RAILS TESTING HANDBOOK

BY SEMAPHORE

Table of contents

Rails Testing Handbook

BDD is one of the things that Semaphore developers practice every day. Rails Testing Handbook will show you how to do it right.

If you'd like to read more, we regularly share our thoughts on engineering, product development, and testing on our Semaphore blog.

This guide was created by Marko Anastasov, Stefan Mijučić, Igor Šarčević, Milica Maksimović, and Dunja Radulov.

Cover and chapter illustrations: Tamara Čubrilo

CHAPTER 1

Introduction

Back in 2008, I was working on a Rails app where one of its features was a multi-step reservation process. Each step had many possible states, some of which were simply static options, and some were branching into a different workflow. We would sometimes write unit tests, but only after finalizing work on a complex method or a class.

Getting anything done was hard. We'd change one thing, then spend about an hour on manual testing in the web browser to check if everything is still working. We'd often be surprised by the fact that we broke a seemingly unrelated use case. Sometimes we'd figure that out on our own, and sometimes we'd ship a new version to production and the client would email us about what had just stopped working. Getting that kind of news felt the worst. It felt like we were wasting a lot of our client's and our own time. We knew we could do better than that.

Then, we discovered Cucumber. There was a part on its website saying that it was built for "behavior-driven development (BDD)" and "acceptance testing", but all we saw was that we could define a test once, and Cucumber would automatically launch a web browser, run the application, and do the work of feature verification instead of us.

That felt like magic. As soon as we started writing Cucumber scenarios for the feature set we've been working on, our development accelerated by an order of magnitude. We could add, remove and change code without worrying that we would break something. When something did break, a failing scenario would let us know, and we'd quickly fix it before deploying a new version to users.

From that point on, we wanted to write runnable test scenarios for every feature that we'd work on. Then, we wanted to have them before we even wrote a single line of implementation code. Afterwards, we wanted tests for every layer of application code below, too. We continued happily working on that project for years to come.

Perhaps you, dear reader, are in a similar situation. Maybe you've heard that there's a connection between writing tests first and good design, and you want to become a better programmer. Or maybe you're familiar with test-driven development in another language, but want to develop the habit specifically for Rails.

This book aims to provide that upgrade. Our discussion and examples will be practical, because our knowledge comes from practicing BDD in developing real products — not just theorizing about it. Over the past decade, our company Rendered Text has evolved from being a small Rails consultancy to helping thousands of organizations deliver better code faster with Semaphore, our CI/CD product. All that time we've been doing BDD, and it has helped us write robust, sustainable code at a steady pace — regardless of whether the app has 2,000 lines of code or 50,000.

As the software development industry matures, practicing BDD is becoming a standard mark of a software craftsman. This text will help you start practicing it. Enjoy!

Marko Anastasov
Rendered Text & *SemaphoreCI.com* **cofounder**

CHAPTER 2

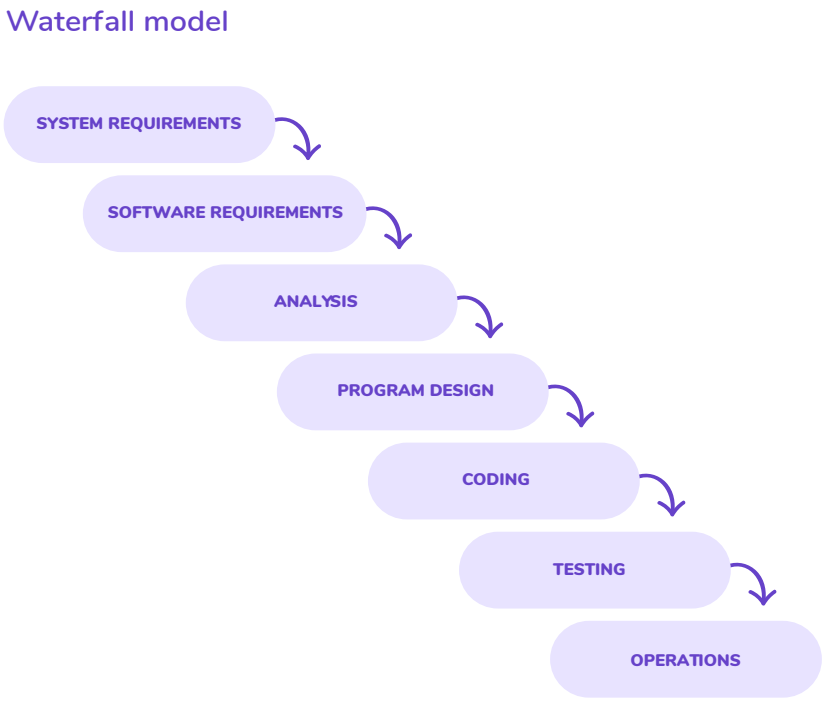
Behavior-driven Development

Introduction

Behavior-driven development (BDD) is about minimizing the feedback loop between business owners and developers. It is a logical step forward in the evolution of software development. In this section, we'll explain the concepts behind BDD and its origin.

Waterfall

If you are a software developer or an engineering manager, you are probably familiar with the Waterfall model, recognizable by the following diagram:



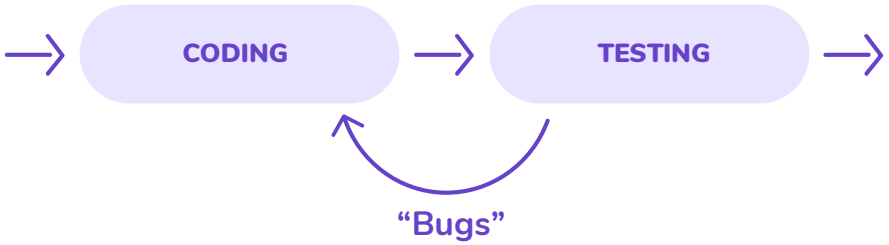
What was later named **"Waterfall"** was first formally described by Winston Royce in his 1970 paper "*Managing the development of large software systems*". Most people assume that this process was presented as the ultimate solution at the time. However, Royce recognized that having a testing phase at the end of the development process is a major problem.

This model is still used to develop software in many companies across the world. Waterfall implies flow, but in practice there are always feedback loops between phases. All major improvements to the model over time have been made by minimizing the feedback loops and making them as predictable as possible. For example, if we write a program, we want to know how long it will take us to find out if it works. On the other hand, if we design a part of the system, we want to find out if it is actually programmable and verifiable, and at what cost.

So, when we look at a feedback loop, we look for methods we can use to minimize it. At first, our goal is to remove obviously wasteful work. Later, we start realizing that we are able to optimize and do things faster and better than we could have ever imagined back when we were doing things the old way.

The First Optimization: Test-first Programming

The first optimization happened by addressing the **Coding and Testing phases.** In the traditional quality assurance-based (QA-based) development model, a programmer writes code and submits it in some way to the QA team. It takes a day, a few days, or weeks to get a report if the code, and the rest of the program work. A lot of times there are bugs, so we need to go back to programming and fix all the issues.



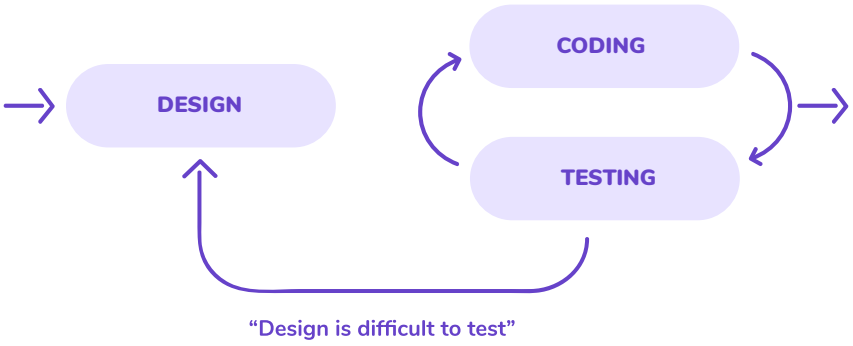
To cut down the feedback loop, we start coding and verifying at the same time. First, we write some code, and then we write some tests for it. Tests produce an excellent side effect, the automated test suite, which we can run at any time to verify every part of the system for which we have written a test. Afterwards, we want to have a test suite that covers the entire system, so that we can work as safely as possible.

The feedback loop of coding followed by testing still takes some time, so we invert it, and we start writing tests before writing a single line of code. The feedback loop becomes very small, and we soon realize that we are writing only the code we need in order to pass the tests that we wrote.

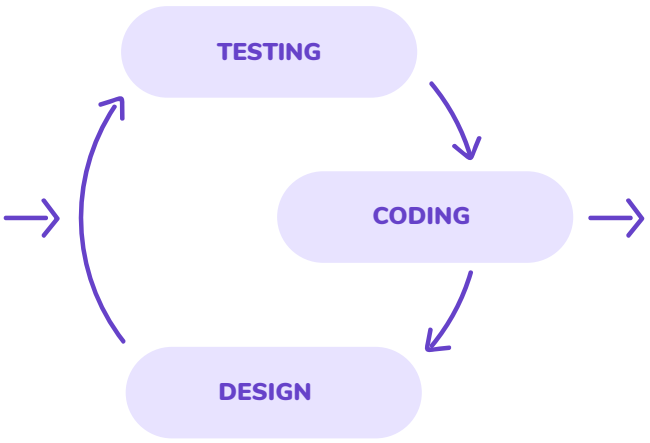
This is called test-first programming. When we work test-first, we use the tests we write to help us "fill in" the implementation correctly. This reduces the number of bugs, increases programmer productivity, and positively affects the velocity of the whole team.

Test-driven Development

Once we have a continuous loop of testing and coding, we're still doing all our program design upfront. We're using test-first programming to make sure that our code works, but there's a feedback loop where we may find out (disturbingly late) that a design is difficult to test, impossible to code, performs badly or just doesn't fit together with the rest of the system.



To minimize this loop, we apply the same technique. We invert it by doing test-first programming before we start designing. In other words, we do the Testing, Coding and Program Design all at the same time. A test influences the code, which in turn influences the design, which influences our next test.



