create" do
    it "creates a project" do
      post :create, project: {name: "Runway", tasks: "start something:2"}
      expect(response).to redirect_to(projects_path)
      expect(assigns(:action).project.name).to eq("Runway")
    end

    it "creates a project with mocks" do
      fake_project = double(create: true)
      allow(CreatesProject).to receive(:new)
          .with(name: "Runway", task_string: "start something:2")
          .and_return(fake_project)
      post :create, project: {name: "Runway", tasks: "start something:2"}
      expect(CreatesProject).to have_received(:new)
      expect(response).to redirect_to(projects_path)
      expect(assigns(:action)).not_to be_nil
    end

    it "fails gracefully" do
      allow_any_instance_of(Project).to receive(:save).and_return(false)
      expect { post :create, :project => {:name => 'Project Runway'} }
        .not_to change { Project.count }
    end
  end

end
```

A few things to discuss here:

- The outermost describe block directly references the ProjectsController, the innermost describe block directly references the action under test. The outermost is required by RSpec and is used to identify the target of action calls like post :create, the inner naming is a useful convention, common but not required. Often the success and failure cases would be split into separate groups, not necessary in this case because the setup is the same for each.
- The post method works exactly as it's ActiveController::TestCase counterpart.
- Here's a use of allow_any_instance_of. We are using allow_any_instance_of to allow the controller to create a different instance than we have access to in the test, but still be able to stub the failure condition. Alternately, we could stub Project.new to return an instance of project that itself has save stubbed out. (In this case, you could also do something similar stubbing the CreatesProject action object).
- The bottom test also uses the change matcher to assert that no new objects are created as a result of the controller action. The syntax is 'expect {an activity block}.not_to change { a value }. The value block is evaluated, then

the activity block is evaluated, then the value block again. The first and last values of the value block are compared to determine if the matcher passes.

These tests are somewhat more verbose than their Minitest equivalents, but may be more expressive.

RSpec and Views

RSpec also allows you to specify views independent of controllers. The RSpec convention is to place view tests in the spec/views folder, with one spec file to a view file, so the view in app/views/projects/index.html.erb is specified in spec/views/projects/index.html.erb_spec.rb.

I rarely write these tests. I find the file structure hard to maintain, and what logic I do have in views is often tested between objects like presenters and integration tests. In general, I find full TDD on views difficult—I often have to see a scratch implementation of a view before I know exactly what to test. That said, they are surprisingly easy to write, because they have no dependency on any other part of the code.

Here's the RSpec translation of the view test we wrote about the project index page showing different CSS classes for on schedule and behind schedule projects:

rspec/05/gatherer/spec/views/projects/index.html.erb_spec.rb

```
require 'spec_helper'

describe "projects/index" do
  let(:completed_task) { Task.create!(completed_at: 1.day.ago, size: 1) }
  let(:on_schedule) { Project.create!(due_date: 1.year.from_now,
      name: "On Schedule", tasks: [completed_task]) }
  let(:incomplete_task) { Task.create!(size: 1) }
  let(:behind_schedule) { Project.create!(due_date: 1.day.from_now,
      name: "Behind Schedule", tasks: [incomplete_task]) }

  it "renders the index page with correct dom elements" do
    @projects = ProjectPresenter.from_project_list(
      [on_schedule, behind_schedule])
    render
    expect(rendered).to have_selector(
        "#project_#{on_schedule.id} .on_schedule")
    expect(rendered).to have_selector(
        "#project_#{behind_schedule.id} .behind_schedule")
  end
end
```

I cheated here in one respect — the have_selector matcher is not part of core RSpec, it's actually part of Capybara, so we need Capybara in the Gemfile (gem 'capybara') and in the spec/rails_helper.rb file, up near the top, we need, require 'capybara/rspec'. But have_selector is so useful, and Capybara is useful in its own right, that it's not much of a project burden to include them.

What does this view test do? Let's look at this in terms of the Given, the When, and the Then.

First we create our given: the data. We use let to create a bunch of objects, the on schedule project and task and the behind schedule project and task. The objects need to be in the database so that the Rails associations all work. But, there are alternate options here to make the test faster, including creating projects and stubbing the on_schedule? method, or using FactoryGirl.build_stubbed to create real objects without having to save them in the database.

In the spec itself, we set the @projects variable. We're testing the view in isolation, so we don't have a controller to set this up for us, we need to create the instance variables that will be given to the view. Since we're using presenter objects in the real code, we need to pass our created projects through the ProjectPresenter.from_project_list method.

Our "when" section is one line, render, which tells RSpec to render the view. Which view? The view specified by the outermost describe block. In our case, it is projects/index—Rails will connect that to the index.html.erb file that's in the file base. Alternately, you can explicitly pass the view as an argument, render template: "projects/index". If the view is a partial, you need to specify that, render partial: "projects/data_row. An optional locals argument with key/value pairs specifies local variables to the partial. As with the regular Rails render method, you can use a shortcut of the form render "projects/data_row, project: @project.

All Rails helpers are loaded. If you want to stub one of their values, the helper methods are accessible via a view object, as in view.stub(:current_user).and_return(User.new). You can also stub a partial that you don't want to render using stub_template, which takes a key/value pair where the key is the exact file name of the partial, and the value is the string you want returned in place of rendering the partial.

In the "then" portion of the spec, the rendered text is available via the method rendered. You can then use any RSpec matcher to set expectations on that value. In this test, we use have_selector, but you can also use match to do a simple regular expression match.

RSpec Routing Tests

RSpec allows for routing tests. By default, they go in the spec/routing directory. There's one custom matcher, route_to, which is somewhat similar to the Rails assert_routing. Here's what our routing specs look like in RSpec:

rspec/05/gatherer/spec/routing/project_routing_spec.rb

```
require 'spec_helper'

describe "project routing" do
  it "routes projects" do
    expect(get: "/projects").to route_to(
        controller: "projects", action: "index")
    expect(post: "/projects").to route_to(
        controller: "projects", action: "create")
    expect(get: "/projects/new").to route_to(
        controller: "projects", action: "new")
    expect(get: "/projects/1").to route_to(
        controller: "projects", action: "show", id: "1")
    expect(get: "/projects/1/edit").to route_to(
        controller: "projects", action: "edit", id: "1")
    expect(patch: "/projects/1").to route_to(
        controller: "projects", action: "update", id: "1")
    expect(delete: "/projects/1").to route_to(
        controller: "projects", action: "destroy", id: "1")
  end
end
```

All of these are using the same form. The argument to expect is a key/value pair where the key is the HTTP verb and the value is the string form of the route. The argument to route_to is a set of key/value pairs where the keys are the parts of the calculated route, including controller, action, and what have you and the values are, well, the values. The route_to matcher tests the routes in both directions.

RSpec also provides a be_routable method, which is designed to be used in the negative to show that a specific path—like, say, the Rails default—is not recognized:

```
expect(get: "/projects/search/fred").not_to be_routable
```

RSpec and Helpers

RSpec helper tests go in spec/helpers. There's not a whole lot of special magic here, just a helper object that you use to call your helper methods.

rspec/05/gatherer/spec/helpers/project_helpers_spec.rb

```
require 'spec_helper'
```

```
describe ProjectsHelper do
  let(:project) { Project.new(name: "Project Runway")}

  it "returns a project name with status info" do
    allow(project).to receive(:on_schedule?).and_return(true)
    expected = '<span class="on_schedule">Project Runway</span>'
    expect(helper.name_with_status(project)).to eq(expected)
  end

  it "returns a project name with status info" do
    allow(project).to receive(:on_schedule?).and_return(false)
    expected = '<span class="behind_schedule">Project Runway</span>'
    expect(helper.name_with_status(project)).to eq(expected)
  end
end
```

If, for some reason, your helper method requires a specific instance variable to be set, cut that out immediately, it's a bad idea. However, if you must, and you want to test it in RSpec, use the assigns method, as in assigns(:project) = Project.new.

Write Your Own RSpec Matchers

RSpec's built-in matchers are flexible, but sometimes you have behavior patterns you specify multiple times in your code and the existing matchers don't quite cover it. Sometimes this is because the specification requires multiple steps, or sometimes the generic matcher doesn't quite match the intent of the code.

RSpec provides tools for creating your own custom matchers to cover just such an eventuality. A basic matcher is really simple. Let's say we wanted a custom matcher to measure project size in points. Rather than say expect(project.size).to eq(5) we'd rather say expect(project).to be_size(5). It's a little contrived, but work with me.

Anything in the spec/support folder is automatically imported when RSpec starts, making it a great place to put custom matchers like this one:

rspec/05/gatherer/spec/support/size_matcher.rb

```
RSpec::Matchers.define :have_size do |expected|
  match do |actual|
    actual.total_size == expected
  end
end
```

The definition starts with a call to RSpec::Matchers.define with a single argument, the name of the matcher, and a block. The block takes an argument which

will eventually be the expected value—which is to say the value passed to the matcher when called.

Inside the block, the match method is called with a block. That block takes one argument, actual, which is the value passed to expect when the matcher is invoked. The block is expected to do something with the expected and actual values and return true if they match by the definition of the matcher being written. In this case, we're calling total_size on the actual value, presumably a Project, and comparing it to the expected value for an equality match. If the matcher takes multiple expected arguments, than then the outer block definition should name all the arguments:define :have_sizes do |first, second|.

Remember the expected value is the value defined by the test, and the actual value is the value defined by the code. The form the matcher gets called will be

```
expect(actual_value).to have_size(expected_value)
```

We can then use this in a test just like any other matcher:

rspec/05/gatherer/spec/models/project_spec.rb

```
describe "with a custom matcher" do
  let(:project) { Project.new }
  let(:task) { Task.new(size: 3) }

  it "uses the custom matcher" do
    project.tasks << task
    expect(project).to have_size(3)
    expect(project).not_to have_size(5)
  end
end
```

In the first call, 3 is the expected value and project is the actual value, in the second call, we're using not_to to negate the matcher, and 5 is the expected value.

There are, of course, additional options. There are other methods you can call inside the define block to further specify behavior. For instance, you can override the messages RSpec will use when the matcher is displayed.

rspec/06/gatherer/spec/support/size_matcher.rb

```
RSpec::Matchers.define :have_size do |expected|
  match do |actual|
    actual.total_size == expected
  end

  description do
    "have tasks totaling #{expected} points"
  end
```

```
  failure_message do |actual|
    "expected project #{project.name} to have size #{actual}"
  end

  failure_message_when_negated do |actual|
    "expected project #{project.name} not to have size #{acutal}"
  end
end
```

We're using three different customizations here, description takes no arguments and returns a string that is used by RSpec formatters that display the matcher's description. The other two are displayed by RSpec when the matcher fails, with failure_message being printed when a to match fails, and failure_message_on_negation being printed when a not_to match fails. You can also adjust the failure message by calling the method diffable in the matcher block, no arguments, no block. Effectively, you are declaring the matcher to be diffable, which means that on failure the expected and actual arguments are displayed in a diff format. RSpec uses this in the string and array matcher built-in matchers.

If you want to be able to chain additional methods after the initial matcher to specify further arguments, then you use the chain method inside the matcher block:

rspec/07/gatherer/spec/support/size_matcher.rb

```
RSpec::Matchers.define :have_size do |expected|
  match do |actual|
    size_to_check = @incomplete? actual.remaining_size : actual.total_size
    size_to_check == expected
  end

  description do
    "have tasks totaling #{expected} points"
  end

  failure_message do |actual|
    "expected project #{project.name} to have size #{actual}"
  end

  failure_message_when_negated do |actual|
    "expected project #{project.name} not to have size #{acutal}"
  end

  chain :for_incomplete_tasks_only do
    @incomplete = true
  end
end
```

The chain method takes an argument and a block, with the argument being the name of the method you want to chain. Any arguments to that method become arguments to the block—in our case the method has no arguments. Typically, inside a chained method you set instance variables, which are then referenced used inside the match method to affect the match. Our for_incomplete_tasks_only method sets a flag, which is used to determine how to query the project for the size being tested.

The new method can then be chained onto the matcher:

rspec/07/gatherer/spec/models/project_spec.rb
```
describe "with a custom matcher" do
  let(:project) { Project.new }
  let(:completed_task) { Task.new(size: 3, completed_at: 1.hour.ago) }
  let(:incompleted_task) { Task.new(size: 2) }

  it "uses the custom matcher" do
    project.tasks = [completed_task, incompleted_task]
    expect(project).to have_size(5)
    expect(project).to have_size(2).for_incomplete_tasks_only
  end
end
```

RSpec and Sharing

RSpec has a powerful mechanism for sharing specs across multiple objects that have common functionality, simply called the “shared example”.

The most common use case or a shared example is some kind of interface or protocol that is shared by multiple classes, and which you need to test separately for each one. Common behavior, but disparate implementations.

There are two parts to shared examples in RSpec: the definition and the usage. Shared examples must be defined before they are used. In a Rails context, the easiest way to do that is to put the shared examples inside the spec/support directory, since the Rails default spec_helper.rb file loads everything in that directory before any specs are run.

Let’s create a contrived example, suggesting that we want projects and tasks to respond to a similar set of adjectives about their size. We create a shared group with the method shared_examples taking a string argument and a block.

rspec/07/gatherer/spec/support/size_group.rb
```
shared_examples "sizable" do
  let(:instance) { described_class.new }

  it "knows a one point story is small" do
    allow(instance).to receive(:size).and_return(1)
```

```
    expect(instance).to be_small
  end

  it "knows a 5 point story is epic" do
    allow(instance).to receive(:size).and_return(5)
    expect(instance).to be_epic
  end
end
```

The block inside the shared_examples method is very similar to a describe block. Inside, you can use let, or it to define specs. The only unusual thing about this block is that rather than create an object explicitly, our let statement at the top is creating a generic instance using the described_class, which is the class referenced by the outermost describe block when the shared example group is used. We're also "setting" the size via a mock, on the theory that, while tasks have a setter for their size, a project doesn't.

To use the shared example group, all you need to do is declare it. RSpec defines multiple equivalent methods for doing so (and even allows you to define your own) one of which is it_should_behave_like followed by the name of the group. Here's what that looks like in the task_spec file.

rspec/07/gatherer/spec/models/task_spec.rb

```
require 'spec_helper'

describe Task do
  it_should_behave_like "sizable"
end
```

When task_spec is executed, the shared group is invoked with Task as the described_class, and the test will look for the small? and epic? methods to pass the test. For the record, the following code will pass.

rspec/07/gatherer/app/models/task.rb

