Testing Rails

by Josh Steiner

thoughtbot

Testing Rails

Josh Steiner

August 21, 2015

Contents

Introduction

Why test?

As software developers, we are hired to write code that *works*. If our code doesn't work, we have failed.

So how do we ensure correctness?

One way is to manually run your program after writing it. You write a new feature, open a browser, click around to see that it works, then continue adding more features. This works while your application is small, but at some point your program has too many features to keep track of. You write some new code, but it unexpectedly breaks old features and you might not even know it. This is called a **regression**. At one point your code worked, but you later introduced new code which broke the old functionality.

A better way is to have the computer check our work. We write sofware to automate our lives, so why not write programs to test our code as well? **Automated tests** are small scripts that output whether or not your code works as intended. They verify that our program works now, and will continue to work in the future, without humans having to test it by hand. Once you write a test, you should be able to reuse it for the lifetime of the code it tests, although your tests can change as expections of your application change.

Any large scale and long lasting Rails application should have a comprehensive test suite. A **test suite** is the collection of tests that ensure that your system works. Before marking any task as "complete" (i.e. merging into the `master` branch of your Git repository), it is imperative to run your entire test suite to catch regressions.

If you have written an effective test suite, and the test suite passes, you can be confident that your entire application behaves as expected.

A test suite will be comprised of many different kinds of tests, varying in scope and subject matter. Some tests will be high level, testing an entire feature and walking through your application as if they were a real user. Others may be specific to a single line of code. We'll discuss the varying flavors of tests in detail throughout this book.

Saving Time and Money

At the end of the day, testing is about saving time and money. Automated tests catch bugs sooner, preventing them from ever being deployed. By reducing the manpower necessary to test an entire system, you quickly make up the time it takes to implement a test in the first place.

Automated tests also offer a quicker feedback loop to programmers, as they don't have to walk through every path in their application by hand. A well written test can take milliseconds to run, and with a good development setup you don't even have to leave your editor. Compare that to using a manual approach a hundred times a day and you can save a good chunk of time. This enables developers to implement features faster because they can code confidently without opening the browser.

When applications grow without a solid test suite, teams are often discouraged by frequent bugs quietly sneaking into their code. The common solution is to hire dedicated testers; a Quality Assurance (QA) team. This is an expensive mistake. As your application grows, now you have to scale the number of hands on deck, who will never be as effective as automated tests at catching regressions. QA increases the time to implement features, as developers must communicate back and forth with another human. Compared to a test suite, this is costly.

This is not to say that QA is completely useless, but they should be hired in addition to a good test suite, not as a replacement. While manual testers are not as efficient as computers at finding regressions, they are much better at validating subjective qualities of your software, such as user interfaces.

Confidence

Having a test suite you can trust enables you do things you would otherwise not be able to. It allows you to make large, sweeping changes in your codebase without fearing you will break something. It gives you the confidence to deploy code at 5pm on a Friday. Confidence allows you to move faster.

Living Documentation

Since every test covers a small piece of functionality in your app, they serve as something more than just validations of correctness. Tests are a great form of living documentation. Since comments and dedicated documentation are decoupled from your code, they quickly go stale as you change your application. Tests must be up to date, or they will fail. This makes them the second best source of truth, after the code itself, though they are often easier to read. When I am unsure how a piece of functionality works, I'll look first at the test suite to gain insight into how the program is supposed to behave.

Test Driven Development

Automated tests are likely the best way to achieve confidence in a growing codebase. To amplify that confidence and achieve bigger wins in time savings and code cleanliness, we recommend writing code using a process called **Test Driven Development** (TDD). TDD is a process that uses tests to *drive* the design and development of your application. It begins with a development cycle called **Red, Green, Refactor**.

Red, Green, Refactor

Red, Green, Refactor is a series of steps that lead you through developing a given feature or fixing a bug:

Red

Write a test that covers the functionality you would like to see implemented. You don't have to know what your code looks like at this point, you just have to know what it will do. Run the test. You should see it fail. Most test runners will output red for failure and green for success. While we say "failure" do not take this negatively. It's a good sign! By seeing the test fail first, we know that once we make it pass, our code works.

Green

Read the error message from the failing test, and write as little code as possible to fix the current error message. By only writing enough code to see our test pass, we tend to write less code as a whole. Continue this process until the test passes. This may involve writing intermediary features covering lower level functionality which require their own Red, Green, Refactor cycle.

Do not focus on code quality at this point. Be shameless! We simply want to get our new test passing. This may involve returning literal values from methods, which will force you to write additional tests to cover all cases.

Refactor

Clean up your code, reducing any duplication you may have introduced. This includes your code *as well as your tests*. Treat your test suite with as much respect as you would your live code, as it can quickly become difficult to maintain if not handled with care. You should feel confident enough in the tests you've written that you can make your changes without breaking anything.

TDD Approaches

When solving problems with TDD, you must decide where to start testing your code. Should you start from a high level, testing how the user interacts with the system, then drill down to the nitty gritty? Or, should you begin with a low level design and progress outwards to the final feature? The answer to this depends, and will vary person-to-person and feature-to-feature.

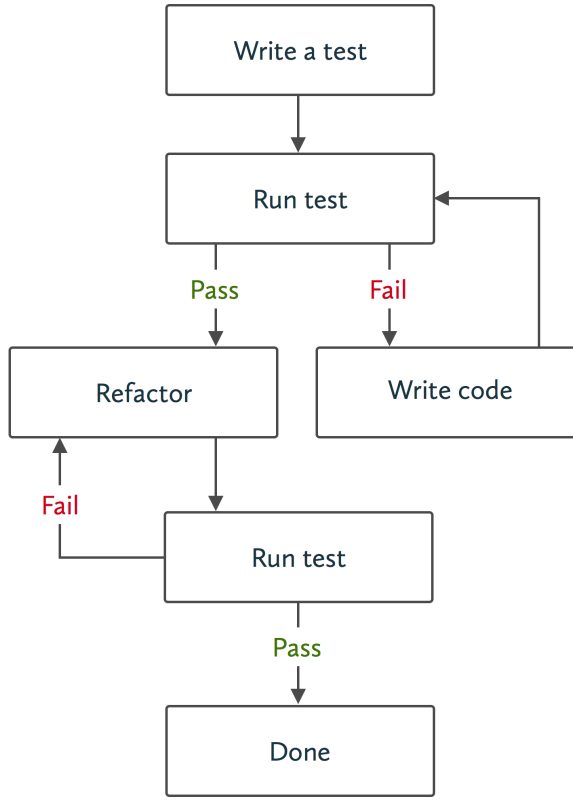

Figure 1.1: TDD Cycle

Outside-In Development

Outside-In Development starts from the highest level of abstraction first. Typically, this will be from the perspective of the user walking through the application in their browser and interacting with elements on the page. We call this an acceptance test, as it ensures that the behavior of the program is acceptable to the end user.

As we develop the feature, we'll gradually need to implement more granular functionality, necessitating intermediate level and lower level tests. These lower level tests may check a single conditional or return value.

By working inwards instead of outwards, you ensure that you never write more code than necessary, because there is a clear end. Once the acceptance test is green, there is no more code to write!

Working outside-in is desirable when you have a good understanding of the problem, and have a rough understanding of how the interface and code will work ahead of time. Because you are starting from a high level, your code will not work until the very end, however your first test will guide your design all the way to completion. You have to trust that your test will bring you there successfully.

Inside-Out Development

Sometimes you don't know what your end solution will look like, so it's better to use an inside-out approach. An inside-out approach helps you build up your code component by component. At each step of the way you will get a larger piece of the puzzle working and your program will be fully functioning at the end of each step. By building smaller parts, one at a time, you can change directions as you get to higher-level components after you build a solid lower-level foundation.

Test Driven vs. Test First

Just because you write your test first, does not mean you are using test driven development. While following the Red, Green, Refactor cycle, it's important to write code only in response to error messages that are provided by test failures. This will ensure that you do not overengineer your solution or implement features that are not tested.