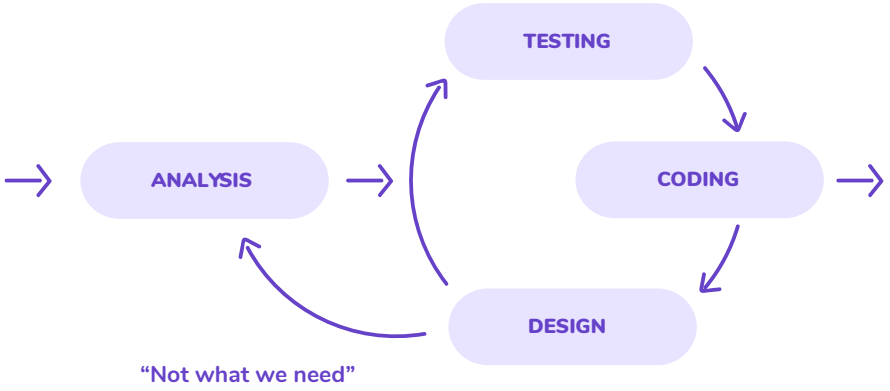
Test-driven development

This is test-driven development (TDD). It drives our design ideas in an organic way, and we implement only the parts of the design that we need, in a way which can easily evolve too. Design now includes a substantial refactoring step, which gives us confidence to under-design instead of over-engineer. We end up having just enough design and appropriate code which meets our current requirements.

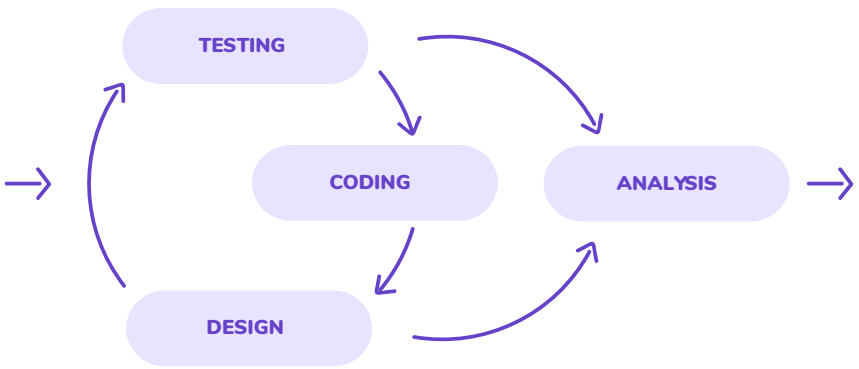
TDD combines test-first programming with design thinking by continuous application of refactoring principles and patterns. The positive side effects are now amplified. We have reduced the number of bugs, and we are not writing any code that doesn't help us with implementation. This further increases a team's productivity by helping it avoid design mistakes which are more costly to fix the later they are detected.

The Second Optimization: Behavior-driven Development

Now that we are designing, coding and testing in one loop, we can look back at the Analysis and try to understand what we need to build. Again, we're interested in optimizing the duration of the loop. In practice, this would involve preparing a list of a dozen or more features and passing it on to developers. They need to complete all of the features before moving forward. This way, we often end up in a situation of implementing features that we don't need. Sometimes we discover new features we didn't expect we would need, or discover something new about the features we knew we needed.

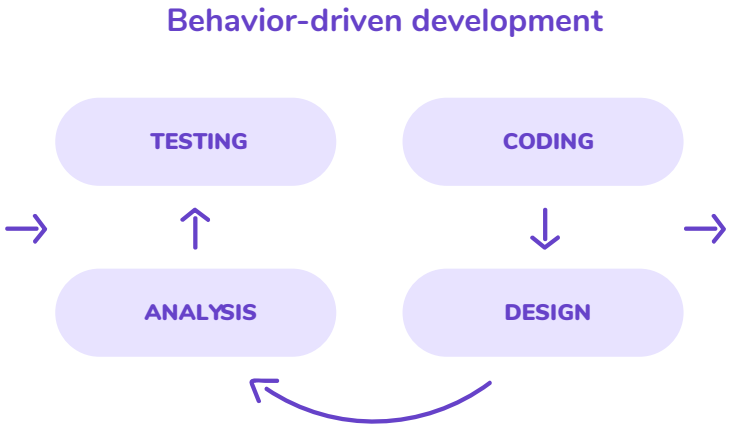


We can apply the same technique and bring Analysis in our loop. Now, we test-drive a feature before we try to implement another. The duration of such cycles for a developer is measured in hours, sometimes even minutes, not days or weeks.



After applying this technique consistently for a while, we notice that we tend to break down all features in the smallest units and consistently deliver them one by one. We improve our understanding of how features affect one another and find ourselves being able to respond to changes faster. We can identify and discard unwanted features quickly and give priority to important features.

By test-driving our analysis, we understand better the behavior of the system we need to build and how to appropriately design and implement it. All that we are doing from day one is producing a test suite, which keeps our entire system constantly verifiable.

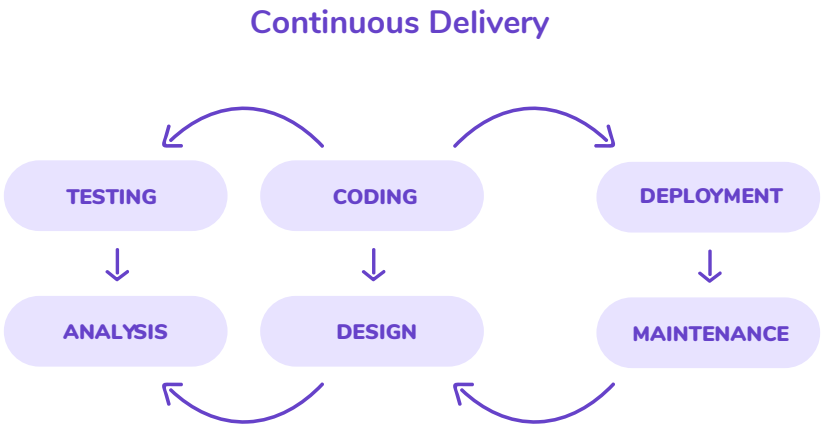


This is called behavior-driven development. It saves time for both the stakeholders (business owners) and the development team. By asking questions early, developers help both themselves and the stakeholders gain a deep understanding of what it is that they are building. Stakeholders get results at a predictable pace and, since the features are worked on in small chunks, estimates can be done more accurately, and new features can be planned and prioritized accordingly.

Going Even Faster

If you take a look at the remaining phases listed in the Waterfall diagram, you may wonder if the same feedback loop minimization can be applied to them as well. The answer is, of course, yes. However, such loops are of a scope that is broader than just design and development, and they involve people working across very different fields. We'll mention them briefly.

Lean Startup would be the closest concept that brings together gathering requirements, feature development and marketing as a way to close the loop on learning what a startup needs to build. Of course, the process goes somewhat differently in enterprises, although they are learning to apply the lean startup principles in many projects as well.



Merging BDD with deployment and operations brings us to the broad concept of continuous delivery. The most important processes are continuous integration (CI) and continuous deployment, which you can easily configure for any project on Semaphore.

Summing up

Behavior-driven development evolved from optimizing various phases in the software development process. By analyzing, testing, coding and designing our system in one short feedback loop, we are able to produce better software by avoiding mistakes and wasteful work.

It is a common misconception that TDD is about testing and that, since it has its origins in TDD, BDD is just another way of approaching software testing. This is not the case, although tests are a nice byproduct. TDD is a holistic approach to software development, derived from one simple idea: wanting to optimize the feedback loops in our work.

None of the steps required to practice BDD are required to make software in general. They also take time to learn and apply effectively. Still, their payoff of a sustainable process that enables us to continuously produce working software is well worth the investment.

CHAPTER 3

Applying BDD to Ruby on Rails Web Applications

Introduction

In this section, we'll focus on applying BDD principles while developing a web application using Ruby on Rails.

Understanding The “Behavior” Point of View

When applying test-driven development (TDD), developers can easily fall into the trap of using unit tests to test what an object or method is, rather than what it does, which is a lot more useful. An example would be writing a test which asserts that a collection of comments is specifically an array, and not one of its unique features, such as being sorted by time. In most cases it shouldn’t matter if we change the implementation of that collection to return a custom enumerable class. More generally, changing the implementation of an object shouldn’t break its test suite as long as what the object does remains the same.

BDD puts focus on behavior — what a thing does — on all levels of development. Initially, the word “behavior” may seem strange. Another way to frame this is to think about descriptions. We can describe every low-level method, object, button or screen to another person — and what we will be describing is exactly what a behavior is. Adopting this approach changes the way we approach writing code.

Understanding The “Behavior” The “Given / When / Then” Communication Pattern Point of View

Most problems in software development are **communication problems**.
For example, business owners fail to describe every use case of a proposed functionality, developers misunderstand the scope of a feature or a product team does not have a protocol to verify if a feature is done. BDD simplifies the language we use to describe scenarios in which software should be used: **Given** some context or state of the world, **When** something happens, **Then** we expect some outcome.

Given, When, Then are simple words we can use to describe a complex feature, code object or a single method equally well. It is a pattern that all members of the team in various roles can understand. These expressions are also built-in in many testing frameworks, such as Cucumber. A clear formulation of the problem and the solution (behavior) that we need to implement helps us write better code.

Overview of BDD Tools for Rails

Ruby on Rails was the first web framework to ship with an integrated testing framework. This acted as a springboard for further advancements of the craft. At the same time, the expressiveness of Ruby and the productivity boost in developing web applications with Rails attracted many experienced and high-profile engineers to the community early on. These are the main reasons why most of the BDD tools and practices gained initial traction and why they have seen significant development in the Rails community.

When you generate a new Rails application with default options, it sets the scene for testing using test/unit, a testing library that comes with Ruby. However, there are tools which make BDD easier to apply. We recommend using RSpec as the main testing library and Cucumber for writing high-level acceptance tests.