```
def epic?
  size >= 5
end

def small?
  size <= 1
end
```

There are a couple of other ways to invoke a shared example group—RSpec defines synonymous methods include_examples and it_behaves_like. You can also use RSpec metadata, which we haven't discussed, but will in *the (as yet) unwritten chp.environment*I don't know how to generate a cross reference to chp.environment.

When invoking the group, you can also have the it_should_behave_like method take a block argument. Inside that block, you can use let statements to define variables, which are then visible to the shared example specs. In other words, an alternative to creating an instance with described_class is to place the burden on the calling spec to create a variable and give it an appropriate name in the it block.

Data About Metadata

RSpec allows you to attach arbitrary key/value pairs to describe and it blocks to serve as metadata describing the test. The key value pairs come after the string argument, as in it "has metadata", focus: true" do. If your metadata is just a list of symbols and all the values are true, you can just pass the symbols, it "has_metadata", :focus do, and RSpec will assign true as the value automatically.

RSpec does not have a lot of structure to go with the metadata. It's available inside the spec using the hash example.metadata, so example.metadata[:fast]. You can use metadata to group tests and then run examples with that tag using the -t command line option: rspec -t fast, or rspec -t name:true for a key/value pair. If you want to exclude a particular tag, the you prefix the tag with ~, as in rspec -t ~name.

Metadata like this is sometimes used to specify a list of tests currently being worked on—the tag focus is often used. Sometimes you will see tags like fast or slow to break out groups of tests to be run together. Also, some tools, such as Capybara use test metadata to affect test execution.

What's Next

Over the rest of the book, we won't consistently give every example in both Minitest and RSpec—we will stick with Minitest's simplicity. However, when a tool is substantially different in an RSpec context than a Minitest one, we'll note that.

CHAPTER 11

Integration Testing with Capybara and Cucumber

An integration test specifies the behavior of multiple parts of your application working together. If a unit test is the main course of your testing meal, then an integration test is the cook preparing that meal, the waiter bringing to you, you eating it, and then paying and leaving a correct tip.

There are three related concepts here. An *integration* test is the generic name for any test that combines more than one unit. In Rails, integration tests are often *end to end* tests (or *black box* tests), meaning that they cover the entire system from the outside, making requests just as a user would, and validating the output that a user would see. An *acceptance* tests combines an end to end test with the idea that not only is the test specifying the behavior that the program expects, it is also specifying that the behavior is correct from the user or customer perspective. So, while every acceptance test is an integration test, not every integration test is an acceptance test.

We're going to focus for the purposes of this chapter on integration tests that are also end-to-end test, assuming that we'll need tools that will simulate HTTP requests and evaluate HTTP responses. For other kinds of integration tests, Minitest and RSpec can do just fine on their own.

What to Test in an Integration Test

We've talked before about the idea of the testing pyramid, where your tests have a relatively large number of unit tests that run quickly and test one small segment of the application, backed by significantly fewer integration tests that run more slowly cover the application as a whole.

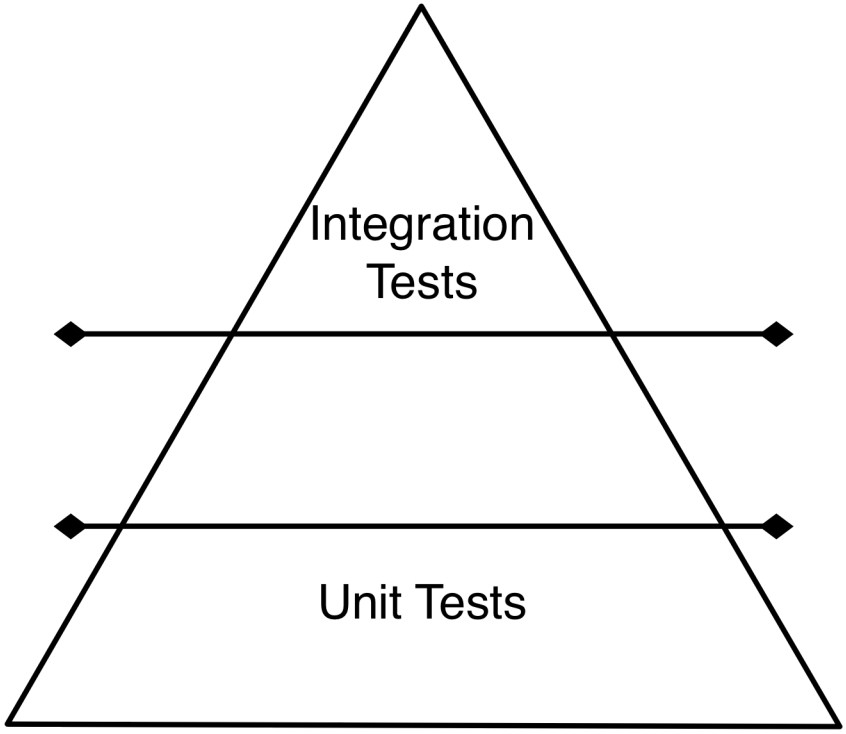


Figure 5—The Testing Pyramid

The pyramid metaphor makes it sound like the integration tests sit passively on top of a foundation of unit tests. It may be more useful to think of integration tests as the frame of a house. Without integration tests, you can't specify how your application works together. Without unit tests, there are all kinds of potential holes that bugs can sneak through.

Prescription 19 By far the biggest and easiest trap you can fall into when dealing with integration tests is the temptation to use them like unit tests.

Which is to say, don't use integration tests to specify logic that is largely internal details of your code base. One way to tell if your integration test is overly concerned with internal details is to think about what problem in the code would make the test fail. By definition, an integration test can fail in many places, but each test you write should have some particular circum-

stance that only that test protects against—a circumstance where that test would be the only test failure.

If that unique point of failure concerns the interaction between two objects (or sometimes, the interaction between two methods of the same object), then an integration test is called for. If the unique point of failure is the internal logic of a single object, then that condition is better covered in a unit test.

In a Rails context, things that are integration test fodder include:

- The interaction between a controller and the model or other objects that provide data
- The interaction between multiple controller actions that comprise a a common work flow.
- Certain security issues that involve the interaction between a user state and a particular controller action.

These things, generally speaking, are *not* integration tests. Use unit tests instead.

- Special cases of business logic, such as what happens if data is `nil` or has an unexpected value.
- Error cases, unless the error case genuinely results in a unique user experience.
- Internal implementation details of business logic.

There are two kinds of problems that happen when we use integration testing to cover things that are better done in unit tests. The first is speed. Integration tests are slower. Not because the tools themselves are slow, but because the tests are winding their way through the Rails stack to get to the method you are actually interested in. An entire suite of unit tests is like a house that's all frame—overbuilt.

The second is accuracy. Because the integration test is not making any assertions until after the units have all executed, it's often a little hard to piece together exactly what went wrong. Often,the way to deal with this is to have a failing integration test trigger a unit test.

There are many different tools that Rails developers can use for integration testing their code. We're going to talk about two of them. *Capybara* is a library that allows for easy integration with a web page, and also integrates with Minitest and RSpec. *Cucumber* is a higher-level tool that allows you to specify interactions in a natural language style.

Setting up Capybara

What is Capybara?

Capybara is a library that allows an automated test to simulate a user interaction with a browser. When simulating this interaction, Capybara works in conjunction with a driver, using the simple Capybara API to determine what elements to interact with, and using the driver to manage the actual interaction. By default, Capybara uses a native Ruby library that doesn't manage JavaScript interactions, but it can be configured to use a headless browser such as PhantomJS or Selenium to allow JavaScript interactions to be simulated.

Capybara and RSpec

Capybara is somewhat designed for use with RSpec, and if you want to use Capybara with RSpec, all you need to do is add Capybara to the testing group of your Gemfile:

```
gem "capybara"
```

You also need to add the following line to your rails_helper.rb, toward the top:

```
require 'capybara/rails'
```

Capybara and Mintest

The setup for Minitest is similar, but uses different gems that bridge the gap between Capybara and Minitest:

```
gem "minitest-rails-capybara"
```

The minitest-rails-capybara gem combines capybara, mintest-rails, minitest-capybara, and a little bit of its own glue. To get all that in our application, put the following line in the test_helper.rb file:

```
require "minitest/rails/capybara"
```

Capybara and its own language

At this point, we are forced to confront another library syntax decision. Capybara defines its validations mostly as things that look like RSpec matchers. Plus, it defines its own optional integration test syntax, which looks similar to RSpec, but uses terms like feature and scenario to drive home the integration testing point. Minitest, for its part, also has an RSpec-like add-on

called Minitest::Spec that I've been ignoring up until know, but which might look more natural with the Capybara matchers.

All of which adds up to a lot of different ways to express the same test. In the interest of minimizing the amount of time we spend talking about syntax, which is the least interesting thing we could be talking about, we'll stick to the same basic Minitest we've been using throughout most of the book. Keep in mind that the other forms exist, so that when your coworker shows you her Capybara tests, you know how to read them.

Outside in Testing

The process we are going to use to manage our Capybara tests is sometimes called "Outside In Testing", because we start using Capybara to write a test from the outside, and use that test to drive our unit tests. In much the same way that TDD uses a failing unit test to drive code, outside-in testing uses a failing acceptance test (or a failing line in an acceptance test) to drive the creation of unit tests.

The process follows this diagram—we're assuming the creation of a new user-facing feature in a Rails application.

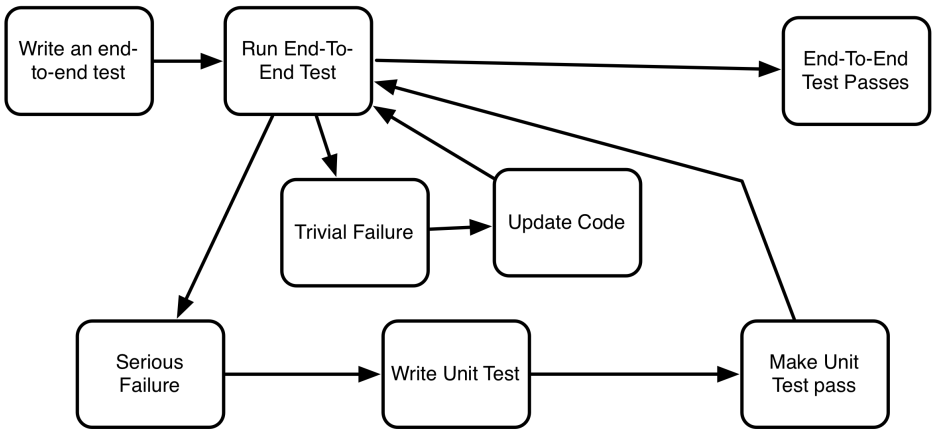


Figure 6—Outside In Testing

Let's walk through the steps.

1. Write the end-to-end test that shows a user interacting with the new feature. This test should specify data, have one or more user interactions, and validate HTML responses to determine that the interactions behaved

as expected. This should be the main, error-free, interaction of the feature. Now, before this process even starts, it's perfectly normal to noodle around with controllers and views trying to figure out exactly what the user interactions look like. You may even get a substantive part of the feature done. You need to be willing to rewrite that code once the tests come in to play.

2. Start running the tests. In a reasonably mature Rails application, the first few steps will often already pass. You'll often be adding a new feature to an existing page, so steps where data is prepared, users are logged in, existing pages are hit, might all work.
3. You will come to a test failure. The first failures you see in the integration test are often trivial: attempts to click on links that don't exist, form fields that aren't there yet. These can be quickly dispatched, generally without unit tests, on the theory that adding new stuff to a view isn't normally testable logic.
4. Eventually, though, you cross the chasm. You'll get to the point where you are validating responses, and there's a lot of logic missing from the code. Now you have a focused, well-defined task to accomplish in your code, and it's time to drop to unit tests.
5. Exactly what unit tests you need to write depend on the situation. You'll probably have model logic to test, and either a controller test or maybe a plain Ruby object managing a transaction. The important thing is that the unit tests may go beyond making the integration test work. They also have some responsibility to cover edge and error cases that wouldn't be covered by integration tests.
6. When you think the unit tests are done, go back to the integration test. There's a good chance it still fails, because you forgot some piece of integration. No problem, that's what the integration test is there for.
7. Finally the integration test passes. Yay! If there's another significant user case in the feature, write that test and start over.

Let's try a sample case and see how this plays out in practice.

Using Capybara

When last we left our application, we allowed for the creation of new projects. Let's follow up on that, and add a sequence where we can see a page for an existing project, and add a task to it. In order to give this a little bit of back end logic to play with, let's set up a situation where the tasks are ordered, and we then want to move one task above the other task.

Writing a Test

That test has a few different parts. Let's take a second and plan the given/when/then of it:

- Given: We'll need one existing project, and at least one existing task on that project so that we can test the ordering. Probably we'll want two tasks, that way we can verify that the UI is correct for the first, last, and middle parts of the list.
- When/Then: Filling out the form for the task and verifying that the new task shows up
- When/Then: Moving a test up and verifying that the order changes.

The fact that we have two distinct when/then pairs suggests this is probably really two tests, but for ease of explanation, we'll keep it as one. Let's see what it looks:

integration/01/gatherer/test/integration/add_task_test.rb

```
require "test_helper"

class AddTaskTest < Capybara::Rails::TestCase

  test "i can add and reorder a task" do
    visit project_path(projects(:bluebook))
    fill_in "Task", with: "Find UFOs"
    select "2", from: "Size"
    click_on "Add Task"
    assert_equal project_path(projects(:bluebook)), current_path
    within("#task_3") do
      assert_selector(".name", text: "Find UFOs")
      assert_selector(".size", text: "2")
      refute_selector("a", text: "Down")
      click_on("Up")
    end
    assert_equal project_path(projects(:bluebook)), current_path
    within("#task_2") do
      assert_selector(".name", text: "Find UFOs")
    end
  end
end
```

The test is kind of long and rambly. It also doesn't completely test the UI, in the sense that it's not validating that all the "Up" and "Down" links that are supposed to be there are actually there—we might do that in a helper test of some kind. It walks through the interaction, conveniently touching a significant part of the Capybara API.

Let's go through the Capybara API, looking at the Capybara method calls we use in this test and then explaining related methods. Then, we'll go through the rest of the process and make the test pass.

The Capybara API: Navigating

Capybara has one method to navigate to arbitrary routes in your system, and it's the first line of our test visit project_path(projects(:runway)). The visit method takes one argument, which is a string url (in our case, a Rails routing method that returns a string URL). The route generated by the visit method is always an HTTP GET. If you want to simulate a POST or any other kind of HTTP method, the recommended mechanism in Capybara is to navigate to a link or form button on the web page that triggers the desired interaction.

The Capybara API: Interacting

After our test hits the project_path url we start to use Capybara methods to interact with the elements on the page. Specifically, we use the fill_in method to place text in a text field, then the select method to choose an option from a select menu, then the click_on method to click on a button and submit a form. All told, Capybara has about ten methods for interacting with DOM elements.

Capybara is very flexible in how it allows you to specify the element you want to work with. You can specify any element by its DOM id. Form elements can also be specified by their name attribute. Form elements that have attached label tags can be specified by the text of the attached label. Elements like HTML anchor tags that have actual internal text can be specified via that actual text. HTML anchor tags whose body is an image can be located by the alt text attribute of the image.

In other words, if you have an HTML snippet like the following:

```
<form>
  <label for="user_email">Email</label>
  <input name="user[email]" id="user_email" />
</form>
```

You can use Capybara to access that form element with any of the following:

```
fill_in("user_email", with: "noel@noelrappin.com")
fill_in("user[email]", with: "noel@noelrappin.com")
fill_in("Email", with: "noel@noelrappin.com")
```

The first one uses the DOM id, the second uses the form name, and the third uses the label text.

By default, Capybara does subset matches, so you could also use “em” as a matcher, should you want to for some reason. If more than one element matches a locator, Capybara raises an error (Note: this is the behavior in the 2.2 version of Capybara, older versions behave differently). If you want an exact match rather than a substring match, then pass the option `exact: false` to any Capybara method that uses a locator.

Which lookup text should you use? It depends on your goals and your context. The label text will usually result in the most readable test. But it’s also the most fragile, since the user-facing text is most likely to change. In contrast, the DOM id is probably the most opaque in the test, but the least likely to change on a whim.

Here are the Capybara form interaction methods. First, the ones you’ll probably use a lot:

- `check(locator)`

Asserts that `locator` finds a check box, and checks it.

- `choose(locator)`

Same as `check`, but for radio buttons. Other radio buttons in the group are unchecked.

- `fill_in(locator, with: "TEXT")`

The `locator` is expected to find an text area or an input text field, and places the value of the `with` option inside it. Technically, the second argument is a generic options hash, but the only option that is unique to this method is `with`. I have no good reason why they chose to create the API with that argument as an option, though I wouldn’t be surprised if at some point in the future, it became a genuine Ruby 2.0 keyword argument.

- `select(value, from: locator)`

The method arguments are similar here, in that the second argument is technically an options hash, but the only usable option is `from`. It looks for a select menu that matches the locator passed to the `from` argument, and sets its value to the first argument. For what it’s worth, the fact that `fill_in` takes its locator argument first, and `select` takes the locator second drives me absolutely bananas.

- `click_on(locator)`

Finds an anchor link or a button via the locator, and simulates a click to it.

Then there are a few methods you will probably use less often:

- attach_file(locator, path)

Looks for a form file upload element that matches the locator, and simulates attaching the file at the given path to the form.

- click_button(locator)
- click_link(locator)

Like click_on, but only works for a button or only for a link.

- uncheck(locator)

Un-checks a checkbox specified by the locator.

- unselect(value, from: locator)

Un-selects the value from the select box specified by the locator. This one is most useful for multi-value select boxes. For a single value select box, all you need to do to unselect a value is select the new value.

The Capybara API: Querying

Capybara has a few methods designed to allow you to query the simulated browser page to check for the existence of various selector patterns in the page. This is one case where the syntax differs slightly between Minitest and RSpec.

We use current_url which has the current URL as a complete string. That's useful for testing whether your navigation links take you where you are going.

The most common query method in Capybara is assert_selector, which can also be written as page.assert_selector or page.has_selector?. All three versions are identical. By default Capybara looks for a CSS selector, using # for DOM id and . for DOM class in much the same way as the Rails assert_select. The assertion passes if the selector is found. If you want to specify that a given selector does not exist on the page, you can use assert_no_selector, refute_selector, or has_no_selector.

Our test first uses assert_selector to validate that .name and .size elements exist and that they match any options given. The Capybara methods all accept options, including the same text and count options as assert_select.

The RSpec equivalent, which is inferred from has_selector? is expect(page).to have_selector("SELECTOR") with the negation being expect(page).not_to have_selector("SELECTOR").

Capybara also has a series of methods of the form has_link?, which take locators rather than full CSS selectors, and find whether a DOM element of the given

type exists which matches the locator. You can find a complete list at http://rubydoc.info/github/jnicklas/capybara/master/Capybara/Node/Matchers.

Our test also makes use of the within method, which takes a selector argument and a block. The idea is that the selector passed to within will find a single element on the page. Inside the block, any Capybara call is scoped to only find or assert against the part of the page that is inside that particular element. So, the part of our test that looks like this:

```
within("#task_2") do
  assert_selector(".name", text: "Cast the designers")
end
```

Will only pass if there is a .name element inside the #task_2 element. This can also be written as assert_selector("#task_2 .name", text: "Cast the designers"), though I occasionally find that form to be a little flaky and hard to read.

Finally, the most useful Capybara method when things go wrong is save_and_open_page, which dumps the contents of the Capybara DOM into a temp file and opens the temp file in a browser (opening in a browser requires a gem called launchy). You won't have any CSS or images with relative file names, but it's still usually enough to tell that, say, you are stuck on a log in screen because you forgot to set up a logged in user.

Making the Capybara Test Pass

Let's go through the integration test process.

Our first error is right on the first line: projects(:bluebook) is a call to a fixture method, but we haven't defined any fixture named bluebook. That's easy enough to fix.

integration/01/gatherer/test/fixtures/projects.yml

```
one:
  name: MyString
  due_date: 2013-11-10

two:
  name: MyString
  due_date: 2013-11-10

bluebook:
  name: Project Blue Book
  due_date: <%= 6.months.from_now %>
```

While we're at it, we'll need two project tasks to make the sorting work. In order for this to work, we add belongs_to :project in app/models/task.rb.

integration/01/gatherer/test/fixtures/tasks.yml
```
one:
  project: bluebook
  title: Hunt the aliens
  size: 1
  completed_at:

two:
  project: bluebook
  title: Write a book
  size: 1
  completed_at:
```

I like using fixtures for integration tests because their transactional nature makes them much faster than creating the objects anew for each test. This makes it much more acceptable to have multiple objects in the integration test. (Note though, that if I use the same "Project Runway" name that is used in test/integration/add_project_test.rb, that test will fail because there will be more tasks than the test expects, so you do need to be careful with this global data.)

The next fail is that we don't even have a show method in the ProjectsController. The show method is easy enough, and probably doesn't need additional testing.

integration/01/gatherer/app/controllers/projects_controller.rb
```
def show
  @project = Project.find(params[:id])
end
```

We're also going to want a template. We know it's going to need a table for the tasks as well as a form to create a new task. Here's one. It's unstyled, but it's got the table and the form to create a new task.

integration/01/gatherer/app/views/projects/show.html.erb
```
<h2>Project: <%= @project.name %></h2>

<h3>Existing Tasks:</h3>

<table>
  <thead>
    <tr>Name</tr>
    <tr>Size</tr>
  </thead>
  <tbody>
    <% @project.tasks.each do |task| %>
      <tr>
        <td class="name"><%= task.title %></td>
        <td class="size"><%= task.size %></td>
        <td class="completed"><%= task.completed_at.to_s %></td>
      </tr>
    <% end %>
```

```
  </tbody>
</table>

<h3>New Task</h3>

<%= form_for Task.new(project_id: @project.id) do |f| %>
  <%= f.hidden_field :project_id %>
  <%= f.label :title, "Task" %>
  <%= f.text_field :title %>
  <%= f.label :size %>
  <%= f.select :size, [1, 2, 3, 4, 5] %>
  <%= f.submit "Add Task" %>
<% end %>
```

At this point we fail because the create task doesn't exist on TasksController. Which means we need to create new logic.

This feels like it'll be boilerplate enough not to need additional controller tests. Often, what I'll do at this point is a mini-spike of the controller method to see how much complexity is called for. In this case, the controller method is really simple:

integration/01/gatherer/app/controllers/tasks_controller.rb

```
def create
  @task = Task.new(
      params[:task].permit(:project_id, :title, :size))
  redirect_to @task.project
end
```

We don't have any obvious error cases here—Task has no validations. And even if we did have error cases, the remedy would be to go back to the project page anyway. So I'm willing to leave this for now. (In practice, I'd add an error message in the failure case).

Our next error is: Capybara::ElementNotFound: Unable to find css "#task_3".

The actual error here is a little subtle. In the original Capybara test, we're identifying each row by the order of the task, so the three rows will have DOM ids, task_1, task_2, and task_3. Not only don't we have those ids in the template we just showed, we don't even have a mechanism for ordering the tasks.

This is a good opportunity for some mini-design. What we want is a) each task has an order, b) the project displays the tasks in order, and c) new tasks come in at the end of the list.

We can handle the first part, giving tasks an order, with a Rails migration

```
$ rails generate migration add_order_to_tasks
```

I'm calling the new attribute project_order to avoid confusion with the SQL order statement.

integration/02/gatherer/db/migrate/20140518182734_add_order_to_tasks.rb

```
class AddOrderToTasks < ActiveRecord::Migration
  def change
    add_column :tasks, :project_order, :integer
  end
end
```

```
$ rake db:migrate
```

We can make products automatically return tasks in order by using ActiveRecord and changing the declaration of the relationship to has_many :tasks, -> { order "project_order ASC" }— that doesn't need a test, as such, because it's really part of the framework.

For the tests to work, we also need to add the order to the fixtures in test/fixtures/tasks.yml—give the first fixture in the file an project_order: 1 line and the second fixture gets an project_order: 2.

The part where new tasks are given an order by the project adds some logic. There are several ways to do this. We could make it a callback on Task, which would be auto invoked when the task is saved, or we could make it a method on Project to ask the project for the next order, or we could create a Task factory object similar to the CreatesProject object we built earlier.

To keep things simple, I'm going to make a method on Project that we'll call when creating the Task. We're now in unit test mode, so I write some unit tests:

integration/02/gatherer/test/models/project_test.rb

```
test "give me the order of the first task in an empty project" do
  project = Project.new(name: "Project")
  assert_equal(1, project.next_task_order)
end

test "give me the order of the next task in a project" do
  project = Project.create(name: "Project")
  project.tasks.create(project_order: 3)
  assert_equal(4, project.next_task_order)
end
```

This code passes with:

integration/02/gatherer/app/models/project.rb

```
def next_task_order
  return 1 if tasks.empty?
  tasks.last.project_order + 1
end
```

Now we need to integrate it. First off, we need to call the new method in the controller. I've slightly reorganized the controller logic so that we have a project to query:

integration/02/gatherer/app/controllers/tasks_controller.rb

```
def create
  @project = Project.find(params[:task][:project_id])
  @project.tasks.create(title: params[:task][:title],
      size: params[:task][:size],
      project_order: @project.next_task_order)
  redirect_to @project
end
```

I still don't think the controller logic warrants another test.

Now, in the app/views/projects/show.html.erb template file, we replace the table row line with the following:

```
<tr id="task_<%= task.project_order %>">
```

Which at long last gives us our #task_3 selector.

And brings us to our next point of failure, Unable to find link or button with text "Up", meaning that the update logic is not yet in place.

What's the logic we want? And how can we write a test for it?

We want an "Up" link for all tasks but the first one, and a "Down" link for all tasks but the last one. Which means we need to be able to tell if a task is first or last. That's testable:

integration/02/gatherer/test/models/task_test.rb

```
test "it finds that a test is first or last" do
  project = Project.create!(name: "Project")
  first = project.tasks.create!(project_order: 1)
  second = project.tasks.create!(project_order: 2)
  third = project.tasks.create!(project_order: 3)
  assert first.first_in_project?
  refute first.last_in_project?
  refute second.first_in_project?
  refute second.last_in_project?
  refute third.first_in_project?
  assert third.last_in_project?
end
```

That's more assertions than I would normally place in a single test, but they are very closely related, so I think it reads better as a single group then any other way.

And the test passes with:

integration/02/gatherer/app/models/task.rb
```
def first_in_project?
  return false unless project
  project.tasks.first == self
end

def last_in_project?
  return false unless project
  project.tasks.last == self
end
```

Now we need to place that logic in the view template, which we can do by adding the following inside the loop in app/views/tasks/show.html.erb. There's a decision point here, which is whether to send that link to the regular update method for TasksController, which already exists, and which would need to be changed. Alternately, we can create new up and down controller actions. It's not strictly RESTful, but then this isn't strictly a RESTful action.

Let's go with the non-restful controller action. We need to define them in the routes file:

integration/02/gatherer/config/routes.rb
```
Gatherer::Application.routes.draw do
  resources :tasks do
    member do
      patch :up
      patch :down
    end
  end

  resources :projects
end
```

Then attach the routes to the template...

```
<td>
  <% unless task.first_in_project? %>
    <%= link_to "Up", up_task_path(task.id), method: :patch %>
  <% end %>
  <% unless task.last_in_project? %>
    <%= link_to "Down", down_task_path(task.id), method: :patch %>
  <% end %>
</td>
```

What we need next is the ability to swap a task's order with one of its neighbors, which implies the ability to find its neighbors. This would seem to be a model concern, which means we are back writing unit tests.

What do these tests need? The same set of three tasks, presumably, and a move_up and move_down method to test.

Although I did write and pass these two tests one at a time, let's look at them together:

integration/02/gatherer/test/models/task_test.rb

```
def project_with_three_tasks
  @project = Project.create!(name: "Project")
  @first = @project.tasks.create!(project_order: 1)
  @second = @project.tasks.create!(project_order: 2)
  @third = @project.tasks.create!(project_order: 3)
end

test "it can move up" do
  project_with_three_tasks
  assert_equal @first, @second.previous_task
  @second.move_up
  assert_equal 2, @first.reload.project_order
  assert_equal 1, @second.reload.project_order
end

test "it can move down" do
  project_with_three_tasks
  assert_equal @third, @second.next_task
  @second.move_down
  assert_equal 2, @third.reload.project_order
  assert_equal 3, @second.reload.project_order
end
```

And one set of task methods that passes the tests and have gone through a refactoring step:

integration/02/gatherer/app/models/task.rb

```
def my_place_in_project
  project.tasks.index(self)
end

def previous_task
  project.tasks[my_place_in_project - 1]
end

def next_task
  project.tasks[my_place_in_project + 1]
end

def swap_order_with(other)
  other.project_order, self.project_order =
      self.project_order, other.project_order
  self.save
  other.save
end

def move_up
```

```
  swap_order_with(previous_task)
end

def move_down
  swap_order_with(next_task)
end
```

And now we are really, really close. All we need to do is wire up the controller:

integration/02/gatherer/app/controllers/tasks_controller.rb

```
def up
  @task = Task.find(params[:id])
  @task.move_up
  redirect_to @task.project
end

def down
  @task = Task.find(params[:id])
  @task.move_down
  redirect_to @task.project
end
```

And…now the integration test passes, and all the tests are green and we celebrate.

Retrospective

Let's take a step back and discuss what happened here. We're supposed to be talking about integration tests, and yet we never touched the integration test after we initially wrote it. What good is that? (Full disclosure: in initial writing I did have to go back and clean up the integration test because I originally had syntax errors and the like. Sometimes I need to fix the assertions after I see what the view looks like.)

The existence of the integration test gives us a couple of benefits as we write these tests.

- There's a structure, we know what needs to be done next. This is valuable, although given the nature of a lot of server-side development, there does tend to be one large gap where the integration test provides the expected outcome, but doesn't give much guidance on how to get there.
- The integration test proves that all of our unit tests work together as a cohesive whole. (Though you really need to test in a browser when you are done…) One danger sometimes cited for unit tests is that it's easy to see the trees and not the forest. Having an end-to-end test forces you to describe the forest.

- Although the integration test is slow compared to unit tests, it's often lightning fast compared to manually setting up the browser integration. If you've ever tested a form with dozens of required fields, you can see the benefit here.

Now, looking back at the original test, I said it was kind of long and unwieldy. I could go back and split out related lines into their own methods.

Or I could try Cucumber.

Cucumber

Cucumber is a tool for writing acceptance tests in plain language (I almost said in plain English, but that's too limiting—Cucumber speaks a lot of different languages, including LOLcat). It's designed to allow you to write simple and clear integration and acceptance tests. It can be used to have a non-developer client or manager co-write or sign off on the acceptance tests, though my personal experience with that has been hit-and-miss. Because it adds a level of indirection to acceptance testing, some people feel that Cucumber is more trouble than it is worth.

I like to look at Cucumber as a tool that does two things. First, it allows me to structure integration tests somewhat cleanly. Second, it allows me the option to describe the intended behavior of the system without using the code part of my brain at all, which I find to be a helpful thing to do.

Setting Up Cucumber

To install Cucumber, we need two gems in the Gemfile:

```
group :development, :test do
  gem 'cucumber-rails', require: false
  gem 'database_cleaner'
end
```

The require: false on cucumber-rails prevents a warning when Rails is loaded. Strictly speaking, database_cleaner isn't required, but it's valuable and gives fixture-like transaction behavior to your non-fixture using tests. The actual cucumber gem will be installed as a dependency of cucumber-rails. As I write this, we're talking about version 1.3.15 of Cucumber and version 1.4.1 of cucumber_rails.

To install Cucumber, there's one command line generator:

```
rails generate cucumber:install
```

This creates a config/cucumber.yml file for runtime options, the actual cucumber command line script, a rake task, a features directory with subdirectories for step_definitions and support and a features/support/env.rb file, which is analogous to test_helper.rb. Also, it modifies the database.yml file to add a cucumber environment that copies the test environment. Cucumber has some additional configuration options, you can type rails generate cucumber:install —help to see them.

Writing Cucumber Features

In Cucumber, you write tests as a series of steps using a very minimal language called Gherkin. An individual Cucumber test is called a Scenario, and a group of them are called a Feature.

Lets take the Capybara integration test from the last section and convert it to Cucumber. Cucumber feature files go in the features directory, and typically end in .feature. So here is features_add_task.feature:

integration/03/gatherer/features/add_task.feature

```
Feature: Adding a task

  Background:
    Given a project

  Scenario: I can add and change priority of a new task
    When I visit the project page
    And I complete the new task form
    Then I am back on the project page
    And I see the new task is last in the list
    When I click to move the new task up
    Then I am back on the project page
    And the new task is in the middle of the list
```

This file has three parts to it. First off, the Feature declaration at the top. Our Cucumber file needs to have one, but the description there is strictly for the humans, we can put anything there that we want. Gherkin is whitespace sensitive, so anything that goes beyond that top line needs to be indented.

The next section is the Background, which is optional. It might not be clear from the above snippet, but the Background line is indented subordinate to Feature. In Cucumber, Background is the equivalent to Minitests's setup and RSpec's before, indicating code that is run to initialize each test. In our case, since we only have one Scenario, it's not really necessary to have a background, but if we did have multiple "Add a task" scenarios, they'd likely all share that one common background.

After the Background comes the Scenario, which is the actual test. Both Background and Scenarios are made up of steps. In Cucumber a step usually consists of one of the words Given, When, or Then followed by a sentence. This corresponds to the same basic structure that we've been using informally to discuss tests all through the book: Given, which indicates a precondition to the action. When indicates a user action that changes the state of the application. Then specifying the result of the state change. You can also start a line with And or But (or *, if you must) if its desirable for readability.

The distinction between Given, When, and Then is for the humans. Cucumber does not require the steps to be in a particular order. And when it comes time to match each step to its definition the Given/When/Then header is not significant in the match.

This scenario is executable right now, using the cucumber command, or we can specify the file with cucumber feature/add_task.feature if we want. The output comes in two parts. The first is a listing of the execution step by step, which starts:

```
Background:       # features/add_task.feature:3
    Given a project # features/add_task.feature:4
      Undefined step: "a project" (Cucumber::Undefined)
      features/add_task.feature:4:in `Given a project'

  Scenario: I can add and change priority of a new task
                  # features/add_task.feature:6
    When I visit the project page
                  # features/add_task.feature:7
      Undefined step: "I visit the project page" (Cucumber::Undefined)
      features/add_task.feature:7:in `When I visit the project page'
```

You can't see this on the page, but all that text other than the Background and Scenario lines will be yellow.

When a scenario is run, Cucumber attempts to run each step by matching it to a step definition and executing the step definition. In this output, Cucumber helpfully tells us the line number of each step it tries to execute, and then tells us that the step is undefined. Which makes sense, because we have not defined any steps yet.

After it goes through all the steps, and reports that we have 1 scenario with 8 steps of which 1 scenario and 8 steps are undefined, Cucumber adds some extra output to the terminal, which starts like this...

```
You can implement step definitions for undefined steps with these snippets:

Given(/^a project$/) do
  pending # express the regexp above with the code you wish you had
```

```
end

When(/^I visit the project page$/) do
  pending # express the regexp above with the code you wish you had
end
```

This output continues for all the undefined steps. What Cucumber is doing here is actually really useful, giving us boilerplate templates for each of the undefined steps that we can paste directly into our editor and then fill out.

So, lets do that, I'm taking the whole ball of wax, all eight of those pending blocks and pasting them into a new file features/step_definitions/add_task_steps.rb. When I do that, and rerun Cucumber, well... a little bit changes:

```
Background:       # features/add_task.feature:3
    Given a project # features/step_definitions/add_task_steps.rb:1
      TODO (Cucumber::Pending)
      ./features/step_definitions/add_task_steps.rb:2:in `/^a project$/'
      features/add_task.feature:4:in `Given a project'

  Scenario: I can add and change priority of a new task
                # features/add_task.feature:6
    When I visit the project page
                # features/step_definitions/add_task_steps.rb:5
    And I complete the new task form
                # features/step_definitions/add_task_steps.rb:9
    Then I am back on the project page
                # features/step_definitions/add_task_steps.rb:13
    And I see the new task is last in the list
                # features/step_definitions/add_task_steps.rb:17
    When I click to move the new task up
                # features/step_definitions/add_task_steps.rb:21
    Then I am back on the project page
                # features/step_definitions/add_task_steps.rb:13
    And the new task is in the middle of the list
                # features/step_definitions/add_task_steps.rb:25
```

The top lines are yellow, the lines under "Scenario" are light blue. What's happening now is that Cucumber is stopping the test at the first pending step and marking each further step as "skipped".

Now its time to tell Cucumber what each of those steps should do.

Writing Cucumber Steps

Sadly, it's unrealistic for Cucumber to know what to do just from a step like Given a project. So we must define all the steps so that they can be executed by Cucumber.

When Cucumber gets a step like Given a project, it searches through all the files in the step definition folder looking for one definition that matches. What does matching mean? Let's look at the boilerplate for that step again:

```
Given(/^a project$/) do
  pending
end
```

The first line of the definition is one of those Given/When/Then words (it doesn't matter which one) followed by a regular expression. Cucumber matches a step to a definition when the end of the step, like a project, matches the regular expression, like /^a project$/. We'll see in a little bit why Cucumber uses regular expressions instead of just strings. So, when Cucumber sees the step Given a project, it will run the code inside the block for the matching step definition. If Cucumber finds more than one matching step definition it will raise an error.

Inside a step definition you can write any arbitrary Ruby code. Instance variables declared in one step definition will be available in later ones in the same test—the variables are cleared between tests. Be a little careful with instance variables, it's not always easy to tell what variables might exist from previous steps, or what state they might be in. Cucumber understands Capybara methods, and it understands RSpec matchers assuming RSpec is installed. Arbitrary methods defined in any step definition file will be available to any step definition.

By default, Cucumber doesn't understand Minitest. If we want to use Minitest assertions, we can place the following file inside features/support.

integration/03/gatherer/features/support/minitest.rb

```
require 'minitest'
module MiniTestAssertions
  def self.extended(base)
    base.extend(MiniTest::Assertions)
    base.assertions = 0
  end

  attr_accessor :assertions
end
World(MiniTestAssertions)
```

The Cucumber specific line here is the last one, where World is Cucumber's global configuration object, and calling it with a module name extends world with that module, triggering the extended method of the code which injects Minitest assertions into Cucumber. Or you could ignore this entire paragraph and just use RSpec matchers.

Our case here is unusual, in that we already have this feature managed as a non-Cucumber integration test, so filling the steps is mostly a question of splitting that test into pieces. My copy looks like this:

integration/03/gatherer/features/step_definitions/add_task_steps.rb

```
Given(/^a project$/) do
  @project = Project.create(name: "Bluebook")
  @project.tasks.create(title: "Hunt the Aliens", size: 1, project_order: 1)
  @project.tasks.create(title: "Write a book", size: 1, project_order: 2)
end

When(/^I visit the project page$/) do
  visit project_path(@project)
end

When(/^I complete the new task form$/) do
  fill_in "Task", with: "Find UFOs"
  select "2", from: "Size"
  click_on "Add Task"
end

Then(/^I am back on the project page$/) do
  assert_equal project_path(@project), current_path
end

Then(/^I see the new task is last in the list$/) do
  within("#task_3") do
    assert_selector(".name", text: "Find UFOs")
    assert_selector(".size", text: "2")
    assert_no_selector("a", text: "Down")
  end
end

When(/^I click to move the new task up$/) do
  within("#task_3") do
    click_on("Up")
  end
end

Then(/^the new task is in the middle of the list$/) do
  within("#task_2") do
    assert_selector(".name", text: "Find UFOs")
  end
end
```

There are only a few differences between the Cucumber steps and the original integration test:

In the original integration test, we defined the project using fixtures, in the Cucumber test, I explicitly create the @project and its tasks in the before

method. Cucumber does not use fixtures by default. Although it's possible to get Cucumber to understand fixtures, it's another page of wonky Rails internals, and frankly nobody needs that right now. We do lose the speed benefit of fixtures and database transactions. That change cascades—references to the fixture project are replaced with references to the instance method.

I've split up checking that the new task is last and clicking to move it up, so each of those needs it's own within block. Also, for some reason, Cucumber didn't like refute_selector, so I replaced it with the identical assert_no_selector. Other than that, the Ruby code is the same, with just a different background structure.

Since we've already written the code once, there's no reason go go through the whole process again. The Cucumber process is very similar to the Capybara only process, I write a scenario, try to make the steps pass one by one, dropping down to unit tests when I need logic. The biggest difference is that Cucumber makes it easier me to write the outline of the scenario without writing the details of the later steps.

More Advanced Cucumber

This section should be titled "Things Cucumber lets you do that are bad ideas". Cucumber allows for a lot of flexibility in the way that steps match with step definitions. By and large, the Cucumber-using community has come to the conclusion that most of these things should be used sparingly, if at all.

Earlier, we alluded to the idea that step definitions were regular expressions and not strings. This allows you to have the same step definition apply to multiple strings. More to the point, you can use regular expression groups to capture those matches. The parts of the string that are in groups are then passed as block variables to the block part of the step definition. This allows you to use a Cucumber step as a method call with parameters.

In our existing initial step, we hardcode the name of the project inside the step definition. If, on the other hand, we wanted to be able to specify the name of the project in the Cucumber feature, we could write the step definition as follows:

```
Given /^a project named "(.*)"$/ do |project_name|
  @project = Project.create!(name: project_name)
end
```

That definition would match steps like:

```
Given a project named "Rails 4 Test Prescriptions"
```

```
Given a project named "Evil Master Plan"
```

In each case the step definition would create a project with the given name.

You can go even further in terms of putting data in the Cucumber feature file. Cucumber allows you to append a table of data to a step. It's a very clever feature, and like a lot of very clever features, it should be used judiciously.

You use something like Markdown table syntax to create the table. In the feature file, it might look something like this:

```
Given the following users:
    | login| email            | password| password_confirmation|
    | alpha| alpha@example.com| alpha1  | alpha1               |
    | beta | beta@example.com | beta12  | beta12               |
```

The step with the table needs to end with a colon. The table uses vertical pipes to delimit entries. They don't actually have to line up, but you'll normally want them too.

Inside the step definition that matches the table, the table comes into the step definition as an argument to the block—if there are other regular expression matches, then the table is the last argument. The argument is a special Cucumber data type, and there are a few different ways you can deal with it, most commonly you want to deal with it as an array of hashes, where the keys are the first row of the table, and every subsequent row provides a set of values, like so:

```
Given /^the following users$/ do |user_data|
  User.create!(user_data.hashes)
end
```

You can do something similar with a large block of string:

```
Given I have typed the following
  """
  some big amount of text
  """
```

That's an indented line, three quotes, some text, and then three more quotes at the end. The text inside the triple quotes will be passed as the last argument to the step definition block.

If you really want to have fun, you can combine scenarios and tables to basically create a loop called a *Scenario Outline*, like so:

```
Scenario Outline: Users get created
  Given I go to the login page
  When I type <login> in the login field
  And I type <password> in the password field
```

```
  Then I am logged in
Examples:
  | login| email            | password| password_confirmation|
  | alpha| alpha@example.com| alpha1  | alpha1               |
  | beta | beta@example.com | beta12  | beta12               |
```

What happens is that the steps inside the outline are regular Cucumber steps. When the outline runs, it runs once for each data row in the Examples table, with the <login> syntax indicating that a value from that table should be inserted in the row.

All these features give you a tremendous amount of power. I advise you to use it sparingly. It's tempting to use these tools to reduce duplication or make your steps more general. But the flip side is that you are often declaring implementation data explicitly in the Cucumber file rather than implicitly in the step definition.

There are at least three problems with explicit Cucumber steps:

- All the flexibility can make for complicated step definitions. Since Cucumber depends on the step definitions doing exactly what they say they are going to do about 100% of the time, complex step definitions are bad because they are more likely to contain errors. Debugging step definitions will make you question your life choices. Keeping step definitions simple makes Cucumber easier to manage.
- Putting a lot of code-like things, including data, attribute names, and CSS selectors in Cucumber feature files makes them hard to read and parse. Since the point of Cucumber is to be natural language-like, writing unreadable steps defeats the purpose.
- Similarly, but more subtly, putting data in the feature file robs the feature file of its ability to declare intent. What is the point of this line: Given a user that has been on the site for 2 months?. It's hard to tell. On the other hand,Given a user that has been on the site long enough to be trusted is much more clear and explains why the step exists. This is a case where specifics imply greater meaning to your somewhat-arbitrary data choices than they actually deserve.

Is Cucumber Worth It?

Depends on what "it" is. Cucumber is a very helpful acceptance test framework, provided your expectations of it are reasonable. It's a lousy unit test framework, and if you try to use it for unit testing, you will hate it and possibly stop eating salads to avoid cucumbers, which is bad for your health.

I use Cucumber for the relatively minimal goals of a) being able to write my integration tests at the level of user behavior and b) easily being able to separate my slower integration tests from my faster unit tests. For those things, it works great.

You will sometimes hear that Cucumber allows for non-developer members of your team to participate in the acceptance testing process, because Cucumber is natural-language like. My experiences in that regard are mixed. I've had some success with writing Cucumber scenarios on my own and giving them to managers or clients for approval. Going the other direction the limiting factor in my experience is not the syntax, but experience in how to specify requirements. Which is tricky for everybody.

Some tips for better cuking:

- Write the scenario in natural language that defines the behavior of the system from the user perspective, smooth out details in the step definition.
- Avoid anything in the feature file that looks like code or data. Including CSS selectors. And database attributes.
- Keep step definitions simple
- Don't worry about duplicating bits of step logic. Prefer multiple simple steps over one big one with complex logic.
- Specifying what isn't on the page is often as important as specifying what is.
- Worry about implementation details in the unit tests. The suggestions about what is an integration test and what is a unit test also apply here.
- Validate against user visible pages rather than database internals. (Sometimes this means going to an admin page or similar to validate that a change has taken place.)

Looking Ahead

We'll be talking more about these tools in future chapters.

In *the (as yet) unwritten chp.javascript*I don't know how to generate a cross reference to chp.javascript, we'll show how both Capybara and Cucumber can be attached to drivers that run the tests against a browser engine that executes JavaScript, allowing for client-side actions to be integration tested. In Chapter 13, *Testing External Services*, on page 231, we'll talk about integration tests that might need to touch a third-party service. And in *the (as yet) unwritten chp.environment*I don't know how to generate a cross reference to chp.environment, we'll talk about how to optimize the command line execution of both tools so as to speed up your feedback loop.

CHAPTER 12

Testing For Security

Web security is a very scary topic. All of our applications depend on cryptography and programming that is beyond our immediate control. Despite that, there are parts of web security that are in our control—all the logins and access checks and injection errors that happen on our site as a result of programming choices we make.

When it comes to security and testing, there is good news and bad news. The good news is that all kinds of access and injection-type bugs are amenable to automated developer testing. Sometimes unit testing will do the trick, other times end-to-end testing, but the effects of a security problem are often easily reproducible in a test environment. The bad news is that you need to actively determine where access and injection bugs might lurk in your code. We are going focus on user logins, roles, and using tests to make sure that basic user authentication holds in your application.

Prescription 20 Security issues are, at base, just bugs. Most of the practices you do to keep your code bug-free will also help prevent and diagnose security issues.

User Authentication and Authorization

We've gotten quite far in our example without adding a user model to it, which we are going to rectify right now.

We want to get users and passwords in the system without spending too much time in the setup weeds, so we can focus our attention on the security issues that having users causes. So, we'll use the Devise gem for basic user authentication, and focus on how to use Devise as part of or security and

testing goals. (Part of me wants to derive user login from first principles, and someday when I publish a book from The Purist Press I'll do that.)

Devise is a big, multi-faceted gem, and we'll only be scratching the surface of what it can do. It handles all kinds of login needs, including confirmation emails, changing passwords, "remember me" cookies, and much more. Full documentation for Devise is available at http://devise.plataformatec.com.br First up, we need to put it in the Gemfile.