It's also important not to skip the refactor step. This is one of the most important parts of TDD, and ensures that your code is maintainable and easy to change in the future.

Benefits of TDD

Confidence

When it comes down to it, TDD is all about confidence. By writing tests *after* your production code, it's all too easy to forget to test a specific code path. Writing your tests first and only writing code in response to a failing test, you can trust that all of our production code is covered. This confidence gives you power to quickly and easily change your code without fear of it breaking.

Time Savings

Consider automated tests an investment. At first, you will add time by writing tests you would otherwise not be writing. However, most real applications don't stay the same; they grow. An effective test suite will keep your code honest, and save you time debugging over the lifetime of the project. The time savings grow as the project progresses.

TDD can also lead to time savings over traditional testing. Writing your test up front gives you useful error messages to follow to a finished feature. You save time thinking of what to do next, because your test tells you!

Flow

It isn't uncommon for developers to reach a state of "flow" when developing with TDD. Once you write your test, the test failures continuously tell you what to do next, which can almost make programming seem robotic.

Improved Design

That TDD itself improves design is arguable (and many have argued it). In reality, a knowledge of object oriented design principles paired with TDD aids design.

TDD helps you recognize coupling up front. Object oriented design principles, like dependency injection, help you write your code in ways that reduce this coupling, making your code easier to test. It turns out that code that is easy to test happens to align with well structured code. This makes perfect sense, because our tests run against our code and good code is reusable.

A Pragmatic Approach

There's a lot of dogmatism surrounding the exercise of TDD. We believe that TDD is often the best choice for all the reasons above; however, you must always consider the tradeoffs. Sometimes, TDD doesn't make sense or simply isn't worth it. In the end, the most important thing is that you can feel confident that your program works as it should. If you can achieve that confidence in other ways, that's great!

Here are some reasons you might *not* test drive, or even test, your code:

- The feature you are trying to implement is outside your wheelhouse, and you want to code an exploratory version before you can write your test. We call a quick implementation like this a spike. After writing your spike, you may then choose to implement the associated test. If you implement the test after your production code, you should at the very least toggle some code that would make it fail in an expected way. This way, you can be certain it is testing the correct thing. Alternatively, you may want to start from scratch with your new knowledge and implement it as part of a TDD cycle.
- The entire program is small or unlikely to change. If it's small enough to test by hand efficiently, you may elect to forego testing.
- The program will only be used for a short time. If you plan to throw out the program soon, it will be unlikely to change enough to warrant regression testing, and you may decide not to test it.
- You don't care if the program doesn't behave as expected. If the program is unimportant, it may not be worth testing.

Characteristics of an Effective Test Suite

The most effective test suites share the following characteristics.

Fast

The faster your tests are, the more often you can run them. Ideally, you can run your tests after every change you make to your codebase. Tests give you the feedback you need to change your code. The faster they are the faster you can work and the sooner you can deliver a product.

When you run slow tests, you have to wait for them to complete. If they are slow enough, you may even decide to take a coffee break or check Twitter. This quickly becomes a costly exercise. Even worse, you may decide that running tests is such an inconvenience that you stop running your tests altogether.

Complete

Tests cover all public code paths in your application. You should not be able to remove publicly accessible code from your production app without seeing test failures. If you aren't sufficiently covered, you can't make changes and be confident they won't break things. This makes it difficult to maintain your codebase.

Reliable

Tests do not wrongly fail or pass. If your tests fail intermittently or you get false positives you begin to lose confidence in your test suite. Intermittent failures can be difficult to diagnose. We'll discuss some common symptoms later.

Isolated

Tests can run in isolation. They set themselves up, and clean up after themselves. Tests need to set themselves up so that you can run tests individually. When working on a portion of code, you don't want to have to waste time running the entire suite just to see output from a single test. Tests that don't clean up after themselves may leave data or global state which can lead to failures in other tests when run as an entire suite.

Maintainable

It is easy to add new tests and existing tests are easy to change. If it is difficult to add new tests, you will stop writing them and your suite becomes ineffective. You can use the same principles of good object oriented design to keep your codebase maintainable. We'll discuss some of them later in this book.

Expressive

Tests are a powerful form of documentation because they are always kept up to date. Thus, they should be easy enough to read so they can serve as said documentation. During the refactor phase of your TDD cycle, be sure you remove duplication and abstract useful constructs to keep your test code tidy.

Example Application

This book comes with a bundled example application, a Reddit clone called Reddat. If you are unfamiliar with Reddit, it is an online community for posting links and text posts. People can then comment on and upvote those posts. Ours will be a simplified version with no users (anyone can post) and only link posts. Make sure that you sign into GitHub before attempting to view the example application and commit links, or you'll receive a 404 error.

Most of the code samples included in the book come directly from commits in the example application. At any point, you can check out the application locally and check out those commits to explore solutions in progress. For some solutions, the entire change is not included in the chapter for the sake of focus and brevity. However, you can see every change made for a solution in the example commits.

The book is broken into chapters for specific topics in testing, which makes it easier to use as a reference and learn about each part step by step. However, it does make it more challenging to see how a single feature is developed that requires multiple types of tests. To get a sense of how features develop naturally please check out the app's commit history to see the code evolve one feature at a time. Additionally, you'll find more tests to learn from that we won't cover in the book.

Make sure to take a look at the application's README, as it contains a summary of the application and instructions for setting it up.

RSpec

We'll need a testing framework in order to write and run our tests. The framework we choose determines the format we use to write our tests, the commands we use to execute our tests, and the output we see when we run our tests.

Examples of such frameworks include Test::Unit, Minitest, and RSpec. Minitest is the default Rails testing framework, but we use RSpec for the mature test runner and a syntax that encourages human readable tests. RSpec provides a Domain Specific Language (DSL) specifically written for test writing that makes reading and writing tests feel more natural. The gem is called RSpec, because the tests read like specifications. They describe *what* the software does and how the interface should behave. For this reason, we refer to RSpec tests as *specs*.

While this book uses RSpec, the content will be based in theories and practice that you can use with any framework.

Installation

When creating new apps, we run `rails new` with the `-T` flag to avoid creating any Minitest files. If you have an existing Rails app and forgot to pass that flag, you can always remove `/test` manually to avoid having an unused folder in your project.

Use `rspec-rails` to install RSpec in a Rails app, as it configures many of the things you need for Rails testing out of the box. The plain ol' `rspec` gem is used for testing non-Rails programs.

Be sure to add the gem to *both* the `:development` and `:test` groups in your `Gemfile`. It needs to be in `:development` to expose Rails generators and rake tasks at the command line.

```
group :development, :test do
  gem 'rspec-rails', '~> 3.0'
end
```

Bundle install:

```
bundle install
```

Generate RSpec files:

```
rails generate rspec:install
```

This creates the following files:

- `.rspec`

  Configures the default flags that are passed when you run `rspec`. The line `--require spec_helper` is notable, as it will automatically require the spec helper file for every test you run.

- `spec/spec_helper.rb`

  Further customizes how RSpec behaves. Because this is loaded in every test, you can guarantee it will be run when you run a test in isolation. Tests run in isolation should run near instantaneously, so be careful adding any dependencies to this file that won't be needed by every test. If you have configurations that need to be loaded for a subset of your test suite, consider making a separate helper file and load it only in those files.

  At the bottom of this file is a comment block the RSpec maintainers suggest we enable for a better experience. We agree with most of the customizations. I've uncommented them, then commented out a few specific settings to reduce some noise in test output.

- `spec/rails_helper.rb`

  A specialized helper file that loads Rails and its dependencies. Any file that depends on Rails will need to require this file explicitly.

The generated spec helpers come with plenty of comments describing what each configuration does. I encourage you to read those comments to get an idea of how you can customize RSpec to suit your needs. I won't cover them as they tend to change with each RSpec version.