RSpec

RSpec is a popular BDD testing library for Ruby. Tests written using RSpec — called specs — are executable examples of expected behavior of a piece of code in a specified context. This is much easier to understand by reading the following code:

```
describe ShoppingCart do
  context “when first created” do
    it “is empty” do
      shopping_cart = ShoppingCart.new
      expect(shopping_cart).to be_empty
    end
  end
end
```

Well-written specs are easy to read, and as a result, understand. Try reading the code snippet above out loud. We are describing a shopping cart, saying that, given a blank context, when we create a new shopping cart, we `expect(shopping_cart).to be_empty`.

Running this spec produces output which resembles a specification:

```
ShoppingCart
  when first created
    is empty
```

We could use RSpec to specify an entire system, however we can also use a tool which helps us write and communicate using more appropriate (broad) terms.

Cucumber

As we explained in the first chapter of this guide, we want to test-drive the analysis phase of every new feature. To do that, we need customer acceptance tests to drive the development of the code which will actually implement the feature. If you are a developer working in a sufficiently complex organization, you may want to have somebody else (a customer or product manager) write acceptance tests for you. In most cases, the developer writes them. This is a good practice, because it helps us understand better what it is that we need to build. Cucumber provides the language and format to do that.

Cucumber reads plain text descriptions of application features, organized in scenarios. Each step in the scenario is implemented using concrete code, and it automates interaction with your application from the user's standpoint. For example:

```
Feature: Reading articles

Scenario: Commenting on an article
  Given I am signed in
  And I am reading an article with “2” comments
  When I reply to the last comment
  Then the article should have “3” comments
  And I should be subscribed to follow-up comments
```

If this were a web application, the scenario above could automatically boot a test instance of the application, open it in a web browser, perform steps as any user would do, and then check if certain expectations have been met.

The BDD Cycle in Rails

In practice, BDD implies an **outside-in** approach. We start with an acceptance test, then write code in the views, and work our way down to the models. This approach helps us discover any new objects or variables we may need to effectively implement our feature early on, and make the right design decisions based on this.

THE BDD CYCLE IN RAILS CONSISTS OF THE FOLLOWING STEPS:

1. **Start with a new Cucumber scenario.** Before you write it, make sure to analyze and understand the problem. At this point you need to know how the user interface allows a user to do a job. Do not worry about the implementation of scenario steps.

2. **Run the scenario and watch it fail.** This will tell you which steps are failing, or pending implementation. At first, most of your steps will be pending (undefined).

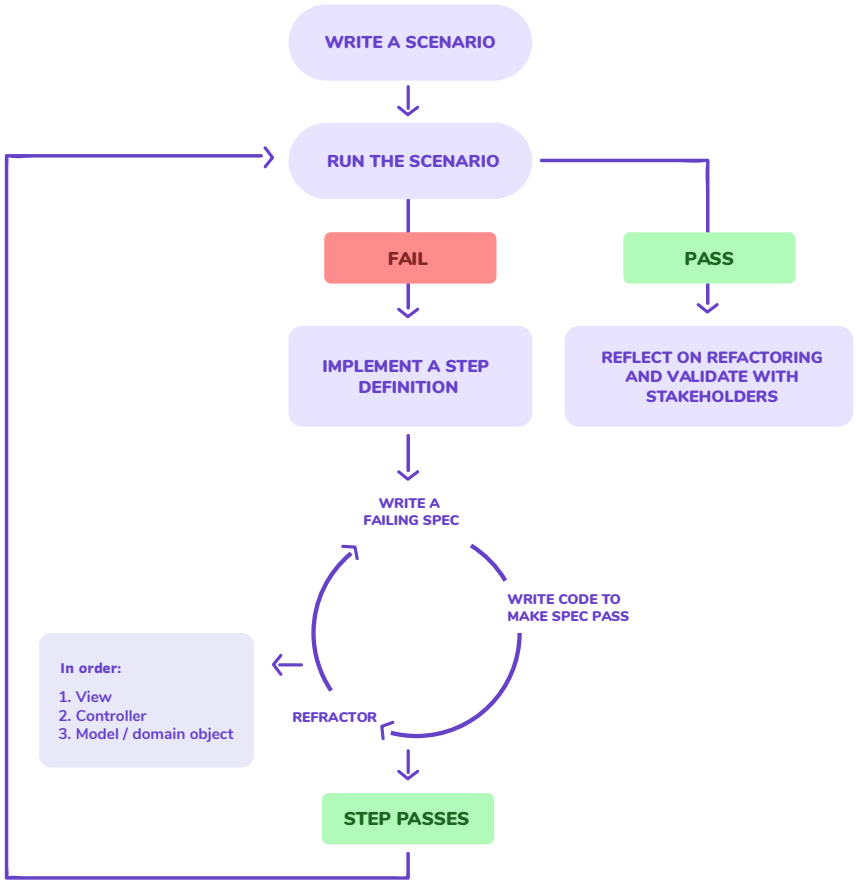
3. **Write a definition of the first failing or pending spec.** Run the scenario and watch it fail.

4. **Test-drive the implementation of a Rails view using the red-green-refactor cycle with RSpec.** You'll discover instance variables, controllers and controller actions that the view will need to do its job. This is also the only phase which has been proved to be optional in practice. An alternative approach is to simply prepare the views and controllers before moving on to the next step.

5. **Test-drive the controller using the red-green-refactor cycle with RSpec.** Make sure that the instance variables are assigned and that the actions respond correctly. The controllers are typically driven with a mocking approach. With the controller taken care of, you will know what the models or your custom objects should do.

6. **Test-drive those objects using the same red-green-refactor cycle with RSpec.** Make sure that they provide the methods needed by the controller and the view. If you are working on a new feature for which a model does not exist yet, you should now generate the model and the corresponding database migrations. At this point you'll know exactly what you need them to do.

7. Once you have implemented all the objects and methods you need and the corresponding specs are passing, **run the Cucumber scenario you started with to make sure that the step is satisfied.**



Once the first scenario step passes, move on to the next one and follow the same steps. Once your entire scenario has been implemented — the scenario is passing, along with all underlying specs — take a moment to reflect if there is something that you can refactor further.

Once you're sure that you've completed the scenario, either move on to the next one, or show your work to others. If you work with a team, create a pull request or an equivalent request for a code review. When there are no more related scenarios left, show your work to your project manager or client, asking them to verify that you've built the right thing by deploying a feature branch to a staging server.

Moving On

In this section, we explored how to apply behavior-driven development when developing a web application using Ruby on Rails, step by step. At this point you should be ready to start writing code the BDD way.

CHAPTER 4

Setting up a BDD Stack on a Rails 5 Application

Introduction

In this chapter we will guide you through the process of generating a new Rails 5 application, with all the necessary tools to set up a behaviour-driven development (BDD) flow.

WE WILL SET UP 5 TOOLS THAT REPRESENT THE BASE OF BDD DEVELOPMENT IN RAILS:

- RSpec for writing unit tests
- Cucumber for writing acceptance tests
- Shoulda Matchers for enhancing model tests
- Factory Bot for factory based database fixtures
- Database Cleaner for setting up a clean environment between test runs
- Rails Controller Testing for expanding our controlling testing arsenal

When you're finished setting up the project on your machine, we will set it up on Semaphore and establish a fully automated continuous integration workflow.

System Prerequisites

To follow our guide, you need to have the following installed on your Unix-based machine.

- Git,
- Ruby 2.4.0,
- Node.js, and
- PostgreSQL 9.5.

Bootstrapping a Rails Application

Let's start by installing Rails on our machines. Run the following command in your terminal:

```
gem install rails
```

Now, we are ready to generate a new Rails application.

```
rails new bdd-app -d postgresql
```

The `-d postgresql` specifies that our application will use PostgreSQL as our database management system instead of SQLite, which is set as a default for Rails.

Let's now switch to your application's directory:

```
cd bdd-app
```

Install your Rails application's dependencies:

```
bundle install --path vendor/bundle
```

By passing the `--path` parameter we are telling bundler to install gems in the `bdd-app/vendor/bundle` directory. If you leave off the parameter, gems will be installed globally, which isn't a good practice if you are working on more than one Ruby application on the development machine.

Finally, set up your database:

```
bundle exec rails db:create
bundle exec rails db:migrate
```

Installing RSpec

First, we will set up RSpec for writing unit specs.

Start by adding `rspec` in the `Gemfile` file, under the development and test group:

```
group :development, :test do
  ...
  gem 'rspec-rails', '~> 3.5'
end
```

Run `bundle install` to install the gem on your local machine.

To finish the install, invoke the Rails generator to set up the spec directory in your application.

```
bundle exec rails generate rspec:install
```

When the command finishes, you should have a new `spec` directory in your project. This directory is the base for all of your unit specs.

Installing Cucumber

Next, we will continue to set up Cucumber, the tool used for writing acceptance tests.

Add `cucumber-rails` gems to the `:development, :test` group of your Gemfile:

```
group :development, :test do
  ...
  gem 'cucumber-rails', require: false
end
```

Run `bundle install` to install the gem on your local machine.

To finish the install, invoke the rails generator to set up the `features` directory in your application:

```
$ bundle exec rails generate cucumber:install

Running via Spring preloader in process 2192
      create  config/cucumber.yml
      create  script/cucumber
       chmod  script/cucumber
      create  features/step_definitions
      create  features/step_definitions/.gitkeep
      create  features/support
      create  features/support/env.rb
       exist  lib/tasks
      create  lib/tasks/cucumber.rake
        gsub  config/database.yml
        gsub  config/database.yml
       force  config/database.yml
```

When the command finishes, you should have a new `features` directory in your project. This directory is the base for all of your acceptance tests.

Installing Shoulda-matchers

Shoulda-matchers gem speeds up our testing time by using wrappers around common Rails functionality, such as validations, associations and redirects.