```
gem 'devise'
```

As of this writing, the current version of Devise is 3.2.4.

After we install the gem with bundle install, we have two generation steps that we need to take. The first is the general installation of the Devise setup:

```
$ rails generate devise:install
      create  config/initializers/devise.rb
      create  config/locales/devise.en.yml
```

This gives us a devise.rb initializer, containing a lot of setup options that we are not going to worry about at the moment, and a locale file containing all the static text Devise uses. We're not going to worry about that file, either.

At the end of the generation process, Devise gives us a useful list of a few tasks that we need to do by hand, to allow Devise to integrate with the application. The relevant tasks are:

- In config/environments/development.rb, set some default mailer options, by adding the suggested default line, config.action_mailer.default_url_options = { host: 'localhost:3000' }. In a real application, we'd need that in our other environments as well, with the host pointing to, well, the host URL.
- Our config.routes.rb needs a root route, for example, by adding root to: projects#index, which is the closest thing we have to a root route.
- Devise uses the Rails flash to distribute messages of success and failure, add the following to app/views/locales/application.html.erb, any place in the file that seems relevant:

```
<p class="notice"><%= notice %></p>
<p class="alert"><%= alert %></p>
```

That's it. If we wanted to copy all of the Devise view code for the dialogs and stuff, we could also run rails generate devise:views from the command line. We'll skip that for now.

Now we need to actually generate a User model compatible with Devise:

```
% rails generate devise User
```

This creates a User model, a migration, a spec or test file, and a factory_girl factory if factory_girl is installed. It also adds a line to the routes.rb file that handles all the login and logout routes.

Our User model has nothing but some Devise commands:

security/01/gatherer/app/models/user.rb
```
class User < ActiveRecord::Base
  # Include default devise modules. Others available are:
  # :confirmable, :lockable, :timeoutable and :omniauthable
  devise :database_authenticatable, :registerable,
         :recoverable, :rememberable, :trackable, :validatable
end
```

Each of those symbols passed to the devise method enables another of Devise's features, and assumes a certain set of database columns, the list of which you can see in the generated migration file.

To get these new columns in the database, we need to run the migrations:

```
% rake db:migrate
```

Finally, Devise has some test helpers that we need to include in our controller tests to enable login behavior in tests. At the bottom of our test_helper.rb file, add the following:

security/01/gatherer/test/test_helper.rb
```
class ActionController::TestCase
  include Devise::TestHelpers
end
```

RSpec fans should go into the rails_helper.rb file and add the following:

security/01/gatherer/spec/rails_helper.rb
```
config.include Devise::TestHelpers, type: :controller
```

This line adds the same test helpers. (RSpec fans should also note that the code directory for this book will often RSpec versions of tests in the text, even though we won't specifically call out all of them in the book).

Adding users and roles

Now that we have Devise installed, let's see how we can use testing to expose security issues.

The most basic security issue is user login. Since our application involves projects that would presumably be limited to a specific, private, set of users, it makes sense that you would need to be logged in to access the application.

This is testable logic—a logged in user can access a page, an ordinary browser who happens along the page can not.

So here's an integration test for the project index page:

security/01/gatherer/test/integration/user_and_role_test.rb

```
Line 1 require "test_helper"
-
- class UserAndRoleTest < Capybara::Rails::TestCase
-
5   def log_in_as(user)
-     visit new_user_session_path
-     fill_in("user_email", :with => user.email)
-     fill_in("user_password", :with => user.password)
-     click_button("Sign in")
10  end
-
-   setup do
-     @user = User.create(email: "test@example.com", password: "password")
-   end
15
-   test "a logged in user can view the project index page" do
-     log_in_as(@user)
-     visit(projects_path)
-     assert_equal projects_path, current_path
20  end
-
- end
```

This test uses Capybara, and we've seen most the component parts before. However, this test does have the first of several answers to the question "how do I simulate a user login in an automated test". In the helper method log_in_as on line 5, we simulate a user login by actually simulating a user login. The method uses Capybara and the standard Devise login route and login form to simulate heading to the login page, which Devise calls the new_user_session_path, filling in the user's email and password, and then clicking a button, for which the default Devise caption is "Sign in". This method will be boilerplate across projects, depending on the name of the model that controls login or how much you customize the login page itself.

Directly simulating a login has the benefit of actually exercising the real login page, and making sure that Devise is correctly integrated with your application. However, it's an extra page load, so it's kind of slow. We'll see a shortcut in our next example. In practice, you should use the real login page at least once in your test suite.

In our actual test, we create a user and call the log_in_as method with that user. We then simply visit the project index page, projects_path, and verify that the program actually got there.

And the test passes, as is.

Which should be a little suspicious.

In our case it means we've only done half the test. We've done the "this is okay" part, but we haven't done the "blocking miscreants" part.

> **Prescription 21** Always do security testing in pairs: the blocked logic and the okay logic.

The test for blocking unauthorized access is just to simulate unauthorized access. Or, in other words, just hit the page without logging in.

security/01a/gatherer/test/integration/user_and_role_test.rb

```
test "without a login, the user can't see the project page" do
  visit(projects_path)
  assert_equal new_user_session_path, current_path
end
```

We're asserting that a user who goes to the project page and is not logged in is redirected to the login page, which is standard Device behavior.

The test fails. To make it pass, we just add Devise's authenticate behavior to the parent controller, which will make our entire application login-protected:

security/01/gatherer/app/controllers/application_controller.rb

```
class ApplicationController < ActionController::Base
  # Prevent CSRF attacks by raising an exception.
  # For APIs, you may want to use :null_session instead.
  protect_from_forgery with: :exception
  before_action :authenticate_user!
end
```

The line we've added is before_action :authenticate_user!. The before_action part means it will run before any action in the application, and the authenticate_user! part is the Devise check to see if the current session has a logged in user.

The good news: both of the new security tests now pass.

The bad news:

```
$ rake test:all
Run options: --seed 16144

# Running:
```

```
FF.F.FFF...E.......................EF..........
```

A lot of other tests fail.

When a change in your code breaks multiple tests, that's often a good time to revisit your testing strategy. It's very common for new data or security constraints to break tests that are unaware of new requirements. But multiple tests breaking can be a sign that your tests are too entangled with the internals of classes that are not directly under test.

Prescription 22 When a single change in your code breaks multiple tests, consider the idea that your testing strategy is flawed.

The specific test failures are all due to code attempting to hit the site without having a log in. It might mean that some of the testing that we are doing at the controller level might better be done at a unit test level against an action object. In our case, it also means we have some spurious tests floating around from the mocks chapter.

What we need to do is simulate a logged in user for each failing test. We have four failing test areas: the project and task controller test, and the two other integration tests we have for adding projects and adding tests.

We need a fake user to log in. Since we really don't care about the details of this user, and we'd need to recreate it for each test, this seems like a fine use case for fixtures. Here's a user fixture for our test user:

security/01/gatherer/test/fixtures/users.yml

```
user:
  email: "test@example.com"
  encrypted_password: <%= User.new.send(:password_digest, 'password') %>
```

The gobbledygook being sent to encrypted_password is there to ensure that our password is being sent to the database encrypted in exactly the way Devise expects so we can match the user on log in. If we were creating the user from Rails, we could just use User.create(email: "test@email.com", password: "password"), but since password is not in the database, but is instead an accessor mediated by Devise, we don't have access to it from the fixture and must go directly to the encrypted_password column in the database.

The controller tests can then use Devise's methods directly in their setup. You need this snippet of code in both projects_controller_test.rb and tasks_controller_tests.rb.

security/01/gatherer/test/controllers/projects_controller_test.rb

```
setup do
```

```
  sign_in users(:user)
end
```

The sign_in method is provided by Devise's test helpers, and allows us to sign in a particular user. This user is to the test session and appears as the currently logged in user for the duration of the test, or until you use the Devise helper method sign_out. Alternately, you could stub the controller method current_user, which is used by most of the rest of the code to determine if there is an active logged in user.

For the integration tests, we don't have sign_in available, since it works at the internal Rails level, but there is a different shortcut. We can use a helper provided by Warden, which is the tool that Devise uses for the authentication functionality. Warden works at the Rack layer. Without spending a page or two explaining Rack, suffice to say that we can use Warden to fake a logged in user from outside the Rails stack, suitable for using within our integration tests.

You'll need the following setup line added in both the add_project_test.rb and add_task_test.rb.

security/01/gatherer/test/integration/add_project_test.rb

```
include Warden::Test::Helpers

setup do
  login_as users(:user)
end
```

The Warden helpers need to be included into the test class, and then login_as is available for our use. The technical difference between login_as and sign_in is not super-relevant for us (the difference in the two names is, however, irritating...). The practical difference is that login_as comes from outside the Rails stack and can therefore be used in integration tests, whereas sign_in only manipulates Rails internals, and can only be used in a test, like a controller test, that is intended to have access to Rails internals.

With that addition to the controller and integration tests, our suite passes again. (RSpec equivalents are available in the security/01/gatherer/spec directory of the code download). There is nothing to be gained right now from trying to move any of these tests to avoid the additional setup, however, I will note that our action tests for creating a project in test/actions/creates_project_test.rb did not need to be changed.

Restricting Access

Having required a login for our application we've solved part of our potential security problem. The next problem involves restricting access to projects that the user is associated with.

We'll start with an integration test. The test needs as it's Given: a project, and at least two users, one who has access and one who does not. The When action is an attempt to view the project show page, and the Then specification is the successful or unsuccessful page view.

Here's the pair of tests:

security/02/gatherer/test/integration/user_and_role_test.rb

```
test "a user who is part of a project can see that project" do
  project = Project.create(name: "Project Gutenberg")
  project.roles.create(user: @user)
  log_in_as(@user)
  visit(project_path(project))
  assert_equal project_path(project), current_path
end

test "a user who is not part of a project can not see that project" do
  project = Project.create(name: "Project Gutenberg")
  log_in_as(@user)
  visit(project_path(project))
  refute_equal project_path(project), current_path
end
```

The tests both create a project, log in as a user, and visit the project page. In the first test, we also add the user to the project. In the second test, we assert that the user goes anywhere but the actual project page.

As with our last pair of security tests, the test that assumes no blocking behavior passes, while the blocking test does not.

In writing these tests, we have a new concept to represent in the code: the combination of a user and a project. This is a design decision. As we write the test, we're making a claim about how we want the application data to be structured. Often just planning the test will expose the need for new data or new structures and something as simple as this migration might be done before writing the test. That's fine, as long as the larger idea of using the test process to drive the design of the code still holds.

We've added the concept of roles to handle the list of users attached to a project. The test suggests that project has a relationship to roles—in fact, the first point of failure for the test, will be that call to project.roles.

We could create a whole round of integration testing to drive the addition of a user interface for adding users to projects, and allow the data model to be driven as part of that process. Or, we could hand wave the UI, add a basic data model and focus on the security aspect.

In a full project, the add user to project story would get full treatment. For our purposes, we can start with just a migration and a new model. We'll assume somebody else did the add user to project story.

Let's create the migration.

```
$ rails generate model role user:references project:references role_name:string
$ rake db:migrate
```

Which gives us the following migration:

security/02/gatherer/db/migrate/20140617021340_create_roles.rb

```
class CreateRoles < ActiveRecord::Migration
  def change
    create_table :roles do |t|
      t.references :user, index: true
      t.references :project, index: true
      t.string :role_name

      t.timestamps
    end
  end
end
```

This goes a tiny bit beyond the test by adding a role_name.

We'll also need to add the associations. To Project:

security/02/gatherer/app/models/project.rb

```
has_many :roles
has_many :users, through: :roles
```

And User.

security/02/gatherer/app/models/user.rb

```
has_many :roles
has_many :projects, through: :roles
```

At this point, the test has the failure we expect, which is that we want to block access to a user who is not part of the project, but we do not.

Time to move to unit tests to drive that logic. But where do the test go? We have more design decisions to make.

There are two distinct responsibilities in blocking a user. There's the actual logic to determine whether a user can see a project and there's the logic to

redirect the user if the access checker fails. Let's test those responsibilities separately.

Let's deal with the access control first. That method is probably either Project#can_be_viewed_by?(user) or User#can_view?(project). The active voice version in User seems clearer.

Right now we're testing just the access logic, which means that all we need for each test is one user and one project:

security/02/gatherer/test/models/user_test.rb

```
require 'test_helper'

class UserTest < ActiveSupport::TestCase

  test "can not view a project it is not a part of" do
    user = User.new
    project = Project.new
    refute user.can_view?(project)
  end

  test "can view a project it is a part of" do
    user = User.create!(email: "user@example.com", password: "password")
    project = Project.create!(name: "Project Gutenberg")
    user.roles.create(project: project)
    assert user.can_view?(project)
  end

end
```

The two cases here are very similar to the integration tests. Either the user is a member of the project and can see it, or the user is not, and can't.

The second test has one subtle point about how Rails handles associations. In the first test, where the user is not a member of the project, we don't need to save them to the database to run the test. In the second test, we do need to save the user, the project, and the role to the database.

We need to save them all to the database because of how Rails handles associations, specifically has_many through: associations such as the relationship between projects and users in this example. If the associations were just ordinary has_many associations, Rails would be able to manage the two-way relationship in memory without touching the database (at least, Rails 4 can, older versions don't do that either). However, when there is join relationship denoted by has_many: through, Rails internals will always need to have saved objects with ids to resolve the relationship. While you can work around this problem (and we'll discuss some workarounds when we talk about fast tests

in *the (as yet) unwritten chp.environment*I don't know how to generate a cross reference to chp.environment), for the moment, the workarounds are more complicated and distracting than the straightforward creation of the data.

The method in User to make the tests pass is:

security/02/gatherer/app/models/user.rb
```
def can_view?(project)
  projects.to_a.include?(project)
end
```

A user can view a project if the user has a role in that project.

Although the unit test is passing, the integration test is still failing because we have not incorporated the access check into the controller. That brings in the second kind of logic we need to test, which is that the controller successfully blocks access to the page if the user does not have access.

The controller gets two tests, one for the has access condition, one without.

security/02/gatherer/test/controllers/projects_controller_test.rb
```
test "a user who is part of the project can see the project" do
  project = Project.create(name: "Project Runway")
  @controller.current_user.stubs(can_view?: true)
  get :show, id: project.id
  assert_template :show
end

test "a user who is not part of the project can not see the project" do
  project = Project.create(name: "Project Runway")
  @controller.current_user.stubs(can_view?: false)
  get :show, id: project.id
  assert_redirected_to new_user_session_path
end
```

The great thing about this pair of tests is that they both stub the can_view? method of the current_user. This is a perfect use case for mock objects. The details of can_view? are complicated to set up, subject to change, and ultimately irrelevant to the controller's behavior. Stubbing the response not only eliminates the need for details, but stub declaration is much clearer and on-point for reading the test and figuring out what is happening. And the test is robust against future changes in the can_view? logic.

To make this test pass, we need to incorporate the check of can_view? into the controller. There are a couple of ways to make this work, here's one:

security/02/gatherer/app/controllers/projects_controller.rb
```
def show
  @project = Project.find(params[:id])
```