Types of Tests

The Testing Pyramid

The various test types we are about to look at fall along a spectrum. At one end are **unit tests**. These test individual components in isolation, proving that they implement the expected behavior independent of the surrounding system. Because of this, unit tests are usually small and fast.

In the real world, these components don't exist in a vacuum: they have to interact with each other. One component may expect a collaborator to have a particular interface when in fact it has completely different one. Even though all the tests pass, the software as a whole is broken.

This is where **integration tests** come in. These tests exercise the system as a whole rather than its individual components. They typically do so by simulating a user trying to accomplish a task in our software. Instead of being concerned with invoking methods or calling out to collaborators, integration tests are all about clicking and typing as a user.

Although this is quite effective for proving that we have working software, it comes at a cost. Integration tests tend to be much slower and more brittle than their unit counterparts.

Many test types are neither purely unit nor integration tests. Instead, they lie somewhere in between, testing several components together but not the full system.

We like to build our test suite using a combination of these to create a **testing pyramid**. This is a suite that has a few high-level integration tests that cover the

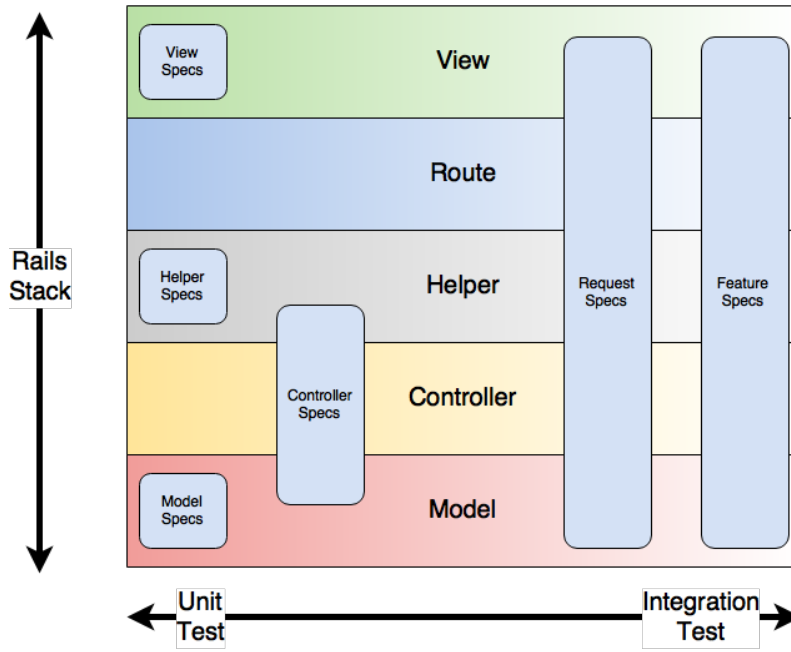

Figure 2.1: Rails Test Types

general functionality of the app, several intermediate-level tests that cover a subsystem in more detail, and many unit tests to cover the nitty-gritty details of each component.

This approach plays to the strength of each type of test while attempting to minimize the downsides they have (such as slow run times).

Feature Specs

Feature specs simulate a user opening your app in a browser and interacting with the page. Since they test that the application works for the end user, they are considered a form of **acceptance tests**, and you may hear them referred to as such. When developing a new feature and practicing outside-in development, this is where we'll typically start.

Submitting a link post

For our first feature, we're going to allow users to create a link post. To do this they'll have to click on a link from the home page, fill in the title and URL of the link, and click "Submit". We'll test that once they do all that, they see the title they entered on the page, and it should link to the URL they provided.

As we're using TDD, we'll start with a test describing the functionality we'd like to see. Let's start with some pseudocode:

```
As a user
When I visit the home page
And I click "Submit a link post"
And I fill in my title and URL
And I click "Submit"
Then I should see the title on the page
And it should link to the given URL
```

One thing to note is that we start our description with *who* the end user is for this test. Our app will only have a single role, so it's safe to use `user`, however many applications will need to differentiate unauthenticated users (often visitors), admins, or domain specific users roles (like a *coach* or *player*).

Capybara

In order for our tests to be able to interact with the browser, we have to install a gem called Capybara. Capybara is an API for interacting with the browser. It provides methods to visit pages, fill in forms, click buttons, and more.

Add it to your Gemfile and `bundle install`:

```
gem "capybara"
```

Require it from your `rails_helper`:

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

The test

With that installed, we're ready to write our spec. Take a look at the completed version. We'll go through it line by line:

```
# spec/features/user_submits_a_link_spec.rb
require "rails_helper"

RSpec.feature "User submits a link" do
  scenario "they see the page for the submitted link" do
    link_title = "This Testing Rails book is awesome!"
    link_url = "http://testingrailsbook.com"

    visit root_path
    click_on "Submit a new link"
    fill_in "link_title", with: link_title
    fill_in "link_url", with: link_url
    click_on "Submit!"

    expect(page).to have_link link_title, href: link_url
  end
end
```

Create a new file at `spec/features/user_submits_a_link_spec.rb`.

We'll first require our `rails_helper.rb`, as this will include our Rails environment and Capybara:

```
require "rails_helper"
```

Next, our feature block:

```
RSpec.feature "User submits a link" do
  ...
end
```

`.feature` is a method that's provided by Capybara. By using it, you have access to all of Capybara's methods for interacting with the page. You may see examples elsewhere calling `.feature` in the global context. This is because old versions of RSpec used monkeypatching to define top level methods on `main` and `Module`. We disabled this functionality by commenting in `config.disable_monkey_patching!`, as this will be the default functionality in future versions of RSpec.

`.feature` takes a string, which you use to describe your feature. We'll usually name this the same thing as we named our file and create a new file for every feature. It gets printed out when we run our specs in `documentation` format.

Inside our feature block, we have a `#scenario` block:

```
scenario "and sees the shortened URL" do
  ...
end
```

This is the container for a single specification. It describes one potential outcome of the user creating a link. Like the feature block, we pass it a string which we'll see in our output when we run our spec. The name for the scenario should be a continuation of the string used for the feature block. When read together, they should construct a sentence-like structure so that they read well when we run our specs.

Now for the spec itself!

We first define a couple variables for our title and URL:

```
link_title = "This Testing Rails book is awesome!"
link_url = "http://testingrailsbook.com"
```

Next, visit the homepage:

```
visit root_path
```

Pretty straightforward. `visit` is a method provided by Capybara which will visit `root_path` as defined by your application. Astute readers will realize that we have not yet defined `root_path`. `root_path` is undefined, because this is a brand new application. We're making this up, as we expect there to be some root route of the application, and we know enough about Rails to know that Rails convention will name it `root_path` once it is defined.

Use Capybara to click a link. This link will bring us to a new page to fill in our form:

```
click_on "Submit a new link"
```

Fill in the form fields:

```
fill_in "link_title", with: link_title
fill_in "link_url", with: link_url
```

If you guessed that `#fill_in` was from Capybara, you'd be right! `#fill_in` finds a method by its name, id, or label and fills it in with the given text. In this case, we're using the ids `link_title` and `link_url`. While we are using ids, note that Capybara does not expect you to add a `#` to the beginning of your id as you would with CSS.

The fields we are looking for don't exist yet, but they will soon. We're using Rails convention over configuration here to guess what the fields are going to be called. We know that Rails gives ids to all fields by joining the model name and the field name. As long as we don't customize those, we can use them to our advantage.

Even if you didn't know that, once we get around to running the test, you'd see it fail because it couldn't find a field with that name, id, or label. At that point, you could open the page in your browser, inspect the element to see what the id is for real, and replace it here.