To install `shoulda-matchers` we need to add `shoulda-matchers` gem to our development and test group inside the `Gemfile`:

```
group :development, :test do
  ...
  gem 'shoulda-matchers',
    git: 'https://github.com/thoughtbot/shoulda-matchers.git',
    branch: 'rails-5'
end
```

Install the gem by running:

```
bundle install
```

We need to configure this gem by specifying the test frameworks and libraries we want to use it with. Open `spec/rails_helper.rb` and paste the following block at the end:

```
Shoulda::Matchers.configure do |config|
  config.integrate do |with|
    with.test_framework :rspec

    with.library :active_record
    with.library :active_model
    with.library :action_controller
    with.library :rails
  end
end
```

Installing FactoryBot

Factory Bot is a gem used for making "fake" objects as test data. It is essentially a fixtures replacement with a clear definition syntax. It allows you to create fake objects in tests without providing a value for each attribute of the given object. If you don't provide any values to the attributes of the object, `factory_bot` uses the default values that are previously defined in factory's definition.

Add the following line to the development and test group of your `Gemfile`:

```
group :development, :test do
  ...
  gem 'factory_bot_rails'
end
```

Install the gem with:

```
bundle install
```

After the installation, you can place factories for your database models in the `spec/factories` directory.

Installing the Database Cleaner

We want to make sure that the state of the application is consistent every time we run our tests. The best way to do this is to clean all the data in the database between each test run, and construct a new clean state for each test.

The database-cleaner gem is a set of strategies for cleaning our database between test runs.

First, add the gem to the :test group in your Gemfile:

```
# Gemfile
group :test do
  gem 'database_cleaner'
end
```

Run bundle install to install the gem.

When the gem is installed, we will set up a cleaning strategy for RSpec tests. Open the spec/rails_helper.rb file, and add the following snippet:

```
RSpec.configure do |config|

  config.before(:suite) do
    DatabaseCleaner.strategy = :transaction
    DatabaseCleaner.clean_with(:truncation)
  end

  config.around(:each) do |example|
    DatabaseCleaner.cleaning do
      example.run
    end
  end

end
```

We don't have to make any modifications to Cucumber. The cucumber-rails generator already created all the necessary hooks to integrate itself with the database-cleaner gem.

Setting up the Rails Controller Testing gem

The rails-controller-testing gem expands the capabilities of our controller specs. It allows us to test variable and template assignments.

Add it to your Gemfile under the test group:

```
gem 'rails-controller-testing'
```

Run bundle install to install the gem.

RSpec automatically integrates with this gem. Adding the gem to your Gemfile is sufficient for the helpers to work in your controller specs.

Setting up a Repository

We will use GitHub to set up your repository for this project.

First, let's make sure that our repository is ready to be published and that we won't push any unnecessary file or directory to GitHub.

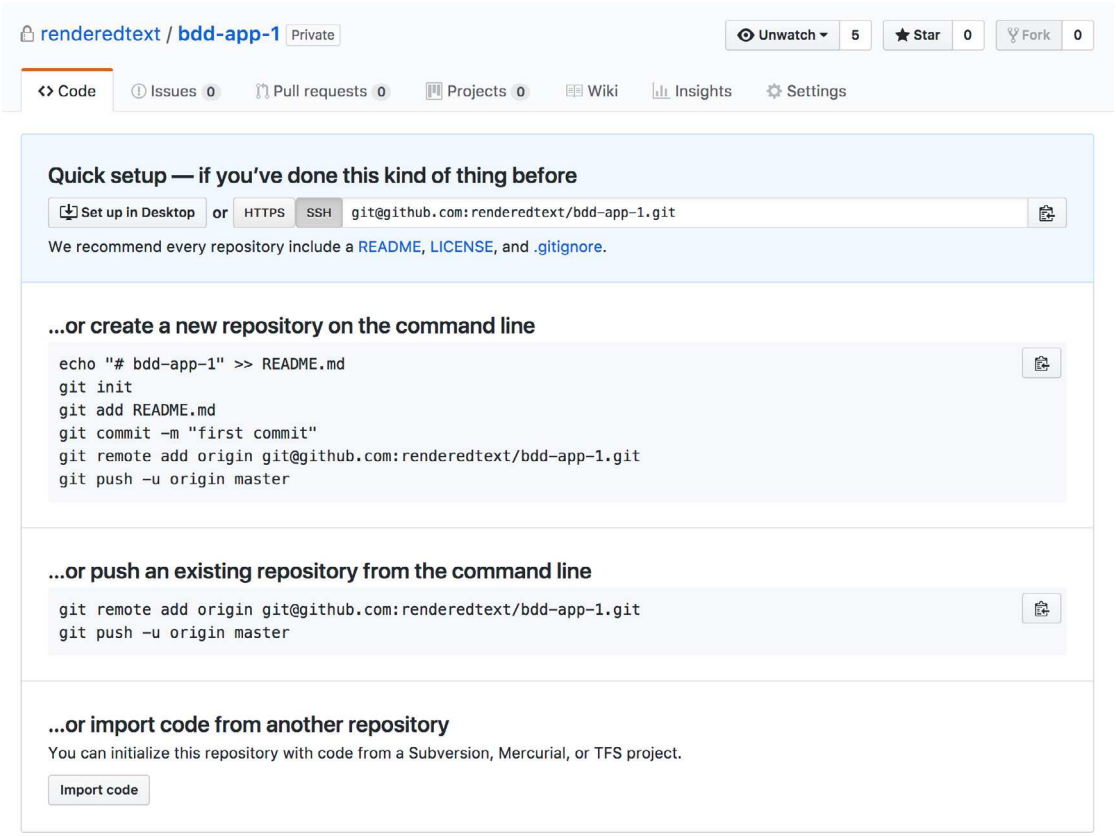
Add the vendor/bundle directory to your `.gitignore` file to prevent pushing gems to Github.

```
echo 'vendor/bundle' >> .gitignore
```

Then, let's create our first commit.

```
git add .
git commit -m "Bootstrapping Rails BDD application"
```

Visit https://github.com/new and create a new repository for your project. GitHub will give you the instructions how to connect your local repository with the remote one.



Finally, add the new remote and push your code to GitHub.

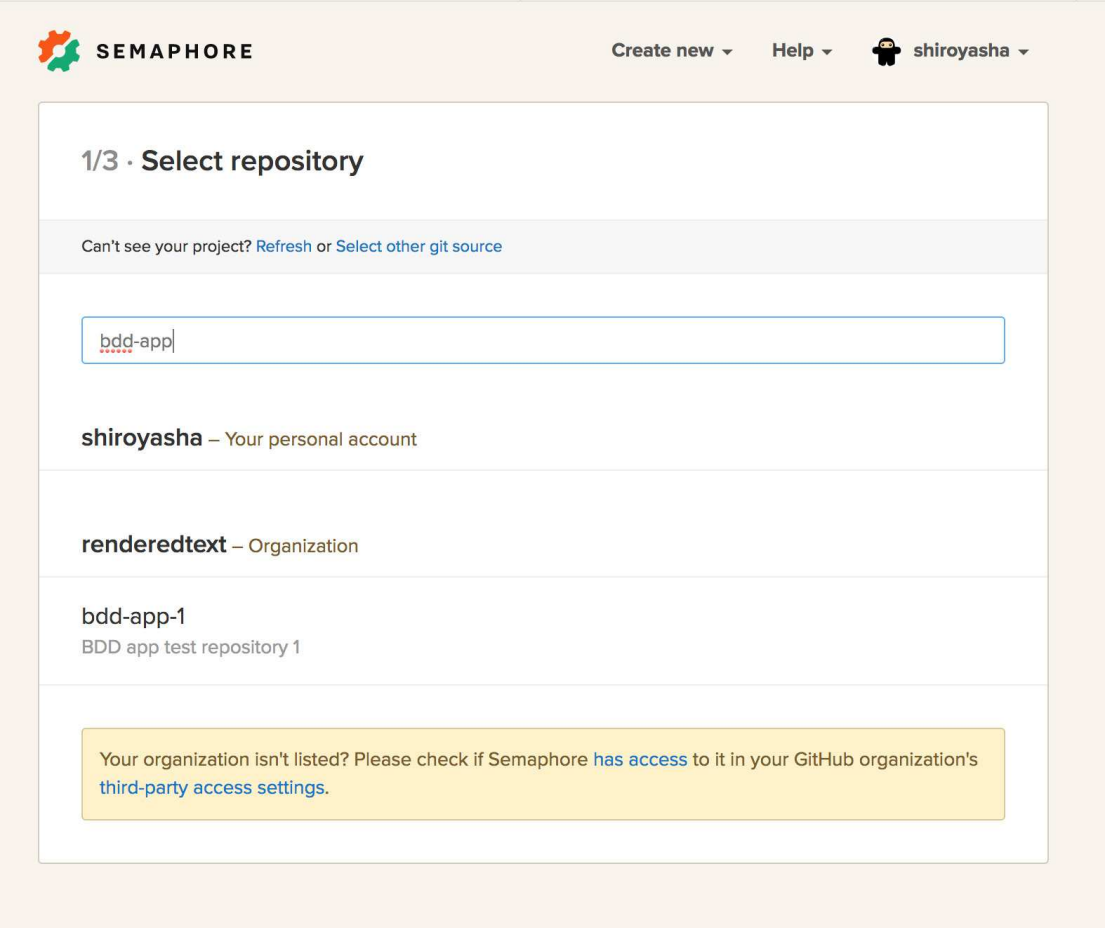
```
git remote add origin git@github.com:<YOUR-USERNAME>/<YOUR-REPO>.git
git push -u origin master
```

Setting up Continuous Integration

As the final step in this chapter, we will make sure that our project is started with a clean, green, build. This will be our first safe point of return for our application, and will act as the solid foundation for further development.

The first step is to create a free Semaphore account and then add a new project to Semaphore.