```
  unless current_user.can_view?(@project)
    redirect_to new_user_session_path
    return
  end
end
```

This controller method redirects back to the login screen and returns if the user is blocked, otherwise it continues normally.

With this code in place, not only does the controller pass, but the integration test passes as well.

We're all green.

More Access Control Testing

The advantage of splitting responsibility and testing into separate controller and model concerns becomes even clearer as we add another requirement.

Let's allow for the possibility of administrative users who can see any project, and public projects that can be seen by any user.

We'll want to represent these properties in the database—in this case, we're doing the design work based on planning our test.

security/03/gatherer/db/migrate/20140621051744_add_public_fields.rb

```
class AddPublicFields < ActiveRecord::Migration
  def change
    add_column :projects, :public, :boolean, default: false
    add_column :users, :admin, :boolean, default: false
  end
end
```

Let's think about where this needs to be tested. The behavior of the User#can_view? method obviously needs to change. But since we've isolated the controller from the details of can_view? the controller logic doesn't change—it's still if the user can view let them, redirect otherwise.

So we can write the logic as unit tests:

security/03/gatherer/test/models/user_test.rb

```
test "an admin user can view a project" do
  user = User.new
  user.admin = true
  project = Project.new
  assert user.can_view?(project)
end

test "a public project can be seen by anyone" do
  user = User.new
```

```
  project = Project.new
  project.public = true
  assert user.can_view?(project)
end
```

These tests straightforwardly set up the passthrough condition for administrators and public projects—the negative condition is already covered by the "user can not view an unrelated project" test.

And the test passes with an additional line in the method.

security/03/gatherer/app/models/user.rb

```
def can_view?(project)
  return true if admin? || project.public?
  projects.to_a.include?(project)
end
```

We're at a refactoring step, but we seem pretty clean at the moment

There's an open question in our testing strategy, which is whether we should have started the process of adding admin users and public projects with an end-to-end integration test.

The answer to the question is dependent on the goal of your tests.

From a TDD-integration test perspective, we don't need an integration test, because the integration logic didn't change. The code changes we made were localized to a single class, so the behavior of any code that uses that class is unchanged. While we could write an integration test that would expose this behavior, this test would be slower than the unit tests we just wrote, and it would be harder to diagnose failures.

That said, you may be in a situation where there is value in writing the test strictly as an acceptance test, to verify behavior as part of a set of requirements, rather than to drive development.

Prescription 23 Write your test as close as possible to the code logic that is being tested.

Using Roles

Now that we have the concept of users and roles in the system, we need to look at other places where users need access to a project. Two interesting places spring to mind.

- The project index list should be limited to only the projects that the user can see.

- Adding tasks should be limited to only the projects a user can see.

Let's take a look at the index page. There are two places that need code here. A User instance needs some way to return the list of projects the user can see, and the ProjectsController#index action needs to call that method. That argues for an integration test, though only weakly (sometimes I'll skip an integration test if the logic is a) very close to Rails default integration and b) would very easily be caught manually).

security/04/gatherer/test/integration/user_and_role_test.rb

```
test "a user can see projects they are a part of on the index page" do
  my_project = Project.create!(name: "My Project")
  my_project.roles.create(user: @user)
  not_my_project = Project.create!(name: "Not My Project")
  log_in_as(@user)
  visit projects_path
  assert_selector "#project_#{my_project.id}"
  refute_selector "#project_#{not_my_project.id}"
end
```

This tests creates two projects, adding one to our user. On visiting the index page, we expect to see the project we've added and not to see the project we haven't.

This test fails on the last line because we haven't actually implemented the restrictions yet.

On the user side, this functionality would be trivial if it weren't for admin users and public projects. As it is, though, this logic is parallel to the can_view? logic.

Here are the unit tests for the User behavior of being able to access a list of projects. Again, I wrote them and passed them one at a time:

security/04/gatherer/test/models/user_test.rb

```
def create_two_projects
  Project.delete_all
  @user = User.create!(email: "user@example.com", password: "password")
  @project_1 = Project.create!(name: "Project 1")
  @project_2 = Project.create!(name: "Project 2")
end

test "a user can see their projects" do
  create_two_projects
  @user.projects << @project_1
  assert_equal(@user.visible_projects, [@project_1])
end

test "an admin can see all projects" do
```

```
  create_two_projects
  @user.admin = true
  assert_equal(@user.visible_projects, [@project_1, @project_2])
end

test "a user can see public projects" do
  create_two_projects
  @user.projects << @project_1
  @project_2.update_attributes(public: true)
  assert_equal(@user.visible_projects, [@project_1, @project_2])
end

test "no dupes in project list" do
  create_two_projects
  @user.projects << @project_1
  @project_1.update_attributes(public: true)
  assert_equal(@user.visible_projects, [@project_1])
end
```

The last test is the direct result of realizing that the code as I left it following the test before would duplicate an entry if a project was both public and had the user as a member.

One other tricky thing here is the Project.delete_all in the setup method. That clears out the projects defined in the fixtures. Without that line, the admin test would fail because Project.all would return not just the two projects created for the test, but also all the projects created by the fixtures, which throws off the total set of projects.

Here's the "fast to green" passing code:

security/04/gatherer/app/models/user.rb

```
  def visible_projects
    return Project.all.to_a if admin?
    (projects.to_a + Project.where(public: true).to_a).uniq.sort_by(&:id)
  end
end
```

First off in our refactor, let's move the Project logic to its own class method:

security/04/gatherer/app/models/project.rb

```
def self.all_public
  where(public: true)
end
```

Pro tip: don't name that Project method public, it'll cause a name collision with Ruby. This makes the visible_projects method look like this:

security/05/gatherer/app/models/user.rb

```
  def visible_projects
    return Project.all.to_a if admin?
```

```
    (projects.to_a + Project.all_public).uniq.sort_by(&:id)
  end
end
```

The User code has a larger problem, which is that it now has two methods, can_view? and visible_projects that duplicate the logic of whether a user can view a project. One possible solution would be to rewrite can_view? in terms of visible_projects.

security/05/gatherer/app/models/user.rb

```
def can_view?(project)
  visible_projects.include?(project)
end
```

If we do that, we get some test failures—our tests that did not save a project to the database, now need to. Actually, we have a test refactoring here—two sets of tests covering the same logic. Let's combine them to make it even clearer that the two methods are in parallel.

security/05/gatherer/test/models/user_test.rb

```
require 'test_helper'

class UserTest < ActiveSupport::TestCase

  def create_two_projects
    Project.delete_all
    @user = User.create!(email: "user@example.com", password: "password")
    @project_1 = Project.create!(name: "Project 1")
    @project_2 = Project.create!(name: "Project 2")
    @all_projects = [@project_1, @project_2]
  end

  def assert_visiblity_of(*projects)
    assert_equal(@user.visible_projects, projects)
    projects.all? { |p| assert(@user.can_view?(p)) }
    (@all_projects - projects).all? { |p| refute(@user.can_view?(p)) }
  end

  test "a user can see their projects" do
    create_two_projects
    @user.projects << @project_1
    assert_visiblity_of(@project_1)
  end

  test "an admin can see all projects" do
    create_two_projects
    @user.admin = true
    assert_visiblity_of(@project_1, @project_2)
  end
```

```
  test "a user can see public projects" do
    create_two_projects
    @user.projects << @project_1
    @project_2.update_attributes(public: true)
    assert_visiblity_of(@project_1, @project_2)
  end

  test "no dupes in project list" do
    create_two_projects
    @user.projects << @project_1
    @project_1.update_attributes(public: true)
    assert_visiblity_of(@project_1)
  end
end
```

What we've done here is added an assertion function, assert_visibility_of, which validates the behavior of can_view? and visible_projects in parallel, thereby asserting that the two methods stay in synch. (In RSpec, I would use a matcher, you can see an example in the sample code at code/security/05/gatherer/spec/models/user_spec.rb).

At this point, our original integration test is still failing, because we still have not integrated the controller with the new model function. We already have a controller index test that we'll need to adapt. We need to focus on specifying the behavior which is unique to the controller, which is that it calls the visible_projects method of the current user.

security/05/gatherer/test/controllers/projects_controller_test.rb

```
test "the index method displays all projects correctly" do
  user = User.new
  project = Project.new(:name => "Project Greenlight")
  @controller.expects(:current_user).returns(user)
  user.expects(:visible_projects).returns([project])
  get :index
  assert_equal assigns[:projects].map(&:__getobj__), [project]
end
```

This tests sets up a user, uses a Mocha expectation to ensure that the user is the Devise current_user, then sets a Mocha expectation that visible_projects is called on that user. This test is simpler than the previous index test—since the integration test is managing the expected response, we no longer need to handle the view logic in this test. All we need to do is verify that the controller is calling the correct model method and passing expected values to the view. The last line of the test describes the contract with the model—the expected list of projects as returned by the visible_projects stub is passed to assigns[:projects], plus a little bit of manipulation because we are actually using a decorator wrapped around the project objects.

The test fails because the controller method still calls Project.all, but we can fix that.

security/05/gatherer/app/controllers/projects_controller.rb

```
def index
  @projects = ProjectPresenter.from_project_list(current_user.visible_projects)
end
```

At this point, the new integration test also passes, but we have an interesting regression failure:

```
  1) Failure:
AddProjectTest#test_a_user_can_add_a_a_project_and_give_it_tasks
 [gatherer/test/integration/add_project_test.rb:20]:
expected to find css "#project_980190963 .name"
with text "Project Runway" but there were no matches
```

Investigating, we find that the test in question is an integration test that creates a new project using CreatesProject and then goes to view the page. The test is failing because the user is not added as a member of the new project, therefore the user can't see the project page.

Fixing this involves changing the CreatesProject action by allowing it to take a user or users and adding them to the project when the project is created. Since CreatesProject is an action, we can isolate the test and just make sure that a passed user is applied to the project—or in this case, we set the API to take an array of users.

security/05/gatherer/test/actions/creates_project_test.rb

```
test "adds users to the project" do
  creator = CreatesProject.new(name: "Project Runway",
      users: [users(:user)])
  creator.build
  assert_equal [users(:user)], creator.project.users
end
```

Then we need to add the new keyword argument to the action and use the value when creating the new project.

security/05/gatherer/app/actions/creates_project.rb

```
attr_accessor :name, :task_string, :project, :users

def initialize(name: "", task_string: "", users: [])
  @name = name
  @task_string = task_string
  @users = users
end

def build
  self.project = Project.new(name: name)
```

```
  project.tasks = convert_string_to_tasks
  project.users = users
  project
end
```

Then we need to update the controller method to pass the user to the action.

security/05/gatherer/app/controllers/projects_controller.rb

```
def create
  @action = CreatesProject.new(
    name: params[:project][:name],
    task_string: params[:project][:tasks] || "",
    users: [current_user])
  success = @action.create
  if success
    redirect_to projects_path
  else
    @project = @action.project
    render :new
  end
end
```

Which breaks the mock that we use in the controller test to bypass the action—which I admit is more fumbling in the code base than I thought we were going to have to do when I started this example.

security/05/gatherer/test/controllers/projects_controller_test.rb

```
test "the project method creates a project (mock version)" do
  fake_project = mock(create: true)
  CreatesProject.expects(:new)
      .with(name: "Runway", task_string: "start something:2",
          users: [users(:user)])
      .returns(fake_project)
  post :create, project: {name: "Runway", tasks: "start something:2"}
  assert_redirected_to projects_path
  refute_nil assigns[:action]
end
```

After all that, we have basic user and roles authentication in the system. Now we need to protect against a couple of attacks that require the user to not use our applications UI directly.

Prescription 24 Adding user authentication can be very disruptive of existing tests. Try to get the basic infrastructure in place early.

Protection against form modification

We have at least one blind spot in our user and role protection. The project show page has a form that submits a new task. That form is submitted to the

TasksController, which does not do any user access control. The use case here is a malicious user not going through the web UI, but creating their own HTTP request and pointing it at the server.

There are two important issues here, at least from my perspective as Rails Testing Author Guy. First is the habit of noticing when you are using a resource that is being accessed due information that comes in from a user request, as opposed to being stored server-side. This is even true when the resource is protected indirectly, as in this case, where we are accessing a Task, which belongs to the actual Project, which is where the access control is attached. Second, we need to discuss how to test such a case.

We have two similar cases to deal with—task creation from the project form via TaskController#create and any of the update and move task methods in the controller.

Let's plan our create test. For Givens, we need a user, a project that the user belongs to and a project the user does not belong to. The When is the creation of the task, and the Then is the actual creation or non-creation of the task.

The design question before us is where to put the access check, and by extension where to write the test. We're at a slight disadvantage, since the potentially malicious request is coming from outside the UI, Capybara isn't going to be effective in crafting an integration test. The code logic for creating tasks is in the TaskController, and that seems the most likely place for an access control test. (Though if we had moved the creation logic to an action item, the way that we did with CreatesProject, then we could put the access logic there.)

The test class already has sign_in users(:user) in its setup, taking the user object from the fixture and logging it in, so all we need to do is create a project and see what happens:

security/05/gatherer/test/controllers/tasks_controller_test.rb

```
test "a user can create a task for a project they belong to" do
  project = Project.create!(name: "Project Runway")
  users(:user).projects << project
  users(:user).save!
  post :create, task: {
      project_id: project.id, title: "just do it", size: "1" }
  assert_equal project.reload.tasks.first.title, "just do it"
end

test "a user can not create a task for a project they do not belong to" do
  project = Project.create!(name: "Project Runway")
  post :create, task: {
```

```
      project_id: project.id, title: "just do it", size: "1" }
  assert_equal project.reload.tasks.size, 0
end
```

The trickiest part in this test is remembering that you need to reload the project in order to see if any tasks have been added to it by the controller, since the controller will create a local ActiveRecord instance backed by the same database row and will update the database row without changing the ActiveRecord instance that is local to the test.

To make the test pass, we just need to add a bit of logic to the controller action:

security/05/gatherer/app/controllers/tasks_controller.rb

```
def create
  @project = Project.find(params[:task][:project_id])
  unless current_user.can_view?(@project)
    redirect_to new_user_session_path
    return
  end
  @project.tasks.create(title: params[:task][:title],
      size: params[:task][:size],
      project_order: @project.next_task_order)
  redirect_to @project
end
```

The new part is the unless statement, which checks to see if the current_user can actually see the project in question. We can trust the current_user value because it is not dependent on any data coming from the user, it's managed by the Rails session.

There are a lot of possibilities from here. We could extract the current controller method to a CreatesTask action item, which would make it easier to separate the access logic from the rest of the code. We could also add similar protection to the update, up, and down methods, which involves the design question of modeling access control from the task's perspective.

Mass Assignment Testing

Mass-assignment is a common Rails security issue, caused by Rails' ability to save an arbitrary hash of attribute names and values to an instance by sending an entire has as a parameter, as in new(params[:user]), create(params[:user]), or update_attributes(params[:user]). The security issue happens when somebody customizes a request and adds unexpected attributes to the incoming parameters, typically an attribute that you wouldn't want an arbitrary user to be able to change, like User#admin, or Project#public. (GitHub was famously

hacked via this vector by a user who added themselves as a committer to the Rails repo.)

Rails 4 provides the concept of *strong parameters* to allow you to identify parts of the parameter hash from an incoming request as required or permissible. In order to be used in a mass-assignment, the attributes need to be identified using the require or permit methods of the Rails parameter object. Attributes which are not whitelisted are not passed on to the ActiveRecord object, and are helpfully listed in the Rails log as a warning and to make debugging these issues easier.

Our Gatherer application currently uses strong parameters in one location, TasksController#update, where we have the line:

```
@task.update_attributes(params[:task].permit(:size, :completed_at))
```

We manage the strong parameters issue in other ways. The TasksController#create method explicitly lists the items that are being passed to create:

```
@project.tasks.create(title: params[:task][:title],
        size: params[:task][:size],
        project_order: @project.next_task_order)
```

In the ProjectsController, the create method similarly lists the attributes that are passed to the CreatesProject action object, which explicitly assigns the one attribute it uses when building the project.

As with other features provided by the framework, the important part from the testing framework is the behavior—preventing the user from setting a particular attribute—rather than the implementation.

Prescription 25

Test for mass assignment any time you have an attribute that needs to be secure and a controller method that touches that class based on user input.

As it happens, we have sitting in the Gatherer code at the moment, ProjectsController#update, which is not using any kind of strong parameter protection, and which we added solely for the purposes of showing mock objects, meaning the parameters have never been tested.

I wonder if a user could maliciously make a project public?

security/05/gatherer/test/controllers/projects_controller_test.rb

```
test "a user can make a project public" do
  sample = Project.create!(name: "Test Project", public: false)
  patch :update, id: sample.id, project: {public: true}
  refute sample.reload.public
```

```
end
```

Turns out the answer is no:

```
  1) Error:
ProjectsControllerTest#test_a_user_can_make_a_project_public:
ActiveModel::ForbiddenAttributesError: ActiveModel::ForbiddenAttributesError
```

But it's no because none of the attributes are permitted. ForbiddenAttributesError means that the params object is being used without any attempt to set permitted attributes. Meaning that nobody could make a project public via this method even if they had access.

To make the test pass, we just need to change one line in the project controller to add the permit call.

security/05/gatherer/app/controllers/projects_controller.rb

```
def update
  @project = Project.find(params[:id])
  if @project.update_attributes(params[:project].permit(:name))
    redirect_to @project, notice: "'project was successfully updated.'"
  else
    render action: 'edit'
  end
end
```

That's fine, but what if the mass assignment takes place outside the controller?

We don't do this in the current version of Gatherer, but if we had passed the entire params object to the CreatesProject action, we could easily have something like this:

```
class CreatesProject

  attr_accessor :params, :project, :users

  def initialize(params: {}, users: [])
    @params = params
    @users = users
  end

  def build
    self.project = Project.new(params.permit(:name))
    project.users = users
    project
  end
```

This code has a mass assignment, params.permit(:name), which is fine, but if you try to test this code by passing a regular hash to CreatesProject, the test will error at params.permit because ordinary hashes don't have a permit method.

The workaround is to create the actual underlying Rails object directly:

```
CreatesProject.new(ActionController::Parameters.new(name: "Project"))
```

Rails provides the class ActionController::Parameters which wraps the hash and allows for the permit and require behavior needed to support strong parameters.

Other Security Resources

There's a limit to what you can test with security using TDD. It's a good idea to use a static analysis tool to look for security issues. Two options are Brakeman, available at http://brakemanscanner.org, which you would run yourself, and CodeClimate, http://www.codeclimate.com, which automatically runs Brakeman on each commit. Brakeman looks for a variety of security issues and gives some tips on working around them.

Prescription 26	Use an automatic security scanner to check for common security issues.

The Open Web Application Security Project, https://www.owasp.org/, has all kinds of useful information on security risks. Of particular interest is the WebGoat, a deliberately insecure application designed to allow you to hack and test solutions. A Rails version is available at https://github.com/OWASP/railsgoat.

CHAPTER 13

Testing External Services

It has been decided that the one thing our project management tool really needs is a little bit of graphical spark. Specifically we've been asked to have users' Twitter avatars show up on the site attached to tasks they have completed. (Also because it's got a Ruby gem that's not to hard to set up.) Since this is *Rails 4 Test Prescriptions* and not *Rails 4 Connecting To Twitter Prescriptions*, we'd like to be able to test our interaction with the Twitter API.

Unfortunately, interacting with a third-party web service introduces a lot of complexity to our testing. Connecting to a web service is slow, even slower than the database connections we've already tried to avoid. Plus, connection to a web service requires an internet connection, and we'd like our test suite to be able to run on the train, on a boat, or during a network outage. Some external services are public—we don't want to post an update to Twitter every time we run our tests, let alone post a credit card payment to PayPal. Some services have rate limits, some services cost money to access, some services actually deal with money. Some services require API keys and authentication.

The point is, we'd really, really like to be able to write and execute tests without actually hitting the service more than strictly necessary.

The strategies we have developed so far for test isolation and using mocks to limit the scope of tests will help us successfully test an external service. We will be able to test it on a train and test it on a boat, and test it in the rain and test it on a goat.

External Testing Strategy

Our external service testing story has two main characters:

- The *client*, which is the part of our application code that uses the external API, either because it needs data accessible via the API, or because it is

sending data to the API to be used by somebody else. In either case, we're dealing with a request and a response, even if the response is just a status code.
- The *server*, which for our purposes is outside our application and reachable via some kind of network request. (Though many of the strategies in this chapter also apply even if the service isn't separated by the network.)

We also have two characters we introduce to the story for design and testing purposes:

- A *fake server*. The fake server intercepts HTTP requests during a test and returns a canned response object. We'll be using the VCR gem to manage our fake server.
- An *adapter*, which is an object that sits between the client and the server for the purpose of mediating access between them.

The following diagram shows the relationship between these objects and the tests we're going to write using them:

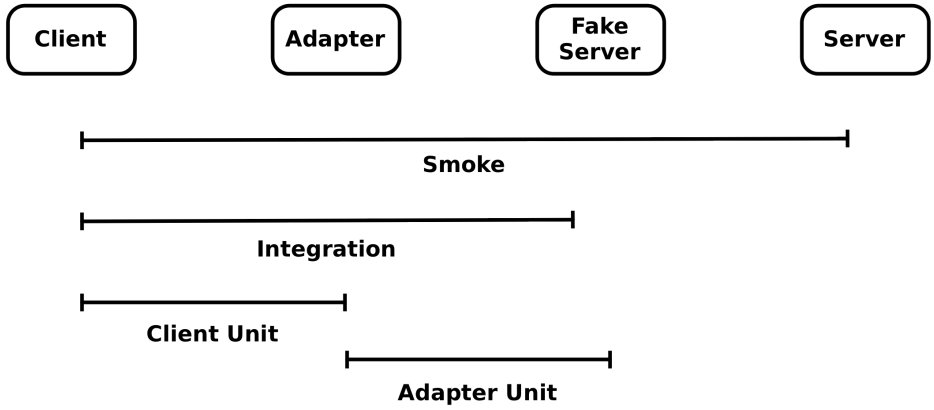


Figure 7—The Objects in a Service Test

There are several different kinds of tests that relate different combinations of these objects:

- A *smoke test*, which goes from the client all the way to the real server. In other words, it's a full end-to-end test of the entire interaction. We don't want to do this often, for all the reasons listed above, but it's useful to be able to guard against changes in the server API.
- An *integration test*, which goes from the client to the fake server. This tests the entire end-to-end functionality of our application, but uses a

stubbed response from the server. This will be our go-to strategy for integration testing the external server.

- A *client unit test*, which starts on the client and ends in the adapter. The adapters responses are stubbed, meaning that the adapter isn't even making fake server calls. This allows us to unit test our client completely separate from the server API.
- An *adapter unit test*, which starts in the adapter, and ends in the fake server. These tests are the final piece of the chain, and allow us to validate the behavior of the adapter separate from any client or the actual server.

Prescription 27 Mediating interaction to an external server through an adapter that is part of your code makes the interaction easier to test and to use.

Our Service Integration Test

We're going to use the Twitter gem, https://github.com/sferik/twitter, to interface with Twitter. We will put that in the Gemfile. We're also going to need the VCR and Webmock gems in our test environment:

```
gem 'twitter'
gem 'vcr', group: :test
gem 'webmock', group: :test
```

And reinstall the bundle with bundle install.

We need a Twitter API key and secret key. In Rails 4, those get placed in the secrets.yml file, which typically is not stored in your code repository, though I've put it in our sample code for ease of setup.

external/01/gatherer/config/secrets.yml

```
development:
  secret_key_base: |
    9cbb1cdd81bd1e79999d8f91ec57b0ced0e59ec961f9af56e77a5e514acf7cc9f646e83aedd
    5293f44685c47e717eade8677fca153d9c65ffb441d4fdff33052

test:
  secret_key_base: |
    9cbb1cdd81bd1e79999d8f91ec57b0ced0e59ec961f9af56e77a5e514acf7cc9f646e83aedd
    5293f44685c47e717eade8677fca153d9c65ffb441d4fdff33052
  twitter_api: "IVCwuly8UuymHROaIwsJ4IcgK"
  twitter_secret: "ZzfZNLEF72ELnELH3jsRG41AY0cLfP2QfAyKMpChxNfJOm1oqw"


production:
  secret_key_base: <%= ENV["SECRET_KEY_BASE"] %>
```

We'll use those keys when we connect to the Twitter API. By the way, those keys will have long since been changed by the time you read this. The existing tests might work, because they will have been stubbed via VCR. But you also might want to go to https://apps.twitter.com and generate your own application and set of keys.

We want to attach the current user to a task when the task is completed, and then to show the user's Twitter avatar next to that task in the project page. As has been our fashion so far, we're going to hand wave over some UI functionality that doesn't relate to our focus. In this case, we're going to pretend that the pair down the hall has already covered the ability to connect a user to a task, and we're just going to add the data migration. This migration includes the connection between a user and a task and also ads the user's Twitter handle, which we need to access their avatar via the Twitter API:

external/01/gatherer/db/migrate/20140629211718_add_user_to_task.rb
```
class AddUserToTask < ActiveRecord::Migration
  def change
    add_column :tasks, :user_id, :integer
    add_column :users, :twitter_handle, :string
  end
end
```

Don't forget to run the migration. No really, don't forget. As I was writing this chapter, I added the migration then started running tests without ever running the migration. Which doesn't work.

```
$ rake db:migrate
```

We need to the association, in app/models/task.rb:

```
belongs_to :user
```

And, for completeness sake, in app/models/user.rb:

```
has_many :tasks
```

With that little bit of setup out of the way, we can write an integration test. In this case, integrating our code with Twitter.

external/01/gatherer/test/integration/shows_twitter_avatar_test.rb
```
require "test_helper"

class TaskShowsTwitterAvatar < Capybara::Rails::TestCase

  include Warden::Test::Helpers

  setup do
    projects(:bluebook).roles.create(user: users(:user))
```

```
    users(:user).update_attributes(twitter_handle: "noelrap")
    tasks(:one).update_attributes(user_id: users(:user).id,
        completed_at: 1.hour.ago)
    login_as users(:user)
  end

  test "i see a gravatar" do
    VCR.use_cassette("loading_twitter") do
      visit project_path(projects(:bluebook))
      url = "http://pbs.twimg.com/profile_images/40008602/head_shot_bigger.jpg"
      within("#task_1") do
        assert_selector(".completed", text: users(:user).email)
        assert_selector("img[src='#{url}']")
      end
    end
  end
end
```

With one exception, this test is a simplified version of the integration test we wrote in Chapter 11, *Integration Testing with Capybara and Cucumber*, on page 179. Our Given here is the project, user, and task which are already defined in our fixtures, plus a simulated login. Our When is visiting the show page for the project, and then we validate that the user associated with the task is displayed, both by their email and by their Twiter avatar. I've actually seeded this test with a known Twitter avatar—mine—by setting the user's twitter handle, and then setting the expected image source to my known Twitter profile url.

You've probably noticed that the test is wrapped in an unfamiliar method call and block, `VCR.use_cassette("loading_twitter") do`. Since I've mentioned VCR a couple of times in this chapter, you may have further assumed that, in this context, VCR has more to do with external service testing than it does with that Blockbuster store in the strip mall that closed seven years ago.

VCR

VCR is one of my favorite testing tools. The concept is simple. When VCR is enabled, it intercepts any third-party HTTP request. By default, the first time the request is made, VCR allows the request to proceed normally. However, VCR saves the response and associated metadata to a YAML file, which VCR calls a *cassette*. When the test is run again, VCR intercepts the request. Rather than actually making the request as a network call, VCR converts the cassette back into a response object and returns that response object. By default, if you then make an HTTP request that the VCR cassette doesn't know about, the test will fail.

Using VCR has many great features. Because the data that VCR stores is based on a real request, it's real data, meaning that your tests are not running on some slapdash mock object you put together. This makes the tests have greater fidelity to the runtime environment, which makes them more trustworthy. VCR is super-simple to use and basically just works. (The main exception to "it just works" is during the initial writing of the test, if the set of requests in a VCR cassette changes, the cassette file may have to be recreated.) You can even set VCR to automatically regenerate the cassette on arbitrary time frame to protect against changes in the API.

Prescription 28 Use the VCR gem to allow your integration tests to run against actual server response data.

VCR is very configurable, allowing you to specify URL's or patterns that should be allowed to pass through without VCR caring (a common one being localhost). You can also do some pattern matching as to what constitutes the same URL for VCR purposes—many API's have some kind of timestamp parameter that you'd want VCR to overlook for the purposes of returning stubbed output.

VCR and Minitest

To make VCR work in our Minitest environment, we need to add some configuration to the test_helper.rb file:

external/01/gatherer/test/test_helper.rb

```
VCR.configure do |c|
  c.cassette_library_dir = 'test/vcr'
  c.hook_into :webmock
end
```

We are specifying two options to start, the directory where VCR is going to place its cassette files, which can be anything you want, but something like test/vcr is customary.

VCR handles the creation and use of the cassette files, but it subcontracts the actual stubbing of HTTP calls to another library, which you can specify, we're using webmock, but that's an implementation detail and is the last time we're going to talk about webmock.

To use a cassette in Minitest, you surround some code with the method VCR.use_cassette which takes a string argument and a block. The argument is the name of the cassette file, so it needs to be unique, and the block is the body of the test that is being recorded by that cassette. There are some options that you can pass to the use_cassette method which we'll get to in a bit.

Inside the block, VCR behaves in one of two ways.

VCR may think that the cassette needs to be recorded. This will most commonly be because the cassette doesn't exist yet, but there are ways to tell VCR to overwrite an existing cassette. If VCR is trying to record a cassette, VCR will allow all HTTP requests to happen, but will save all the requests and responses in a single file, which it will store in the directory specified in the config and with the filename specified by the argument to use_cassette.

If VCR does not believe that the cassette needs to be recorded, it will act in play mode. In play mode, any HTTP request that matches a request in the cassette file will be intercepted, and its response will be crafted by the data on the cassette. Any HTTP request that does not match a request in the cassette file will trigger an error and a test failure. By default, VCR will also fail if there is a request in the cassette file that is not matched during the test, behaving like a mock expectation. If there are multiple requests in the cassette file that match, VCR will use them one at a time in sequence as there are further requests to the same URL during the test.

VCR and RSpec

The setup for VCR in RSpec land is a little bit different. You can use RSpec metadata to specify that any RSpec it or describe block uses a VCR cassette. The configuration goes into the rails_helper.rb file, and contains one extra line:

external/01/gatherer/spec/rails_helper.rb

```
VCR.configure do |c|
  c.cassette_library_dir = 'spec/cassettes'
  c.hook_into :webmock
  c.configure_rspec_metadata!
end
```

It will probably not come as any surprise that the extra line c.configure_rspec_metadata! configures the RSpec metadata.

Then we can specify VCR as a metadata option in the spec. The entire spec in RSpec looks like this:

external/01/gatherer/spec/features/shows_twitter_avatar_spec.rb