With the fields filled in, we can submit the form!

```
click_on "Shorten!"
```

And, finally, the assertion:

```
expect(page).to have_link link_title, href: link_url
```

There's a lot going on here, including some "magic", so let's go through each of the components. While this syntax may look strange, remember that RSpec is just Ruby code. This could be rewritten as such:

```
expect(page).to(have_link(link_title, { href: link_url }))
```

`#expect` is an RSpec method that will build an assertion. It takes one value, which we will run an assertion against. In this case it's taking the `page` object, which is a value provided by Capybara that gives access to the currently loaded page. To run the assertion, you call `#to` on the return value of `#expect` and pass it a **matcher**. The matcher is a method that returns truthy or falsy when run through our assertion. The matcher we've passed here is `#have_link`. `#have_link` comes from Capybara, and returns true if it finds a link with the given text on the page. In this case we also pass it a `href` option so it checks against the link's URL.

So now for the magic. `#have_link` doesn't actually exist. RSpec defines matchers automatically for methods that start with `has_` and end with `?`. Capybara defines the method `#has_link?` on the `page` object, so we could think of this whole line as raising an error if `page.has_link?(...)` returns false. RSpec will automatically look for the method `#has_link?` when it sees `#have_link`.

If you've found all this a bit complex, don't worry. How it works is less important than being able to write assertions using the syntax. We'll be writing a lot of tests, so the syntax will soon become familiar.

Take a look at the rspec-expectations gem to see all of the build in matchers.

Running our spec

Now that our spec is written, we have to run it! To run it, find your command line and enter `rspec`. `rspec` will look in our `spec` directory and run all of the files that end in `_spec.rb`. For now, that's a single file and a single test.

You should see something like this:

```
User submits a link
  they see the page for the submitted link (FAILED - 1)

Failures:

  1) User submits a link they see the page for the submitted link
     Failure/Error: visit root_path
     NameError:
       undefined local variable or method `root_path' for
         #<RSpec::ExampleGroups::UserSubmitsALink:0x007f9a2231fe98>
       ./spec/features/user_submits_a_link_spec.rb:8:in
         `block (2 levels) in <top (required)>'

Finished in 0.00183 seconds (files took 2.53 seconds to load)
1 example, 1 failure

Failed examples:

rspec ./spec/features/user_submits_a_link_spec.rb:4 # User submits a link they
  see the page for the submitted link

Randomized with seed 5573
```

Let's go through this bit by bit:

```
User submits a link
  they see the page for the submitted link (FAILED - 1)
```

This is the summary of all the tests we ran. It uses the names provided in our `.feature` and `#scenario` block descriptions. Note that here we read these names together, which is why we wrote them to read nicely together. We see that the scenario `they see the page for the submitted link` failed.

The format we see here is called `documentation` and is due to our configuration in our `spec_helper.rb`. When we run a single spec file, it gives the output it an expressive format. When we run multiple spec files, this format can become cumbersome with all the output, so it uses a more concise `dot` syntax. We'll see that soon.

```
Failures:

  1) User submits a link they see the page for the submitted link
     Failure/Error: visit root_path
     NameError:
       undefined local variable or method `root_path' for
         #<RSpec::ExampleGroups::UserSubmitsALink:0x007f9a2231fe98>
       ./spec/features/user_submits_a_link_spec.rb:8:in
         `block (2 levels) in <top (required)>'
```

This section outlines all of the failures. You will see one failure for each spec that failed. It outputs the error message, the line that failed, and the backtrace. We'll look at this in more detail in a second.

```
Finished in 0.00183 seconds (files took 2.53 seconds to load)
1 example, 1 failure
```

Next is a summary of the tests that were run, giving you the total time to run and the number of tests that were run and that failed.

```
Failed examples:

rspec ./spec/features/user_submits_a_link_spec.rb:4 # User submits a link they
see the page for the submitted link
```

This section outputs the command to run each of the failing specs, for easy copy and pasting. If you initially run your entire suite with `rspec` and want to focus in on a single failing test, you can copy this line and enter it into your terminal to run just that spec. `rspec` takes one or multiple files and will even parse line numbers as you see above by passing the filename with the line number at the end (`rspec ./path/to/file:5` if you wanted to run the spec on line `5`).

```
Randomized with seed 5573
```

Finally, we see the seed we ran our specs with. We run our specs in a random order to help diagnose specs that may not clean up after themselves properly. We'll discuss this in more detail in our section on intermittent failures.

Passing our test

Now that we know how to read the RSpec output, let's pass our test. To do this, we'll read the error messages one at a time and write only enough code to make the current error message pass.

The first error we saw looked like this:

```
Failure/Error: visit root_path
NameError:
  undefined local variable or method `root_path' for
    #<RSpec::ExampleGroups::UserSubmitsALink:0x007f9a2231fe98>
  ./spec/features/user_submits_a_link_spec.rb:8:in
    `block (2 levels) in <top (required)>'
```

It looks like `root_path` is undefined. This helper method comes from Rails when you define the route in `config/routes.rb`. We want our homepage to show all of the links that have been submitted, so it will point to the `index` action of our `LinksController`:

```
root to: "links#index"
```

This is the smallest amount of code we can write to fix that error message. Run the test again:

```
Failure/Error: visit root_path
ActionController::RoutingError:
  uninitialized constant LinksController
```

Okay, we need to define our `LinksController`. In `app/controllers/links_controller.rb`:

```
class LinksController
end
```

Define the controller class. We get this failure:

```
Failure/Error: visit root_path
NoMethodError:
  undefined method `action' for LinksController:Class
```

Hmm, so this one's a bit more cryptic. It's saying that `action` is undefined for our new `LinksController` class. This one requires a bit of Rails knowledge to debug. If you are familiar with Rails, you know that `action` is the word we use to refer to a routable method, specific to controllers. So, what makes a controller different from other classes? Well, it needs to inherit from `ApplicationController`.

```
class LinksController < ApplicationController
```

Run the test again:

```
Failure/Error: visit root_path
AbstractController::ActionNotFound:
  The action 'index' could not be found for LinksController
```

Okay, let's define that method in our controller (remember that `action` is Rails lingo for a method in a controller that can be routed to):

```
def index
end
```

```
Failure/Error: visit root_path
ActionView::MissingTemplate:
  Missing template links/index, application/index with {:locale=>[:en],
    :formats=>[:html], :variants=>[], :handlers=>[:erb, :builder, :raw, :ruby,
    :coffee]}. Searched in:
      * "/Users/jsteiner/code/thoughtbot/testing-behind/app/views"
```

We're missing our template! It tells us all the places it looked and the formats it looked for. In this case it's looking for an HTML template at `links/index`.

We can create an empty file there for now:

```
mkdir app/views/links
touch app/views/links/index.html.erb
```

Rerun the test:

```
Failure/Error: click_on "Submit a new link"
Capybara::ElementNotFound:
  Unable to find link or button "Submit a new link"
```

`app/views/links/index.html.erb` needs a link that reads "Submit a new link". We know it's going to go to our new link page:

```
<%= link_to "Submit a new link", new_link_path %>
```

The new failure:

```
Failure/Error: visit root_path
ActionView::Template::Error:
  undefined local variable or method `new_link_path' for
    #<#<Class:0x007ff23228ee58>:0x007ff232226088>
```

Now we're missing a route for new new link page. Define it in `config/routes.rb`:

```
resources :links, only: [:new]
```

Note here that we limit which routes are created with `only`. This will prevent us from having routes that we don't yet support.

Rerunning the test we get a familiar error.

```
Failure/Error: click_on "Submit a new link"
AbstractController::ActionNotFound:
  The action 'new' could not be found for LinksController
```

At this point, I hope you understand the process we use to develop our features. You may now understand why having fast tests is important, as you can see we run them a lot!

Now, do we really run them after every single small code change? Sometimes. Not always. As you become more experienced with TDD, you'll find that you can predict the output your tests will give as you develop a feature. Once you can do that, you can skip some test runs while still only writing code to pass the test that would have appeared. This allows you to practice TDD while saving some time. However, it is imperative that you only write code in response to the test that would have failed. If you can't accurately predict what failure message you'll see, you should run the tests.

I'll leave the implementation of the rest of this feature as an exercise for the reader. Take a peak at my commit if you get stuck.