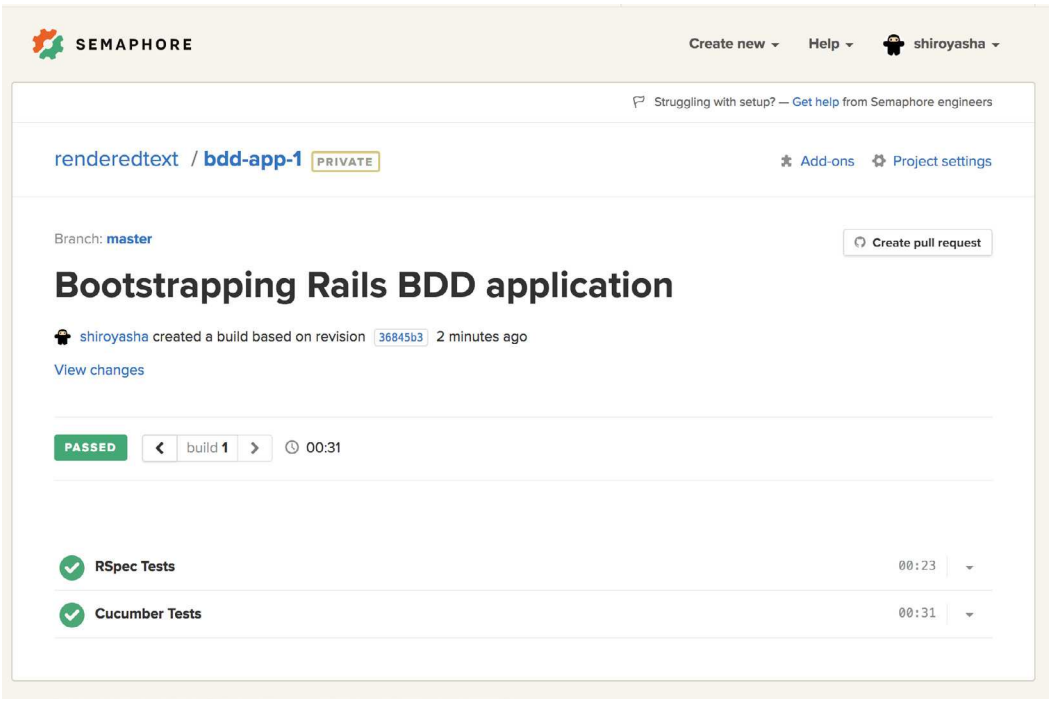
Choose your GitHub repository from the list:



Follow the setup guide. Semaphore will recognize that you have a Rails project with Postgresql, and will generate all the commands for a successful first build.

Let's tweak the command list, and add our RSpec and Cucumber test suites.

Hit Build With These Settings to start the first build for your project.



Conclusion

Congratulations! You've set up a great foundation for behaviour-driven development with Rails 5, RSpec and Cucumber.

CHAPTER 5

Bootstrapping a User Authentication System

Introduction

Most of web facing applications need some sort of an authentication mechanism to control access to the internal resources of the system. This is a natural continuation to the previous chapter where we've set up a base for behaviour driven development.

In this chapter, we will cover setting up a user authentication system in Rails using the Devise gem.

Defining our Goals

Before we start writing any code, we should define our **end goals**. This will focus our attention and help us write code more efficiently.

To cover all the basic use cases, we want to allow users to *sign up*, *log in*, *and log out*. Let's write down these requirements in a Cucumber feature file.

```
# features/authentication.feature

Feature: Authentication

  In order to use the website
  As a user
  I should be able to sign up, log in and log out
```

A Cucumber feature starts with a title and a comment written in plain English. While the original goal of this syntax design was to allow non-technical people to understand the scenarios, this also helps developers clarify their thoughts. By writing a scenario in advance, we define the scope of the scenario and spell out the functionality in plain English, effectively codifying the application design. This helps clarify our thoughts and manage the work that's ahead of us.

We continue with defining scenarios that describe the behaviour of our feature. First, the sign up scenario:

```
# features/authentication.feature

Scenario: Signing up
       Given I visit the homepage
       When I fill in the sign up form
       And I confirm the email
       Then should see that my account is confirmed
```

Next, we describe how our users will log in:

```
# features/authentication.feature

  Scenario: User Logs In
    Given I am a registered user
    And I visit the homepage
    When I fill in the login form
    Then I should be logged in
```

Finally, we will cover logging out:

```
# features/authentication.feature

  Scenario: User Logs Out
    Given I am a registered user
    And I am logged in
    And I visit the homepage
    When I click on the log out button
    Then I should be redirected to the log in page
```

Writing Tests for our Cucumber Feature

Now that the high level goals are laid out, we continue to define tests based on the steps in Cucumber file.

Cucumber helps us out at this point. If we run `bundle exec cucumber`, it displays a list of steps that are not defined.

Let's use the output of Cucumber to create a step definition file for user authentication:

```
# features/step_definitions/authentication_steps.rb

Given("I visit the homepage") do
  pending # Write code here that turns the phrase above into concrete actions
end

When("I fill in the sign up form") do
  pending # Write code here that turns the phrase above into concrete actions
end

When("I confirm the email") do
  pending # Write code here that turns the phrase above into concrete actions
end

Then("should see that my account is confirmed") do
  pending # Write code here that turns the phrase above into concrete actions
end

Given("I am a registered user") do
  pending # Write code here that turns the phrase above into concrete actions
end

When("I fill in the login form") do
  pending # Write code here that turns the phrase above into concrete actions
end

Then("I should be logged in") do
  pending # Write code here that turns the phrase above into concrete actions
end

Given("I am logged in") do
  pending # Write code here that turns the phrase above into concrete actions
end

When("I click on the log out button") do
  pending # Write code here that turns the phrase above into concrete actions
end

Then("I should be redirected to the log in page") do
  pending # Write code here that turns the phrase above into concrete actions
end
```

We have now our Cucumber steps, however all of them are still pending. Let's define them one by one.

A step definition is nothing more than plain Ruby code. You can write anything in the step definitions, but usually we use Capybara helpers to set up tests, and RSpec expectation to verify that the system ended up in the desired state.

We will use Capybara to navigate our application. It defines helpers for visiting pages in our application with the `visit` helper, clicks on links with `click_link` helper, and it has more features that are described in its documentation.

```
Given("I visit the homepage") do
  visit root_path
end

When("I fill in the sign up form") do
  click_link "Sign up"

  fill_in "user_email", :with => "tester@testdomain.test"
  fill_in "user_password", :with => "pa$$word"
  fill_in "user_password_confirmation", :with => "pa$$word"

  click_button "Sign up"
end
```

For the email verification steps, we will use the `email_spec` gem that provides helpers such as `open_email`, `visit_in_email`, and other email handling shortcuts that are very useful for testing.

```
Given("I confirm the email") do
  open_email("tester@testdomain.test")

  visit_in_email("Confirm my account")
end
```

The `Then` steps are usually used to set up expectation on the system. In our case, we will use the `expect` assertion to test whether the current page has the `Your email address has been successfully confirmed` message.

```
Then("I should see that my account is confirmed") do
  message = "Your email address has been successfully confirmed"

  expect(page).to have_content(message)
end
```

In the "I am a registered user" step we will make sure that there is an existing user account in our system. For this purpose, we will assume that we have a FactoryBot factory that can create a valid user account in our system.

```
Given("I am a registered user") do
  @registered_user = FactoryBot.create(:user,
                                       :email => "tester@testdomain.test",
                                       :password => "pa$$word")
end
```

The rest of the steps are a variation of the above cases. We either use Capybara to navigate our application and fill in forms, or we use the RSpec expect matcher to verify that a message appeared on the screen.

```
When("I fill in the login form") do
  fill_in "user_email", :with => "tester@testdomain.test"
  fill_in "user_password", :with => "pa$$word"

  click_button "Log in"
end

Then("I should be logged in") do
  expect(page).to have_content("Logged in")
end

Given("I am logged in") do
  visit root_path

  fill_in "user_email", :with => "tester@testdomain.test"
  fill_in "user_password", :with => "pa$$word"

  click_button "Log in"
end

When("I click on the log out button") do
  click_button "Log out"
end

Then("I should be redirected to the log in page") do
  expect(page).to have_content("Log in")
end
```

Every step is now defined. Let's run Cucumber to see what is the current state of our authentication system:

```
$ bundle exec cucumber

Failing Scenarios:
cucumber features/authentication.feature:7 # Scenario: Signing up
cucumber features/authentication.feature:13 # Scenario: User Logs In
cucumber features/authentication.feature:19 # Scenario: User Logs Out

3 scenarios (3 failed)
13 steps (3 failed, 10 skipped)
0m0.193s
```

All our tests are failing. This is the expected outcome as we have not yet implemented any code to support it.

Setting Up Devise

Devise is the de-facto authentication system for Rails applications. It takes care of storing user passwords securely and it comes prepackaged with all the views necessary for the user account management.

To install Devise we will add it to the `Gemfile`:

```
gem 'devise'
```

Run `bundle install` to install the gem.

Following a successful installation, run this command to generate Devise views, controllers, and models in your application:

```
bundle exec rails generate devise:install
```

The output of the `devise:install` command will suggest several changes in the application. We will use several of them.

To make the user mailer work in our local development environment, we will add the following snippet to the `config/environments/development.rb` and `config/environments/test.rb` files:

```
config.action_mailer.default_url_options = { host: 'localhost',
port: 3000 }
```

Add a route to our homepage so that Devise knows where to redirect registered users:

```
root to: “home#index”
```

Add alerts and notifications to the application layout `app/views/layouts/application.html.erb`. This will be used by Devise to display errors in the user authorization flow.

```
<p class=”notice”><%= notice %></p>
<p class=”alert”><%= alert %></p>
```

Finally, create a user model, and run the migrations:

```
$ bundle exec rails generate devise User
$ bundle exec rails db:migrate
```

Devise is now installed. Let’s run Cucumber again to see what is our progress:

```
$ bundle exec cucumber
...

Scenario: Signing up
  Given I am on the homepage
    uninitialized constant HomeController (ActionController::RoutingEr-
ror)
    ./features/step_definitions/authentication_steps.rb:2:
      in `”I am on the homepage”’
    features/authentication.feature:8:in `Given I am on the homepage’
  And I fill in the sign up form
  And I confirm the email
  Then I should see that my account is confirmed

...
```

Bootstrapping our Homepage

In the last section, we observed that the goals in our Cucumber feature require a Homepage, with a `Home` controller.

First, create an empty Home controller:

```
class HomeController < ApplicationController

  def index
  end

end
```

We will use Cucumber to guide us further in this process.

```
$ bundle exec cucuber

Given I am a registered user
   HomeControlle#index is missing a template for this request.

   request.formats: [“text/html”]
   request.variant: []
```

Based on the Cucumber error, we can deduce that we need to create a view for our controller action.