```
require 'rails_helper'

include Warden::Test::Helpers

describe "task display" do

  fixtures :all

  before(:each) do
```

```
    projects(:bluebook).roles.create(user: users(:user))
    users(:user).update_attributes(twitter_handle: "noelrap")
    tasks(:one).update_attributes(user_id: users(:user).id,
        completed_at: 1.hour.ago)
    login_as users(:user)
  end

  it "shows a gravatar", :vcr do
    visit project_path(projects(:bluebook))
    url = "http://pbs.twimg.com/profile_images/40008602/head_shot_bigger.jpg"
    within("#task_1") do
      expect(page).to have_selector(".completed_by", text: users(:user).email)
      expect(page).to have_selector("img[src='#{url}']")
    end
  end

end
```

When we declare the test, we also declare :vcr as metadata. This is equivalent to placing the entire spec inside a VCR.use_cassette block. The cassette is automatically named using the names of the describe blocks as directory names and the name of the it block as a file name. If you want to pass options to the VCR call, then you would change the metadata declaration from :vcr to vcr: {options}.

Alternately, you can still use VCR.use_cassette inside RSpec specs just as we did in Minitest.

VCR and Cucumber

To use VCR with cucumber, first place the same configuration code in a file somewhere in the Cucumber features/support directory. Once that is done, you have two options. You could put VCR.use_cassette calls inside step definitions, or you can use Cucumber tags.

To use tags, you need to define them, by writing additional configuration code, presumably in the same file you put the basic configuration. The tag configuration will look like this:

```
VCR.cucumber_tags do |t|
  t.tag '@vcr', use_scenario_name: true
  t.tags '@twitter', '@facebook'
end
```

Any options you would pass to use_cassette can be passed as key value pairs after the tag name. If you want to define multiple tags at once, you can use the tags method.

The tags can then be used like normal Cucumber tags:

```
@vcr
Scenario: Get the user's Twitter avatar
  Given a logged in user
  When that user has completed a task
  And I view the project page
  Then I should see the user's Twitter avatar
```

When a VCR related tag is used for a Cucumber scenario, VCR is activated for the duration of the scenario. The resulting cassette file is named after the tag, unless the use_scenario_name: option is true, in which case VCR generates a name based on the feature and scenario names, similar to how it generates a name when using RSpec meta data.

Client User tests

VCR is set up, now let's make the Twitter integration work. As it stands, the test fails because the user data is not in the view at all. The test suggests that the user email and the Twitter avatar should be in the view, let's add those to the view file.

We have a design decision to make about how our application should interact with Twitter. We have many options, ranging from calling the gem and service directly from the view, or placing the interaction within the User class.

My design here tends toward more objects and structure, on the grounds that we're using this avatar to stand in for a more complex third party integration. So, the set of classes might feel like overkill, but I want us to see what this is like with all the moving parts.

With that throat-clearing out of the way, let's write some tests. It seems like getting the Twitter avatar image URL is a User responsibility.

The relevant view code looks like this, in app/views/projects/show.html.erb

```
<td class="completed">
  <% if task.complete? %>
    <%= task.user.email %>
    <%= task.completed_at.to_s %>
    <img src="<%= task.user.avatar_url %>" />
  <% end %>
</td>
```

If I was being stricter, I'd that the avatar_url is only a view-level responsibility. Which would imply a decorator, the same way we have one already in the code base for the ProjectsController#index action. We'll hold that thought for a possible refactoring.

The test now fails at because user.avatar_url does not exist. Let's write tests for that method.

We're now writing client unit tests, which assume the existence of an adapter, but typically stub the adapter. We can use this test as a place to design the adapter's interface with the rest of the application. In this case, the logic from the user object's perspective is simple. We pass the user's email to the adapter and expect to call a method on the adapter which returns the avatar url.

The test looks like this:

external/01/gatherer/test/models/user_test.rb

```
test "can get a twitter avatar URL" do
  user = User.new(email: "test@example.com")
  fake_adapter = mock("avatar").responds_like_instance_of(AvatarAdapter)
  fake_adapter.expects(:image_url).returns("fake_url")
  AvatarAdapter.expects(:new).with(user).returns(fake_adapter)
  user.avatar_url
end
```

This test does not depend in any way on the actual HTTP request, instead, it defines the API of the adapter. The test creates a User instance, then creates a fake adapter using Mocha's mock method, qualified with responds_like_instance_of to ensure that any method we stub actually exists in the AvatarAdapter class. Then we set the expectation that the avatar will receive the image_url method.

Prescription 29 Use the adapter to test client behavior without being dependent on the actual server API.

The When part of this test is the last line, the call to the user.avatar_url method. At this point, we've set up the following expectations via Mocha for what happens next.

- The AvatarAdapter class will be sent the new method with the user instance as an argument.
- The resulting instance of AvatarAdapter will be send the image_url method.

To make this test pass, we need a method in User:

external/01/gatherer/app/models/user.rb

```
def avatar_url
  adapter = AvatarAdapter.new(self)
  adapter.image_url
end
```

And a new AvatarAdapter class:

external/01/gatherer/app/models/avatar_adapter.rb
```
class AvatarAdapter

  def initialize(user)

  end

  def image_url

  end

end
```

Right now the AvatarAdapter class is just a skeleton, which is all we needed to make the unit tests pass.

Now it's time to get that adapter done.

Why An Adapter?

Using an adapter class to mediate interaction with the external service is a good idea even when, like Twitter, the external service already has a Ruby gem. The adapter encapsulates logic that is specific to the interaction between your application and the service.

An adapter is useful the more your code has any or all of the following qualities:

- The external service will be accessed from multiple points in your code.
- The interaction with the external service has logic of its own, such as authentication or type changing or common sets of options.
- There's a mismatch between the language or metaphor of the API and the domain terms and structures of your code.

Our Twitter example doesn't have the first one, at least not yet. We do have the second feature, the adapter needs to manage a Twitter client object that nothing else in the application needs to care about or be aware of the existence of. It also manages an argument to the Twitter API, the :bigger argument which specifies the size of the image to download.

Whether we have a mismatch between the API and our code is a matter of interpretation, but I think we do. At the very least, the Twitter client exposes a lot of data that our application doesn't care about, so limiting access to the full set of Twitter data seems like a good idea.

My experience with adapters like the one we've written is that they tend to attract functionality as you use them, with the side effect that it's much eas-

ier for a full range of complexity to be available at each use. For example, if we allow the adapter to take an argument to image_url to represent the size, then that ability automatically is available whenever the adapter is used. This is especially valuable for security and error handling, which are easy to leave off when you are creating each connection separately.

Adapter tests

The adapter tests work between the adapter and the server, using VCR as a medium.

external/02/gatherer/test/models/avatar_adapter_test.rb

```
require 'test_helper'

class AvatarAdapterTest < ActiveSupport::TestCase

  test "accurately receives image url" do
    user = stub(twitter_handle: "noelrap")
    VCR.use_cassette("adapter_image_url") do
      adapter = AvatarAdapter.new(user)
      url = "http://pbs.twimg.com/profile_images/40008602/head_shot_bigger.jpg"
      assert_equal url, adapter.image_url
    end
  end

end
```

This test has no dependency on the client, which we show by passing in a stub rather than an actual User instance. Inside a VCR cassette we create a new adapter, and assert that the adapter provides the expected URL when queried. The test also doesn't have a particular dependency on the Twitter gem, beyond the specific URL value being from Twitter's asset storage. This is a test of the behavior of the adapter, not of the implementation.

The passing code requires a little bit of Twitter connection setup:

external/02/gatherer/app/models/avatar_adapter.rb

```
class AvatarAdapter

  attr_accessor :user, :client

  def initialize(user)
    @user = user
  end

  def client
    @client ||= begin
      Twitter::REST::Client.new(
        consumer_key: Rails.application.secrets.twitter_api,
```

```
        consumer_secret: Rails.application.secrets.twitter_secret)
    end
  end

  def image_url
    client.user(user.twitter_handle).profile_image_uri(:bigger).to_s
  end

end
```

The Twitter gem requires a client to be created using the API keys we put in the secrets.yml file. Our adapter lazily creates that client as needed.

Once the client is created, it calls the user method on the client with the twitter handle, and then grabs the profile_image_uri property. The :bigger argument is used by the gem to determine the size of the resulting image, and we need to call to_s, because the actual property is an internal class of the Twitter gem and all we want is the actual URL. All of these are features of the interaction with the Twitter API that the rest of our application doesn't need to care about because the adapter is managing that information.

At this point, our adapter tests passes. Even better, all the parts of the interaction have now been written and connected, so our integration test also passes, too.

If we go to the test/vcr directory we now see two cassette files, one for each test. They are both about 150 lines long, so I'm not putting them in the book. The first few lines look like this:

```
---
http_interactions:
- request:
    method: post
    uri: <big long URI to twitter>
    body:
      encoding: UTF-8
      string: grant_type=client_credentials
    headers:
      Accept:
      - "*/*"
      User-Agent:
      - Twitter Ruby Gem 5.11.0
      Content-Type:
      - application/x-www-form-urlencoded; charset=UTF-8
      Accept-Encoding:
      - gzip;q=1.0,deflate;q=0.6,identity;q=0.3
  response:
    status:
      code: 200
```

```
message: OK
```

but they each chronicle two HTTP requests to the Twitter API, one for authentication of our API key, and one to get the user data. If you delete the files and try again, you'll hopefully notice that the first test run is slow because the HTTP requests are actually being made, then once the VCR cassette is in place, the test speeds up. (On my machine, the suite went from 2.2 seconds to 0.8 seconds).

Testing for Error Cases

Our application design allows us multiple ways to simulate errors for testing purposes.

- We can make an API call to the actual service that results in an error and capture the result using VCR. This could be used in an integration or adapter test.
- We can stub a method that is internal the adapter. For example, we could stub the client method to return a double that simulates an API error. We would use this in an adapter test.
- We can stub the adapter to return an unexpected value in a client test.

Which you do depends on the details of the library you are working with. Often, stubbing the external service makes sense for the same reason that stubbing ActiveRecord methods does—crafting a call that will reliably return an error is not always possible. If possible, try to stub methods of your adapter rather then methods of the third party gem, it's generally a good idea to only stub code that you control.

Keeping tests consistent with the location of the logic being tested is still a good policy. Use adapter tests to ensure that the adapter behaves gracefully when it gets weird responses from the server (normally, server or gem exceptions shouldn't leak out of the adapter). Use unit tests to make sure that the clients are able to handle whatever the adapter does in response to unexpected input.

Prescription 30	Test the error code based on which object in the system needs to respond to the error.

Smoke tests and VCR options

So far, we've used VCR to record an interaction once and preserve it for all time. VCR provides other options. These options are passed as key/value

arguments to VCR.use_cassette(re_record_interval: 7.days) or, if you are using RSpec metadata, as the value part of the metadata, as in vcr: {re_record_interval: 7.days}.

That re_record_interval that we just used as an example allows you to use the same VCR test as both an integration and smoke test. The re_record_interval is an amount of time. If the requests on the associated cassette are older than the interval, than VCR will attempt to reconnect to the HTTP server and re-record the cassette. If the server is unavailable, VCR will just use the existing cassette.

This allows you to protect your application against API changes in the server. If the server API changes, eventually, your VCR cassette will pick up the change and your test will fail. This can be a useful feature. It can also be a little on the opaque side when a test randomly fails, but better in testing than production.

VCR lets you set different record modes with the :record option. The default, which we've been using thus far, is :once, which records new interactions if they don't exist, but raises an error for new interactions if the cassette already exists.

Here are all the VCR recording options:

Option	Description
:all	Always connect to HTTP and re-record. Useful for forcing cassette updates.
:new_episodes	Replays a request that is already on the cassette. New requests are made via HTTP and added to the cassette.
:none	Only replay existing cassettes, never make an actual request.
:once	Replay existing cassettes, record new requests if the cassette doesn't exist. Raise an error on new requests if it does.

If you want to specify the recording options for all cassettes at once, you can do so in the configuration with the default_cassette_options method.

```
VCR.configure do |c|
  # existing configuration
  c.default_cassette_options(record: :all)
end
```

So, by occasionally adding the above line to the configuration for one test run, you get one run of smoke tests as all the VCR cassettes go back to their servers for data.

There are a couple of other important configuration options.

Sometimes you want VCR to not deal with specific requests. The c.ignore_localhost = true property handles one common case, where you don't want VCR to touch requests back to the actual application being tested, such as Ajax requests, or callbacks for authentication information. The method c.ignore_hosts takes an optional list of hostnames to ignore, and the method ignore_request { |req| CODE } ignores any request for which the associated block returns true.

By default, VCR looks for an exact match on the URI being requested when attempting to match a cassette request to a new HTTP request from the code, including query string and HTTP method. However, some services never have the exact same URI twice. For example, Amazon API calls contain a time stamp. In other cases there may be additional headers that are relevant to how the request is processed. VCR provides the match_requests_on configuration option to manage how the matching between cassette and request works.

You use match_requests_on as an option to a VCR call, or as a default. The argument is a an array of elements that you want to use in the match. Valid values are :method, :uri, :host, :path, :query, :body, :headers, and :body_as_json. You can also use the method uri_without_params as a substitute for uri where there are parameters in the query string that are not important for the match.

For example, the following configuration was used to match all the the dynamic elements in an application that used the Amazon API.

```
VCR.configure do |c|
  c.cassette_library_dir = 'test/vcr'
  c.hook_into :webmock
  c.ignore_localhost = true
  c.default_cassette_options = {
    :match_requests_on => [:method,
      VCR.request_matchers.uri_without_params(
        "Timestamp", "Signature", "AWSAccessKeyId", "AssociateTag")]
  }
end
```

The match_requests_on option is used to match on the combination of HTTP method and URI, but with a several parameters to the URI ignored.

VCR has other options for more elaborate use, you can see the full list in the documentation at https://relishapp.com/vcr/vcr/v/2-9-2/docs.

The World is a Service

Once you get used to the idea of having adapters mediate access between your application and external services, it's not that far a leap to have adapters internally to mediate between different parts of your application. This approach

is sometimes called *Hexagonal Architecture*, and there are many, many resources online describing Hexagonal Architecture as it applies to Rails, such as http://victorsavkin.com/post/42542190528/hexagonal-architecture-for-rails-developers. And many, many resources online saying that Hexagonal is an awful idea. (David Heinemeier Hansson is a particularly vocal critic).

We've taken baby steps in this direction by creating action objects such as CreatesProject, which are somewhat like adapters between the controller and model. Many web frameworks use adapters between model objects and the database. Rails does not, but the pattern is not uncommon.

In the next chapter, as we test JavaScript, we'll also explore the interaction between our application and the browser document object model as another possible site for a service and adapter relationship.

The Pragmatic Bookshelf

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

Visit Us Online

This Book's Home Page
http://pragprog.com/book/nrtest2
Source code from this book, errata, and other resources. Come give us feedback, too!

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

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

New and Noteworthy
http://pragprog.com/news
Check out the latest pragmatic developments, new titles and other offerings.

Buy the Book

If you liked this eBook, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: *http://pragprog.com/book/nrtest2*

Contact Us

Online Orders:	*http://pragprog.com/catalog*
Customer Service:	*support@pragprog.com*
International Rights:	*translations@pragprog.com*
Academic Use:	*academic@pragprog.com*
Write for Us:	*http://pragprog.com/write-for-us*
Or Call:	+1 800-699-7764