Submitting an invalid link

All links should have a title and URL, so we should prevent users from submitting invalid links. Since this is part of the "User submits a link" feature, we can add it to the same feature block under a different scenario. A basic feature spec might look like this:

```
# spec/features/user_submits_a_link_spec.rb
context "the form is invalid" do
  scenario "they see a useful error message" do
    link_title = "This Testing Rails book is awesome!"

    visit root_path
    click_on "Submit a new link"
    fill_in "link_title", with: link_title
    click_on "Submit!"

    expect(page).to have_content "Url can't be blank"
  end
end
```

This test intentionally leaves the URL blank, and expects to see an error message on the page for the missing URL. While we could test every possible path (without a title, without a URL, without both), we really only need to test one at an integration level. This will assure us that an error message renders if the link is invalid. To ensure that each of our fields are valid, we instead test this at the model layer. You can see how I tested this in the respective commit, but we won't cover model specs until the next chapter.

There are a couple new methods in this test. The first is `#context`. As you might guess, it allows you to provide additional context to wrap one or more scenarios. In fact, you can even nest additional context blocks, however we recommend against that. Specs are much easier to read with minimal nesting. If you need to nest scenarios more than a couple levels deep, you might consider pulling out a new feature file.

The other new method is `#have_content`. Like `#have_link`, this method comes from Capybara, and is actually `#has_content?`. `#has_content?` will look on the page for the given text, ignoring any HTML tags.

Passing the test

As always, I'll run the test now and follow the error messages to a solution. I'll leave this up to the reader, but feel free to check out the commit to see what I did.

Four Phase Test

You'll note that in each of our tests so far, we've used some strategic spacing. This spacing is meant to make the tests easier to read by sectioning it into multiple phases. The pattern here is modeled after the Four Phase Test, which takes the form:

```
test do
  setup
  exercise
  verify
  teardown
end
```

Setup

During setup, we create any objects that your test depends on.

Exercise

During exercise, we execute the functionality we are testing.

Verify

During verify, we check our expectations against the result of the exercise phase.

Teardown

During teardown, we clean-up after ourselves. This may involve reseting the database to it's pre-test state or resetting any modified global state. This is usually handled by our test framework.

Four phase testing is more prominently used with model and unit tests, however it is still useful for our acceptance tests. This is especially true for simple tests like the two we've demonstrated, however some acceptance tests may be large enough to warrant even more grouping. It's best to use your discretion and group things into logical sections to make code easier to read.

Viewing the homepage

Now that we can create links, we should test that we actually see them on the homepage. Again, we'll start with some pseudocode:

```
As a user
Given a link has already been submitted
When I visit the home page
Then I should see the link's title on the page
And it should link to the correct URL
```

This test is a little different than our last. This time we have a "given". Instead of creating a link ourselves, we're going to assume one already exists. The reason behind this is simple. Walking through our application with Capybara is slow. We shouldn't do it any more than we have to. We've already tested that we can submit a link, so we don't need to test it again. Instead, we can create records directly in the database.

We could go about creating records the way you'd expect:

```
link = Link.create(title: "Testing Rails", url: "http://testingrailsbook.com")
```

This *would* work, but it has some serious downfalls when using it to test software. Imagine we have a large application, with hundreds of tests, each one having created a `Link` the manual way. If we were to add a required field to links, we would have to go through all of our tests and add the required field for *all* of these tests to get them to pass again. There are two widely used fixes for this painpoint. The first one is called fixtures.

Fixtures

Fixtures allow you to define sample data in YAML files that you can load and reuse through your tests. It might look something like this:

```
# fixtures/links.yml
testing_rails:
  title: Testing Rails
  url: http://testingrailsbook.com

# In your test
link = links(:testing_rails)
```

Now that we've extracted the definition of our *Testing Rails* link, if our model adds new required fields we only have to update our fixtures file. This is a big step up, but we still see some problems with this solution.

For one, fixtures are a form of **Mystery Guest**. You have a Mystery Guest when data used by your test is defined outside the test, thus obscuring the cause and effect between that data and what is being verified. This makes tests harder to reason about, because you have to hunt down another file to be able to understand the entirety of what is happening.

As applications grow, you'll typically need variations on each of your models for different situations. For example, you may have a fixture for every user role in your application, then even more users for different roles depending on whether or not the user is a member of a specific organization. All these possible states a user can be in grow the number of fixtures you will have to recall.

FactoryGirl

We've found factories to be a better alternative to fixtures. Rather than defining hardcoded data, factories define generators of sorts, with predefined logic where necessary. You can override this logic directly when instantiating the factories in your tests. They look something like this:

```
# spec/factories.rb
FactoryGirl.define do
  factory :link do
    title "Testing Rails"
    url "http://testingrailsbook.com"
  end
end


# In your test
link = create(:link)

# Or override the title
link = create(:link, title: "TDD isn't Dead!")
```

Factories put the important logic right in your test. They make it easy to see what is happening at a glance and are more flexible to different scenarios you may want to set up. While factories can be slower than fixtures, we think the benefits in flexibility and readability outweigh the costs.

Installing FactoryGirl

To install FactoryGirl, add `factory_girl_rails` to your `Gemfile`:

```
group :development, :test do
  ...
  gem "factory_girl_rails"
  ...
end
```

We'll also be using Database Cleaner:

```
group :test do
  ...
  gem "database_cleaner"
  ...
end
```

Install the new gems and create a new file `spec/support/factory_girl.rb`:

```
# spec/support/factory_girl.rb
RSpec.configure do |config|
  config.include FactoryGirl::Syntax::Methods

  config.before(:suite) do
    begin
      DatabaseCleaner.start
      FactoryGirl.lint
    ensure
      DatabaseCleaner.clean
    end
  end
end
```

This file will lint your factories before the test suite is run. That is, it will ensure that all the factories you define are valid. While not necessary, this is a worthwhile check, especially while you are learning. It's a quick way to rest easy that your factories work. Since `FactoryGirl.lint` may end up persisting some records to the database, we use Database Cleaner to restore the state of the database after we've linted our factories. We'll cover Database Cleaner in depth later.

Now, this file won't require itself! In your `rails_helper` you'll find some commented out code that requires all of the files in `spec/support`. Let's comment that in so our FactoryGirl config gets loaded:

```
# Uncomment me!
Dir[Rails.root.join("spec/support/**/*.rb")].each { |f| require f }
```

Last, we need to create our factories file. Create a new file at `spec/factories.rb`:

```
FactoryGirl.define do
end
```

This is where we'll define our factory in the next section.

The test

With FactoryGirl set up, we can write our test. We start with a new file at `spec/features/user_views_homepage_spec.rb`.

```
require "rails_helper"

RSpec.feature "User views homepage" do
  scenario "they see existing links" do
  end
end
```

We require our `rails_helper` and create the standard feature and scenario blocks.

```
link = create(:link)
```

To setup our test, we create a link using FactoryGirl's `.create` method, which instantiates a new `Link` object with our (currently non-existant) factory definition and persists it to the database.

`.create` is loaded into the global context in `spec/support/factory_girl.rb`:

```
config.include FactoryGirl::Syntax::Methods
```

While we'll be calling `.create` in the global context to keep our code cleaner , you may see people calling it more explicitly: `FactoryGirl.create`. This is simply a matter of preference, and both are acceptable.

Now, we'll need to add a factory definition for our `Link` class in `spec/factories.rb`:

```
# spec/factories.rb
factory :link do
  title "Testing Rails"
  url "http://testingrailsbook.com"
end
```

We define a default title and URL to be created for all links created with FactoryGirl. We only define defaults for fields that we validate presence of. If you add more than that, your factories can become unmanageable as all of your tests become coupled to data defined in your factories that isn't a default. Not following this advice is a common mistake in Rails codebases and leads to major headaches.