Let’s start with an RSpec unit test for a home controller:

```
# spec/controllers/home_controller_spec.rb

require “rails_helper”

RSpec.describe HomeController do
  let(:user) { instance_double(User) }

  before { log_in(user) }

  describe “GET #index” do
    it “returns status ok” do
      get :index

      expect(response.status).to be(200)
    end
  end

end
```

In the previous unit test, we have made use of the `log_in` helper to simulate a user who is logged in in our controller unit specs. We need to define this helper in our `spec/rails_helper.rb` test helper file.

First, load all the Devise test helpers, and a custom `ControllerHelpers` class.

```
require “support/controller_helpers”

RSpec.configure do |config|

  config.include Warden::Test::Helpers
  config.include Devise::TestHelpers, :type => :controller
  config.include ControllerHelpers, :type => :controller
```

Then, define the `log_in helper` in the controller helpers file:

```
module ControllerHelpers

  def log_in(user)
    warden = request.env[‘warden’]

    allow(warden).to receive(:authenticate!).and_return(user)
    allow(controller).to receive(:current_user).and_return(user)
  end

end
```

With this, the `log_in helper` is available in all our controller specs.

Finally, let’s add a view for our homepage:

```
$ mkdir app/views/home
$ echo “<h1>Homepage</h1>” > app/views/home/index.html.erb
```

Running Cucumber again, we can now see that the first step is green. Our home page is ready. Let’s protect it from non-authenticated users.

Add the `authenticate_user!` filter to the `application_controller.rb` file:

```
class ApplicatienController < ActionController::Base
  protect_from_forgery with: :exception

  before_action :authenticate_user!
end
```

This will protect all our pages from non-authenticated users and redirect them to the sign up page.

When we run Cucumber again, we can see that the second step in the sign up flow is completed and green. The only thing that’s missing now is the email verification step.

Verifying Email Verification Steps

First, add the email_spec gem to the Gemfile.

```
group :development, :test do
  gem 'email_spec'
end
```

Run bundle install to run the gem on your system.

To include the email helpers in the Cucumber environment, add the following to the features/support/env.rb file:

```
# Make sure this require is after you require cucumber/rails/world.
require 'email_spec' # add this line if you use spork
require 'email_spec/cucumber'
```

Running the Cucumber steps again, we can see that the email spec steps are working, but the verification step is not sent.

```
And I confirm the email
 Could not find email  in the mailbox for tester@testdomain.test.
 Found the following emails:

 [] (EmailSpec::CouldNotFindEmailError)
 ./features/step_definitions/authentication_steps.rb:15:
   in `"I confirm the email"'
 features/authentication.feature:10:in `And I confirm the email'

Then I should see that my account is confirmed
```

This is not a big surprise. By default, Devise does not send confirmation emails.

First, add :registerable, :confirmable to the app/models/user.rb:

```
devise :database_authenticatable, :registerable,
         :recoverable, :rememberable, :trackable, :validatable,
         :registerable, :confirmable
```

Create a new migration to add the new fields to the database:

```
bundle exec rails g migration add_confirmable_to_devise
```

Add the following columns to the generated migration:

```
class AddConfirmableToDevise < ActiveRecord::Migration[5.1]
  def change
    add_column :users, :confirmation_token, :string
    add_column :users, :confirmed_at, :datetime
    add_column :users, :confirmation_sent_at, :datetime
  end
end
```

Change the config.reconfirmable settings in the `config/initializers/devise.rb`:

```
config.reconfirmable = false
```

Change the user factory to generate confirmed users by default:

```
# spec/factories/users.rb

FactoryBot.define do
  factory :user do
    confirmed_at Time.now
  end
end
```

Finally, run the migrations to set up email confirmations:

```
bundle exec rails db:migrate
```

When you run Cucumber, you will notice that the scenario for signing up is green.

Implementing the Log Out Behaviour

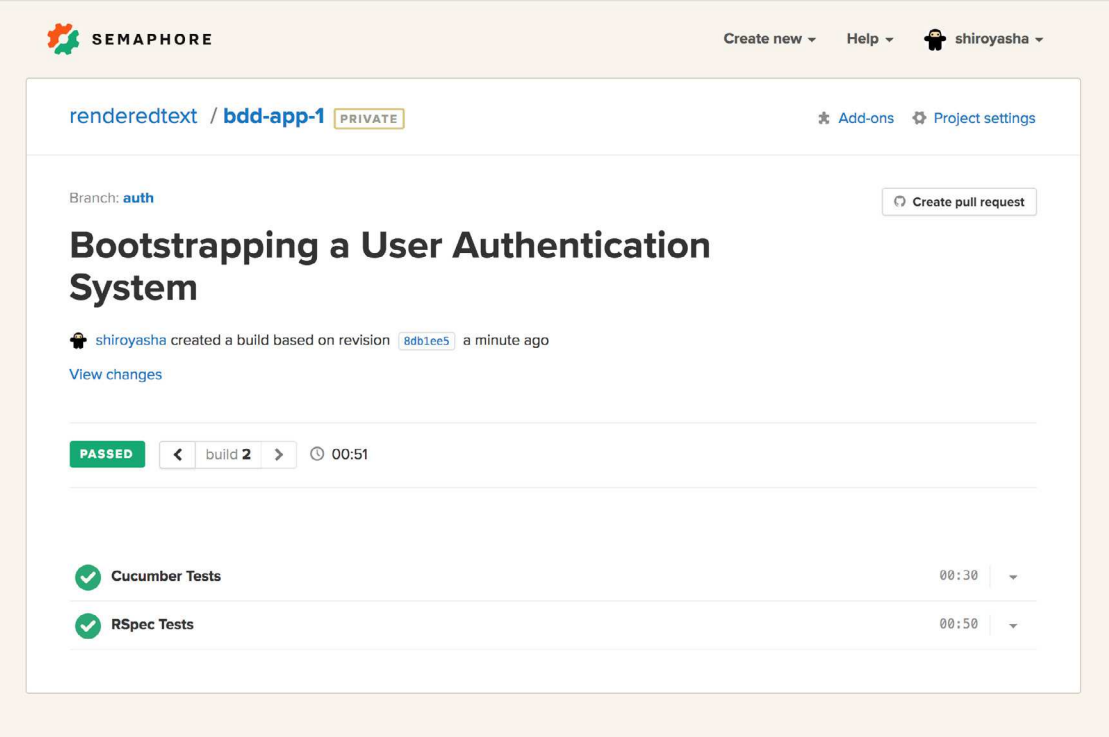
The only scenario that is not green at this point should be the log out scenario. Let's add a log out link to the application:

Add the following snippet to the `app/views/layouts/application.html.erb` file:

```
<% if user_signed_in? %>
  <%= link_to('Log out', destroy_user_session_path, method: :delete) %>
<% end %>
```

In the above snippet, both the `user_signed_in?` and `destroy_user_session_path` helpers are defined by Devise. Everything should be green at this point.

Let's push our code to GitHub, and wait for Semaphore to turn green.



Congratulations! We now have a working authentication system in Rails. It's time to start building the unique functionality of our application.

CHAPTER 6

Developing a CRUD Application by Following the BDD approach

Introduction

We've come a long way and covered the concept and theory behind behavior-driven development, set up the tools, and mapped new concepts to a basic Rails application. We are sure that you are eager to jump into the code and start following the BDD path at this point.

However, pure theory is not enough for you to effectively bootstrap your BDD journey. A sneak peek at a real world example will help you to close this gap.

We're going to build a CRUD (short for create, read, update, delete) application that helps the users keep track of their books. We will proceed in baby steps, writing our application from outside-in.

Cucumber and RSpec will be our main tools. We will also cover the Git flow, give hints how and when to communicate with your team, and how to make sure that a new feature is ready for shipping.

Understanding the Application

Our goal is to write a Rails-based web application that can keep an inventory of your books. Users of our application will be able to add, edit and remove books from their inventories.

While the scope of this project is small, we will write it in a way that will make it easy to maintain it and further develop it. The project should be able to grow, and to be safely and continuously updated. We will keep this in mind while writing specs and implementation, making sure that our tests are not crude and that they can cope with changing requirements.

Starting the Development of a New Feature

Every new feature starts from a green (all tests pass) point in the projects history. You should keep your master branch on git green all the time to provide a safe starting point for your team.

Let's switch to the master branch and make sure that we have the latest revision on our local development machine:

```
git checkout master
git pull
```

Now, let's create a new branch to make sure that our development doesn't hinder the productivity of our colleagues. We will call this branch book-inventory that reflects the scope and our primary goal on this branch:

```
git checkout -b book-inventory
```

Every change in this chapter will be a part of this branch.

Defining our End Goals with Cucumber Features

Behavior-driven development encourages us to start every new feature by defining our end goals first. This helps us stay focused on the task at hand. Write just enough code for the feature we are working on, while avoiding over-designing and trying to do too many things at once.

With Cucumber we can make sure that our end goals are solidified with a high level language and reputedly tested.

A Cucumber feature starts with a title and a comment, written in plain English. While the original goal of this syntax design was to allow non-technical people to understand the scenarios, this also helps us, developers, clarify our thoughts. By writing a scenario in advance, we define the scope of the scenario and write the functionality in plain English, effectively codifying the application design. This helps clarify our thoughts and manage the work that's ahead of us.

Let's describe our book inventory with a Cucumber feature.

```
# features/book_inventory.feature

Feature: Book Inventory

  In order to be able to keep track of my books
  As a user
  I should be able to keep an inventory of my books
```

Now, we start to peel the layers of the feature. We ask ourselves what basic assumptions we have about the users who are using this feature. For a start, we want to allow only registered and signed in users to keep track of their books:

```
# features/book_inventory.feature

Feature: Book Inventory

  In order to be able to keep track of my books
  As a user
  I should be able to keep an inventory of my books

  Background:
    Given I am a registered user
    And I am logged in
```

In Cucumber, we follow a `Given/When/Then` structure to represent our features. The `Given` part describes our assumptions about the current state, the `When` part represents user actions, and the `Then` part describes what changes when the user takes an action.

We will now describe our first interaction with the system — listing books in the inventory.

```
Scenario: Listing books in my inventory
  Given I have populated my inventory with several books
  When I visit the homepage
  Then I should see the list of my books
```

Defining Cucumber Steps and Making our Goals Testable

Now that our first goal is set, let's run Cucumber and follow its output in our implementation.

```
$ bundle exec cucumber

4 scenarios (1 undefined, 3 passed)
18 steps (1 skipped, 2 undefined, 15 passed)
0m0.758s

You can implement step definitions for undefined steps with these
snippets:

Given("I have populated my inventory with several books") do
  pending # Write code here that turns the phrase above into
  concrete actions
end

Then("I should see the list of my books") do
  pending # Write code here that turns the phrase above into
  concrete actions
end
```

Neat! Cucumber tells us that we have set up our goals, but we have not defined their meaning.

The first step is to copy the output from Cucumber into a Cucumber step definition.

```
# features/step_definitions/book_inventory_steps.rb

Given("I have populated my inventory with several books") do
  pending # Write code here that turns the phrase above into
  concrete actions
end

Then("I should see the list of my books") do
  pending # Write code here that turns the phrase above into
  concrete actions
end
```

Next step is to implement the step definitions and set up tests that verify the completeness of our new feature.

Don't sweat if you don't understand every part of the code bellow. Try to focus on the flow and general principle of implementation.

```
# features/step_definitions/book_inventory_steps.rb

Given(/^I have populated my inventory with several books$/) do
  FactoryBot.create(:book,
                    :user => @registered_user,
                    :name => "Don Quixote",
                    :author => "Miguel de Cervantes")

  FactoryBot.create(:book,
                    :user => @registered_user,
                    :name => "Moby Dick",
                    :author => "Herman Melville")
end

Then(/^I should see the list of my books$/) do
  expect(page).to have_content("Don Quixote")
  expect(page).to have_content("Moby Dick")
end
```

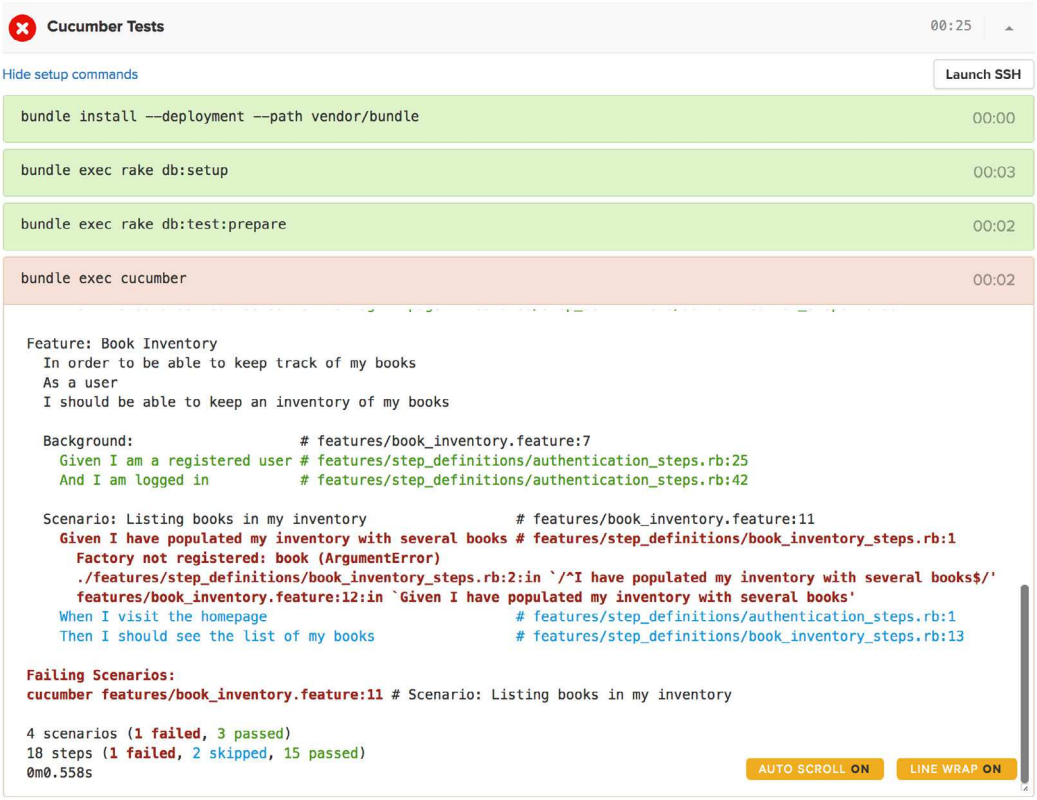
In the above snippet, we use FactoryBot to inject records into the database. The `@registered_user` user comes from the `Given I am a registered user` step definition that we defined in the previous chapter while setting up a user account management system.

This is a good point to stop, and create out first commit.

```
git add . && git commit -m "Describe book inventory listing" && git push
```

Creating a commit creates a safe point of return. If our implementation goes astray, we can safely delete everything and return to this revision in Git and try again with a different approach.

Apart from creating a safe point of return, we also want to validate that our specs are set up correctly. We should have RED tests on our CI at this point.



Cucumber-driven Implementation of the Book Inventory Listing

We continue to use the output of Cucumber to guide us in the right direction.

```
$ bundle exec cucumber features/book_inventory.feature

  Scenario: Listing books in my inventory
    Given I have populated my inventory with several books
      Factory not registered: book (ArgumentError)
      ./features/step_definitions/book_inventory_steps.rb:16:
        in `/^I have populated my inventory with several books$/'
      features/book_inventory.feature:12:
        in `Given I have populated my inventory with several books'
    When I visit the home page
    Then I should see the list of my books
```

Cucumber tells us that the book model, along with its factory, is not available in our system. This is not a surprise. We still need a lot of code to complete our feature, from data models to views and factories.

Rails offers a quick and easy way to generate a good chunk of our implementation by scaffolding.

We want to scaffold a book model that belongs to users. Every book will have a name and an author. Let's pass this information to Rails:

```
bundle exec rails generate scaffold book user:belongs_to \
  name:string author:string
  --no-controller-specs \
  --no-view-specs \
  --no-helper-specs \
  --no-system-tests \
  --no-request-specs \
  --no-helper-specs

bundle exec rails db:migrate db:test:prepare
```

Let's observe the changes in Cucumber:

```
$ bundle exec cucumber features/book_inventory.feature

  Scenario: Listing books in my inventory
            # features/book_inventory.feature:11
    Given I have populated my inventory with several books
      # features/step_definitions/book_inventory_steps.rb:15
    When I visit the home page
     # features/step_definitions/book_inventory_steps.rb:20
    Then I should see the list of my books
     # features/step_definitions/book_inventory_steps.rb:24
     expected to find text "Don Quixote" in "Log out Homepage" (
     RSpec::Expectations::ExpectationNotMetError)
      ./features/step_definitions/book_inventory_steps.rb:25:in
     `/^I should see the list of my books$/'
      features/book_inventory.feature:14:in
     `Then I should see the list of my books'
```

The first two steps are passing. We have satisfied a good part of our goals just by generating a book model.

Time to commit the changes.

```
git add .
git commit -m "Generate book scaffold"
git push
```

Setting Up Unit Tests for the Book Inventory Listing

The first two steps have passed, but we still need to deal with the third step. It seems that our book is not listed on the page.

Before we continue, let's set up specs for our controller and model so we can be sure that individual units are working as expected. For unit testing we use RSpec.

Controllers are best tested with a mocking approach, since they tie many things together. The freedom you get from mocking should be used as an advantage in molding the controller method's code to an ideal shape. This aids us to test the **behavior** of our systems.

With this style of testing we focus on the interaction of our components instead of focusing on the result of an operation. For example, we would write our tests for a `UserMailer as expect(UserMailer).to receive(:send_signup_email)` instead of testing whether the mail was successfully delivered. This is popularly called the London/ Interaction school of testing after London's Extreme Tuesday Club where it became popular.

We will start with the controller's index action.

```
# spec/controllers/books_controller_spec.rb

require "rails_helper"

RSpec.describe BooksController do

  let(:user) { instance_double(User) }

  before { log_in(user) }

  describe "GET #index" do
    let(:books) { [
      instance_double(Book),
      instance_double(Book)
    ] }

    before do
      allow(user).to receive(:books).and_return(books)

      get :index
    end

    it "looks up all books that belong to the current user" do
      expect(assigns(:books)).to eq(books)
    end
  end

end
```

In the above code block, we use the get method that is included in every controller spec. Controller specs have helpers for other HTTP calls as well, covered in detail on RSpec Rails documentation page.

RSpec tells us that users and books are not yet connected:

```
$ bundle exec rspec spec/controllers/books_controller_spec.rb

1) BooksController GET
   #index looks up all books that belong to the current user
   Failure/Error: allow(user).to receive(:books).and_return(books)
     the User class does not implement the instance method: books
```

Running RSpec again validates our implementation. We now want to make sure that our data layer and the associations are correct.