The specific title and URL is unimportant, so we don't override the factories' defaults. This allows us to focus on what is important and makes the test easier to read.

```
visit root_path

expect(page).to have_link link.title, href: link.url
```

Nothing novel here. Visit the homepage and assert that we see the title linking to the URL.

Passing the test

This is left as an exercise for the reader. Feel free to check out the associated commit to see what I did.

Voting on links

One of the most important parts of Reddit is being able to vote for posts. Let's implement a basic version of this functionality in our app, where you can upvote and downvote links.

Here's a basic test for upvoting links:

```
# spec/features/user_upvotes_a_link_spec.rb
RSpec.feature "User upvotes a link" do
  scenario "they see an increased score" do
    link = create(:link)

    visit root_path

    within "#link_#{link.id}" do
      click_on "Upvote"
    end

    expect(page).to have_css "#link_#{link.id} [data-role=score]", text: "1"
  end
end
```

There are a couple new things in this test. First is the `within` block. `within` takes a selector and looks for a matching element on the page. It then limits the scope of everything within the block to elements inside the specified element. In this case, our page has a potential to have multiple links or other instances of the word "Upvote". We scope our finder to only look for that text within the list element for our link. We use the CSS id `#link_#{link.id}` which is given by `content_tag_for`.

The second new method is `has_css`, which asserts that a given selector is on the page. With the `text` option, it ensures that the provided text is found within the given selector. The selector I use includes a data attribute: `[data-role=score]`. We'll frequently use `data-role`s to decouple our test logic from our presentation logic. This way, we can change class names and tags without breaking our tests!

Model Specs

As you can probably guess, model specs are specs for testing your Rails models. If you've written unit tests before, they may seem similar, although many model specs will interact with the database due to the model's dependency on ActiveRecord, so they are not truly unit tests.

Instance Methods

In the last chapter, we added functionality for users to vote on links with some instance methods on our `Link` class to help with this.

Link#upvote

The first method is `#upvote`, which increments the `upvotes` count on the link by 1. A simple way to test this behavior is to instantiate an object with a known upvote count, call our `#upvote` method, and then verify that the new upvote count is what we expect. A test for that might look like this:

```
# spec/models/link_spec.rb
RSpec.describe Link, "#upvote" do
  it "increments upvotes" do
    link = build(:link, upvotes: 1)

    link.upvote

    expect(link.upvotes).to eq 2
  end
end
```

`.describe` comes from RSpec and creates a group for whatever functionality you are describing. It takes a subject, in our case the `Link` class, and the behavior as a string. Typically, we'll use the name of our method, in this case `#upvote`. We prefix instance methods with a `#` and class methods with a `.`.

```
link = build(:link, upvotes: 1)
```

`.build` is another FactoryGirl method. It's similar to `.create`, in that it instantiates an object based on our factory definition, however `.build` does not save the object. Whenever possible, we're going to favor `.build` over `.create`, as persisting to the database is one of the slowest operations in our tests. In this case, we don't care that the record was saved before we increment it so we use `.build`.

Our *verify* step is slightly different than we've seen in our feature specs. This time, we aren't asserting against the `page` (we don't even have access to the page, since this isn't a Capybara test). Instead, we're asserting against our system under test: the link. We're using a built in RSpec matcher `eq` to confirm that the *expected* value, `2`, matches the *actual* value of `link.upvotes`.

With the test written, we can implement the method as such:

```
# app/models/link.rb
def upvote
  increment!(:upvotes)
end
```

Link#score

Our score method should return the difference of the number of upvotes and downvotes. To test this, we can instantiate a link with a known upvote count and downvote count, then compare the expected and actual scores.

```
# spec/models/link_spec.rb
RSpec.describe Link, "#score" do
  it "returns the upvotes minus the downvotes" do
    link = Link.new(upvotes: 2, downvotes: 1)

    expect(link.score).to eq 1
  end
end
```

In this test, you'll notice that we forego FactoryGirl and use plain ol' ActiveRecord to instantiate our object. `#score` depends on `#upvotes` and `#downvotes`, which we can set without saving our object. Since we never have to save our object, we don't need FactoryGirl to set up a valid record.

With a failing test, we can write our implementation:

```
# app/models/link.rb
def score
  upvotes - downvotes
end
```

Class Methods

Testing class methods works similarly to testing instance methods. I added some code to sort the links from highest to lowest score. To keep our business logic in our models, I decided to implement a `.hottest_first` method to keep that logic out of the controller.

We order our model specs as close as possible to how we order our model's methods. Thus, I added the spec for our new class method under the validations tests and above the instance method tests.

```
# spec/models/link_spec.rb
RSpec.describe Link, ".hottest_first" do
  it "returns the links: hottest to coldest" do
    coldest_link = create(:link, upvotes: 3, downvotes: 3)
    hottest_link = create(:link, upvotes: 5, downvotes: 1)
    lukewarm_link = create(:link, upvotes: 2, downvotes: 1)

    expect(Link.hottest_first).to eq [hottest_link, lukewarm_link, coldest_link]
  end
end
```

This is a fairly common pattern, as many of our ActiveRecord model class methods are for sorting or filtering. The interesting thing to note here is that I intentionally scramble the order of the created links. I've also chosen numbers for the upvotes and downvotes to ensure that the test will fail if we incidentally are testing something other than what we intend. For example, if we accidentally implemented our method to sort by upvotes, the test would still fail.

Validations

We use a library called shoulda-matchers to test validations. `shoulda-matchers` provides matchers for writing single line tests for common Rails functionality. Testing validations in your model is important, as it is unlikely validations will be tested anywhere else in your test suite.

To use `shoulda-matchers`, add the gem to your Gemfile's `:test` group:

```
gem "shoulda-matchers"
```

After bundle installing, you can use the built in matchers (see more online) like so:

```
RSpec.describe Link, "validations" do
  it { is_expected.to validate_presence_of(:title) }
  it { is_expected.to validate_presence_of(:url) }
  it { is_expected.to validate_uniqueness_of(:url) }
end
```

`is_expected` is an RSpec method that makes it easier to write one line tests. The `it` these tests refer to is the test's `subject`, a method provided by RSpec when you pass a class as the first argument to `describe`. RSpec takes the subject you pass into `describe`, and instantiates a new object. In this case, `subject` returns `Link.new`. `is_expected` is a convenience syntax for `expect(subject)`. It reads a bit nicer when you read the whole line with the `it`. The following lines are roughly equivalent:

```
RSpec.describe Link, "validations" do
  it { expect(Link.new).to validate_presence_of(:title) }
  it { expect(subject).to validate_presence_of(:url) }
  it { is_expected.to validate_uniqueness_of(:url) }
end
```

Associations

While `shoulda-matchers` provides methods for testing associations, we've found that adding additional tests for associations is rarely worth it, as associations will be tested at an integration level. Since we haven't found them useful for catching regressions or for helping us drive our code, we have stopped using them.

Request Specs

Request specs are integration tests that allow you to send a request and make assertions on its response. As end-to-end tests, they go through the entire Rails

stack from route to response. Unlike feature specs, request specs do not work with Capybara. Instead of interacting with the page like you would with Capybara, you can only make basic assertions against the response, such as testing the status code, redirection, or that text appeared in the response body.

Request specs should be used to test API design, as you want to be confident that the URLs in your API will not change. However, request specs can be used for any request, not just APIs.

In this chapter, we'll add a basic API to our app to show how you might test one with request specs.

Viewing links

The first endpoint we'll create is for an index of all existing links, from hottest to coldest. We'll namespace everything under `/api/v1`.

```
# spec/requests/api/v1/links_spec.rb
require "rails_helper"

RSpec.describe "GET /api/v1/links" do
  it "returns a list of all links, hottest first" do
    coldest_link = create(:link)
    hottest_link = create(:link, upvotes: 2)

    get "/api/v1/links"

    expect(json_body["links"].count).to eq(2)

    hottest_link_json = json_body["links"][0]
    expect(hottest_link_json).to eq({
      "id" => hottest_link.id,
      "title" => hottest_link.title,
      "url" => hottest_link.url,
      "upvotes" => hottest_link.upvotes,
      "downvotes" => hottest_link.downvotes,
    })
```

```
  end
end
```

We name our request spec files after the paths they test. In this case requests to `/api/v1/links` will be tested in `spec/api/v1/links_spec.rb`.

After setting up our data, we make a `GET` request with the built-in `get` method. We then assert on the number of records returned in the JSON payload. Since all of our requests will be JSON, and we are likely to be parsing each of them, I've extracted a method `json_body` that parses the `response` object that is provided by `rack-test`.

```
# spec/support/api_helpers.rb
module ApiHelpers
  def json_body
    JSON.parse(response.body)
  end
end

RSpec.configure do |config|
  config.include ApiHelpers, type: :request
end
```

I pulled the method out to its own file in `spec/support`, and include it automatically in all request specs.

We could have tested the entire body of the response, but that would have been cumbersome to write. Asserting upon the length of the response and the structure of the first JSON object should be enough to have reasonable confidence that this is working properly.

Creating links

Next, we'll test creating a new link via our API:

```
# spec/requests/api/v1/links_spec.rb
RSpec.describe "POST /api/v1/links" do
  it "creates the link" do
```

```
    link_params = attributes_for(:link)

    post "/api/v1/links", link: link_params

    expect(response.status).to eq 201
    expect(Link.last.title).to eq link_params[:title]
  end

  context "when there are invalid attributes" do
    it "returns a 422, with errors" do
      link_params = attributes_for(:link, :invalid)

      post "/api/v1/links", link: link_params

      expect(response.status).to eq 422
      expect(json_body.fetch("errors")).not_to be_empty
    end
  end
end
```

`attributes_for` is another FactoryGirl method, which gives you a hash of the attributes defined in your factory. In this case, it would return:

```
{ title: "Testing Rails", url: "http://testingrailsbook.com" }
```

This time, we `POST` to `/api/v1/links`. `post` takes a second hash argument for the data to be sent to the server. We assert on the response status. `201` indicates that the request succeeded in creating a new record. We then check that the last `Link` has the title we expect to ensure it is creating a record using the data we submitted.

In the second test, we introduce a new FactoryGirl concept called traits. Traits are specialized versions of factories. To declare them, you nest them under a factory definition. This will give them all the attributes of the parent factory, as well as any of the modifications specified in the trait. With the new trait, our `Link` factory looks like this:

```
# spec/factories.rb
factory :link do
```

```
  title "Testing Rails"
  url "http://testingrailsbook.com"

  trait :invalid do
    title nil
  end
end
```

The `:invalid` trait nulls out the `title` field so we can easily create invalid records in a reusable manner.

View Specs

View specs allow you to test the logic in your views. While this logic should be minimal, there are certainly times where you'll want to pull out the handy view spec to test some critical functionality. A common antipattern in test suites is testing too much in feature specs, which tend to be slow. This is especially a problem when you have multiple tests covering similar functionality, with minor variations.

In this section, we'll allow image links to be rendered inline. The main functionality of displaying link posts was tested previously in a feature spec. Aside from the already tested logic for creating a link, rendering a link post as an inline image is mostly view logic. Instead of duplicating that functionality in another feature spec, we'll write a view spec, which should cover our use case and minimize test suite runtime.

Rendering Images Inline

In order to keep our link rendering logic DRY, I moved all of it into `app/views/links/_link.html.erb`. This way, we can reuse that partial anywhere we want to display a link, and it can correctly render with or without the image tag.

The associated spec looks like this:

```
# spec/views/links/_link.html.erb_spec.rb
require "rails_helper"
```

```
RSpec.describe "links/_link.html.erb" do
  context "if the url is an image" do
    it "renders the image inline" do
      link = build(:link, url: "http://example.com/image.jpg")

      render partial: "links/link.html.erb", locals: { link: link }

      expect(rendered).to have_selector "img[src='#{link.url}']"
    end
  end
end
```

In this spec, we build a link with an image URL, then `render` our partial with our link as a local variable. We then make a simple assertion that the image appears in the rendered html.

When I initially implemented this partial, I had forgotten to also render the image on the link's show page. Since some functionality I expected to see wasn't implemented, I wrote a test to cover that case as well.

```
# spec/views/links/show.html.erb_spec.rb
require "rails_helper"

RSpec.describe "links/show.html.erb" do
  context "if the url is an image" do
    it "renders the image inline" do
      link = build(:link, url: "http://example.com/image.jpg")
      assign(:link, link)

      render

      expect(rendered).to have_selector "img[src='#{link.url}']"
    end
  end
end
```

This test is similar to the previous one, but this time we are rendering a view as opposed to a partial view. First, instead of a local variable we need to assign an

instance variable. `assign(:link, link)` will assign the value of the variable `link` to the instance variable `@link` in our rendered view.

Instead of specifying the view to render, this time we let RSpec work its "magic". RSpec infers the view it should render based on the name of the file in the `describe` block.

Controller Specs

Controller specs exist in a weird space between other test types. They have some overlap with many of the other test types discussed so far so their use can be controversial.

In terms of scope they aren't really unit tests because controllers are so tightly coupled to other parts of the Rails infrastructure. On the other hand, they aren't integration tests either because requests don't go through the routes and don't render the view.

As their name implies, controller specs are used to test the logic in a controller. We've previously seen that feature specs can drive the creation of a controller. Given that Rails developers actively try to keep logic out of their controllers and that feature specs do cover controllers, controller tests can often be redundant. A good rule of thumb is that you don't need a controller test until you introduce conditional logic to your controller. In our experience, we tend to write very few controller specs in our applications.

As previously mentioned, feature specs are *slow* (relative to other spec types). They are best used to test flows through an application. If there are multiple ways to error out of a flow early, it can be expensive to write the same feature spec over and over with minor variations.

Time for a controller spec! Or what about a request spec? The two spec types are quite similar and there are many situations where either would do the job. The main difference is that controller specs don't actually render views or hit URLs and go through the routing system.

So if you have logic in a controller and

- the forking logic is part of two distinct and important features, you may want a **feature spec**

- you care about the URL, you may want a **request spec**
- you care about the rendered content, you may want a **request spec** or even a **view spec**
- none of the above apply, you may want a **controller spec** or a **request spec**

One common rule of thumb is to use feature specs for **happy paths** and controller tests for the **sad paths**.

The "happy path" is where everything succeeds (e.g. successfully navigating the app and submitting a link) while the "sad path" is where a failure occurs (e.g. successfully navigating the app but submitting an invalid link). Some flows through the app have multiple points of potential failure so there can be multiple "sad paths" for a given "happy path".

All this being said, let's look at an actual controller spec! In this section, we'll be rewriting the tests for the invalid link case to use a controller spec rather than a feature spec.

Invalid Links

In this test, we want to try and submit an invalid link and expect that it will not succeed but that the form will be re-rendered.

The specs looks like this:

```
# spec/controllers/links_controller_spec.rb
require "rails_helper"

RSpec.describe LinksController, "#create" do
  context "when the link is invalid" do
    it "re-renders the form" do
      post :create, link: attributes_for(:link, :invalid)

      expect(response).to render_template :new
    end
  end
end
```

Just like with the request spec, the `post` method will make a `POST` request. However, unlike the request spec, we are making the request directly to a controller action rather than to a URL.

The first parameter to `post` is the action we want to exercise. In addition, we may pass an optional hash of params. Since we are simulating a form submission, we need a hash of attributes nested under the `link` key. We can generate these attributes by taking advantage of the invalid link factory we created earlier. Finally, the controller is inferred from the `RSpec.describe`.

This will make a `POST` request to the `LinksController#create` action with an invalid link as its payload.

Controller specs expose a `response` object that we can assert against. Although we cannot assert against actual rendered content, we can assert against the name of the template that will be rendered.

It is worth noting here that this spec is *not* equivalent to the feature spec it replaces. The feature test tested that an error message *actually appeared on the page*. The controller test, on the other hand, only tests that the form gets re-rendered.

This is one of those situations where you have to make a judgment call. Is it important to you to test that not only does the application handles the error, but also that an error message shows up on the page? Is it worth trading a slow and somewhat duplicated feature spec for a faster controller test that doesn't test the UI? Would a request spec be a good compromise option? What about a controller spec plus a view spec to test the both sides independently?

All of these options are valid solutions. Based on the context you will pick the one that gives you the best combination of confidence, coverage, and speed.

Helper Specs

Helpers are generally one-off functions that don't really fit anywhere else. They can be particularly easy to test due to their small scope and lack of side-effects.

We will add some formatting to the display of a link's score. While a high score means that a link is popular, a low score can have multiple meanings. Is it new? Is it controversial and thus has a high number of both positive and negative votes? Is it just boring?

To make some of the context more obvious, we will format the score as `5 (+7, -2)` instead of just showing the net score.

Formatting the score

Formatting is not a model-level concern. Instead, we are going to implement it as a helper method. In TDD fashion we start with a test:

```
# spec/helpers/application_helper_spec.rb
require "rails_helper"

RSpec.describe ApplicationHelper, "#formatted_score_for" do
  it "displays the net score along with the raw votes" do
    link = Link.new(upvotes: 7, downvotes: 2)
    formatted_score = helper.formatted_score_for(link)

    expect(formatted_score).to eq "5 (+7, -2)"
  end
end
```

Since we don't need to persist to the database and don't care about validity, we are using `Link.new` here instead of `FactoryGirl`.

Helpers are modules. Because of this, we can't instantiate them to test inside a spec, instead they must be mixed into an object. RSpec helps us out here by providing the `helper` object that automatically mixes in the described helper. All of the methods on the helper can be called on `helper`.

It is worth noting here that this is not a pure unit test since it depends on both the helper *and* the `Link` model. In a later chapter, we will talk about **doubles** and how they can be used to isolate code from its collaborators.

Intermediate Testing

Testing in isolation

In a previous chapter we discussed **unit tests**, tests that exercise a single component of a system in isolation. That's nice in theory, but in the real world most objects depend on **collaborators** which may in turn depend on their own collaborators. You set out to test a single object and end up with a whole sub-system.

Say we want to add the ability to calculate whether or not a link is controversial. We're starting to have lot of score-related functionality so we extract it into its own `Score` object that takes in a `Link` in its constructor. `Score` implements the following: `#upvotes`, `#downvotes`, `#value`, and `#controversial?`.

The spec looks like:

```
# spec/models/score_spec.rb
require "rails_helper"

RSpec.describe Score do
  describe "#upvotes" do
    it "is the upvotes on the link" do
      link = Link.new(upvotes: 10)
      score = Score.new(link)

      expect(score.upvotes).to eq 10
    end
  end
```

```
  describe "#downvotes" do
    it "is the downvotes on the link" do
      link = Link.new(downvotes: 5)
      score = Score.new(link)

      expect(score.downvotes).to eq 5
    end
  end

  describe "#value" do
    it "is the difference between up and down votes" do
      link = Link.new(upvotes: 10, downvotes: 3)
      score = Score.new(link)

      expect(score.value).to eq 7
    end
  end

  describe "#controversial?" do
    it "is true for posts where up/down votes are within 20% of each other" do
      controversial_link = Link.new(upvotes: 10, downvotes: 9)
      score = Score.new(controversial_link)

      expect(score).to be_controversial
    end

    it "is false for posts where up/down votes have > 20% difference" do
      non_controversial_link = Link.new(upvotes: 10, downvotes: 5)
      score = Score.new(non_controversial_link)

      expect(score).not_to be_controversial
    end
  end
end
```

The **system under test** (often abbreviated SUT) is the unit we are trying to test. In this case, the SUT is the instance of `Score` which we've named `score` in each test.

However, `score` can't do it's work alone. It needs help from a **collaborator**. Here, the collaborator (`Link`) is passed in as a parameter to `Score`'s constructor.

You'll notice the tests all follow the same pattern. First, we create an instance of `Link`. Then we use it to build an instance of `Score`. Finally, we test behavior on the score. Our test can now *fail for reasons completely unrelated to the score object*:

- There is no `Link` class defined yet
- `Link`'s constructor expects different arguments
- `Link` does not implement the instance methods `#upvotes` and `#downvotes`

Note that the collaborator doesn't *have* to be an instance of `Link`. Ruby is a **duck-typed** language which means that collaborators just need to implement an expected set of methods rather than be of a given class. In the case of the `Score`'s constructor, any object that implements the `#upvotes`, and `#downvotes` methods could be a collaborator. For example if we introduce comments that could be upvoted/downvoted, `Comment` would be another equally valid collaborator.

Ideally, in a pure unit test we could isolate the SUT from its collaborators so that only the SUT would cause our spec to fail. In fact, we should be able to TDD the SUT even if collaborating components haven't been built yet.

Test doubles

RSpec gives us **test doubles** (sometimes also called **mock objects**) which act as fake collaborators in tests. The name derives from stunt doubles in movies that stand in for the real actor when a difficult stunt needs to be done. Test doubles are constructed with the `double` method. It takes an optional hash of methods it needs to respond to as well as their return values.

Let's try using this in our spec:

```
# spec/models/score_spec.rb
require "rails_helper"

RSpec.describe Score do
  describe "#upvotes" do
    it "is the upvotes on the link" do
```

```
    link = double(upvotes: 10, downvotes: 0)
    score = Score.new(link)

    expect(score.upvotes).to eq 10
  end
end

describe "#downvotes" do
  it "is the downvotes on the link" do
    link = double(upvotes: 0, downvotes: 5)
    score = Score.new(link)

    expect(score.downvotes).to eq 5
  end
end

describe "#value" do
  it "is the difference between up and down votes" do
    link = double(upvotes: 10, downvotes: 3)
    score = Score.new(link)

    expect(score.value).to eq 7
  end
end

describe "#controversial?" do
  it "is true for posts where up/down votes are within 20% of each other" do
    controversial_link = double(upvotes: 10, downvotes: 9)
    score = Score.new(controversial_link)

    expect(score).to be_controversial
  end

  it "is false for posts where up/down votes have > 20% difference" do
    non_controversial_link = double(upvotes: 10, downvotes: 5)
    score = Score.new(non_controversial_link)

    expect(score).not_to be_controversial
```

```
    end
  end
end
```

Here, we've replaced the dependency on `Link` and are constructing a double that responds to the following interface:

- `upvotes`
- `downvotes`

Historical note

Older versions of RSpec had methods named `stub` and `mock` as aliases to `double`. This is particularly confusing because **mocking** and **stubbing** are entirely different concepts (more on them later). To make things worse, RSpec also monkeypatched `Object` with an `Object.stub` method which *did* do real stubbing. You shouldn't run into these unless you are working with a legacy test suite.

Benefits

Taking this approach yields several benefits. Because we aren't using real collaborators, we can TDD a unit of code even if the collaborators haven't been written yet.

Using test doubles gets painful for components that are highly coupled to many collaborators. Let that pain drive you to reduce the coupling in your system. Remember the final step of the TDD cycle is *refactor*.

Test doubles make the interfaces the SUT depends on explicit. Whereas the old spec said that the helper method relied on a `Link`, the new spec says that methods on `Score` depend on an object that must implement `#upvotes`, and `#downvotes`. This improves the unit tests as a source of documentation.

A pragmatic approach

Sometimes you need to test a component that is really tightly coupled with another. When this is framework code it is often better just to back up a bit

and test the two components together. For example models that inherit from `ActiveRecord::Base` are coupled to ActiveRecord's database code. Trying to isolate the model from the database can get really painful and there's nothing you can do about it because you don't own the ActiveRecord code.