We will use shoulda-matchers to write specs for the user and book models.

```
# spec/models/book_spec.rb

require “rails_helper”

RSpec.describe Book, type: :model do

  # associations
  it { is_expected.to belong_to(:user) }

  # columns
  it { is_expected.to have_db_column(:name).of_type(:string) }
  it { is_expected.to have_db_column(:author).of_type(:string) }
  it { is_expected.to have_db_column(:created_at).of_type(:datetime) }
  it { is_expected.to have_db_column(:updated_at).of_type(:datetime) }

end
```

```
# spec/models/user_spec.rb

RSpec.describe User, type: :model do

  it { is_expected.to have_many(:books).dependent(:delete_all) }

end
```

The tests for the model are ready. Let’s connect our users and books in our application:

```
# app/models/user.rb

has_many :books, :dependent => :delete_all
```

Let's write a minimal implementation of the index method that matches our assumptions:

```
class BooksController < ApplicationController

  # ...

  def index
    @books = current_user.books
  end

  # ...

end
```

Running RSpec again tells us that every unit in our system passes our assumptions:

```
$ bundle exec rspec

Finished in 0.10461 seconds (files took 2.46 seconds to load)
16 examples, 0 failures
```

Completing the Book Inventory Listing

Now that we confirmed that every unit works, we can go back to Cucumber.

```
$ bundle exec cucumber features/book_inventory.feature

  Scenario: Listing books in my inventory
    Given I have populated my inventory with several books
    When I visit the home page
    Then I should see the list of my books
      expected to find text "Don Quixote" in "Log out Homepage"
        (RSpec::Expectations::ExpectationNotMetError)
      ./features/step_definitions/book_inventory_steps.rb:25:
        in `/^I should see the list of my books$/'
      features/book_inventory.feature:14:
        in `Then I should see the list of my books'
```

We still hit the same error. This is a good point to introduce debug steps in the Cucumber steps:

```
# features/step_definitions/book_inventory_steps.rb

Then(/^I should see the list of my books$/) do
  puts page.body  # Debug step: Display the HTML page.

  expect(page).to have_content("Don Quixote")
  expect(page).to have_content("Moby Dick")
end
```

Run Cucumber again:

```
$ bundle exec cucumber features/book_inventory.feature

    Then I should see the list of my books
      # features/step_definitions/book_inventory_steps.rb:24

       <!DOCTYPE html>
      <html>
        <body>
          <h1>Hello</h1>
        </body>
      </html>

      expected to find text “Don Quixote” in “Hello”
      (RSpec::Expectations::ExpectationNotMetError)
      ./features/step_definitions/book_inventory_steps.rb:27:in
      `/^I should see the list of my books$/’
      features/book_inventory.feature:14:in
       `Then I should see the list of my books’
```

The root_path is not directed to the book index controller. Let’s edit Rails routes:

```
Rails.application.routes.draw do
  resources :books

  devise_for :users

  root to: “books#index”
end
```

At this point, the Home controller defined in the previous step becomes deprecated. Delete it:

```
rm app/controllers/home_controller.rb
rm spec/controllers/home_controller_spec.rb
```

Now, we go back to delete the debug step and run Cucumber again:

```
$ bundle exec cucumber features/book_inventory.feature

1 scenario (1 passed)
5 steps (5 passed)
0m0.409s
```

A green test! Let's run the whole test suite just to make sure that everything passes:

```
$ bundle exec cucumber

Scenario: User Logs In
  Given I am a registered user
  And I visit the homepage
  When I fill in the login form
  Then I should be logged in
    expected to find text "Homepage" in "Log out Signed in successfully."
    ./features/step_definitions/authentication_steps.rb:39:
      in `"I should be logged in"'
    features/authentication.feature:17:
      in `Then I should be logged in'

Failing Scenarios:
cucumber features/authentication.feature:13 # Scenario: User Logs In

4 scenarios (1 failed, 3 passed)
18 steps (1 failed, 17 passed)
0m0.843s
```

Changing the root path broke our authentication step. Let's rewrite the spec to follow our new requirements:

```
# features/step_definitions/authentication_steps.rb

Then("I should be logged in") do
  expect(page).to have_content("Books")
end
```

Let's make sure that everything works.

```
$ bundle exec cucumber

2 scenarios (2 passed)
9 steps (9 passed)
0m0.768s
```

Everything is green! It's time to make a commit.

```
git add .
git commit -m "Implement book inventory listing"
git push
```

We have now finished our first feature by following the BDD path.

Creating New Books

In the previous section, we have completed one full BDD cycle. Guided by high level specs, we dug deep into unit tests, and finally to the implementation.

When you start with BDD for the first time, a lot of the steps seem cumbersome, giving you the feeling that BDD drastically reduces productivity. However, this is very far from the truth. As developers get used to BDD, we become more efficient with defining Cucumber scenarios. Writing tests becomes second nature to most of us, and the red -> green -> refactor cycle increases our productivity by making sure that we can make changes in the code without breaking an existing feature.

We will continue to follow the BDD pattern to implement the rest of the book inventory feature, but this time with an increased tempo.

Starting from the top, we define our goals for adding new books into the inventory.

```
# features/book_inventory.feature

Scenario: Adding a new book to the inventory
  When I submit a new book to my inventory
  Then I should see the new book in my inventory
```

```
# features/step_definitions/book_inventory_steps.rb

When(/^I submit a new book to my inventory$/) do
  click_link “New Book”

  fill_in “book_name”, :with => “War and Peace”
  fill_in “book_author”, :with => “Leo Tolstoy”

  click_button “Create Book”
end

Then(/^I should see the new book in my inventory$/) do
  visit root_path

  expect(page).to have_content(“War and Peace”)
  expect(page).to have_content(“Leo Tolstoy”)
end
```

While defining step definitions, we heavily rely on Capybara to interact with the browser. Commands like `fill_in`, `click_link`, and `visit all` come from here. There are more commands available in the framework, ranging from simple UI interactions, to more complex commands that allow you to run custom JavaScript on the page. Keeping a browser tab open with Capybara's documentation is an excellent way to boost your Cucumber knowledge.

```
$ bundle exec cucumber

Scenario: Adding a new book to the inventory
          # features/book_inventory.feature:16
  When I submit a new book to my inventory
       # features/step_definitions/book_inventory_steps.rb:18
  Then I should see the new book in my inventory
       # features/step_definitions/book_inventory_steps.rb:27
       expected to find text “War and Peace” in “Log out Books Use Name
Author New Book” (RSpec::Expectations::ExpectationNotMetError)
      ./features/step_definitions/book_inventory_steps.rb:30:in
      `/^I should see the new book in my inventory$/’
      features/book_inventory.feature:18:in `
      Then I should see the new book in my inventory’

Failing Scenarios:
cucumber features/book_inventory.feature:16 # Scenario: Adding a new
book to the inventory

5 scenarios (1 failed, 4 passed)
22 steps (1 failed, 21 passed)
```

With the high level goals set in place, we dig deeper into individual units. This time, we want to make sure that our controllers know how to create new books and attach them to an existing user.

Let’s define a spec for the `#create` action, and make sure that a new book is created and attached to the user.

```
# spec/controllers/books_controller_spec.rb

describe “POST #create” do
       let(:book) { FactoryBot.build_stubbed(:book) }
  let(:params) { { :name => “Moby-Dick”, :author => “Herman Melville” } }

  before do
            allow(book).to receive(:save)
    allow(user).to receive_message_chain(:books, :build).and_return(book)
  end

  it “saves the book” do
    post :create, :params => { :book => params }

    expect(book).to have_received(:save)
  end
end
```

The implementation should look as follows:

```
# app/controllers/books_controller.rb

def create
  @book = current_user.books.build(book_params)

  @book.save
end
```

Let's now define what happens when the book creation succeeds. For this purpose, we will introduce a new context that describes the state of the system after the save action was called.

A good practice is to set up the *context* in a before step. In our case, the context is that the save action returns true.

```
context "when the book is succesfully saved" do
  before do
    allow(book).to receive(:save).and_return(true)

    post :create, :params => { :book => params }
  end

  it "redirects to the book show page" do
    expect(response).to redirect_to(book_path(book))
  end

  it "redirects to the book show page" do
    expect(flash[:notice]).to eq("Book was successfully created.")
  end
end
```

Here's the implementation:

```
# app/controllers/books_controller.rb

def create
  @book = current_user.books.build(book_params)

  if @book.save
    redirect_to @book, notice: 'Book was successfully created.'
  end
end
```

Let's now cover the negative path with another context:

```
context “when the book can’t be saved” do
  before do
    allow(book).to receive(:save).and_return(false)

    post :create, :params => { :book => params }
  end

  it “redirects back to the new page” do
    expect(response).to render_template(:new)
  end
end
```

The implementation should look as follows:

```
# app/controllers/books_controller.rb

def create
  @book = current_user.books.build(book_params)

  if @book.save
    redirect_to @book, notice: ‘Book was successfully created.’
  else
    render :new
  end
end
```

As we are always attaching the book to the current user, we can edit the book form and remove the now deprecated user input field:

```
# remove the following lines from app/views/books/_form.html.erb

<div class=”field”>
  <%= form.label :user_id %>
  <%= form.text_field :user_id, id: :book_user_id %>
</div>
```

It's time to commit the new scenario:

```
git add .
git commit -m “Add books to the inventory”
git push
```

Updating and Deleting

We will now finish the remaining two features — editing a book and removing a book from the book inventory.

This time, we will implement two scenarios in one BDD cycle. As developers get used to the tooling they can get more ambitious and implement more scenarios in one go, sometimes a whole feature can be defined before the implementation.

As usual, let's start from the top and define the Cucumber scenarios:

```
# features/book_inventory.feature

Scenario: Changing the name of a book
  Given I have a book in my inventory
  When I change the title of my book
  Then I should see the book with the new title in my inventory

Scenario: Removing a book from my inventory
  Given I have a book in my inventory
  When I remove a book from my inventory
  Then I should not see it listing in the inventory anymore
```

Guided by Cucumber's output, we construct the following step definitions:

```
# features/step_definitions/book_inventory_steps.rb

Given(/^I have a book in my inventory$/) do
  FactoryBot.create(:book, :user => @registered_user, :name => “War and
Peace”, :author => “Leo Tolstoy”)
end

When(/^I change the title of my book$/) do
  visit root_path

  click_link “Edit”

  fill_in “book_name”, :with => “Guerra y paz”

  click_button “Update Book”
end

Then(/^I should see the book with the new title in my inventory$/) do
  visit root_path

  expect(page).to_not have_content(“War and Peace”)
  expect(page).to have_content(“Guerra y paz”)
end

When(/^I remove a book from my inventory$/) do
  visit root_path

  click_link “Destroy”
end

Then(/^I should not see it listing in the inventory anymore$/) do
  expect(page).to_not have_content(“War and Peace”)
end
```

Now that the high level goals are set, it's time to descend into the units. First, we'll set up tests for the `#update` action, covering the positive and negative paths of execution:

```
# spec/controllers/books_controller_spec.rb

describe "PATCH #update" do
  let(:book) { FactoryBot.build_stubbed(:book) }

  before do
    allow(Book).to receive(:find).and_return(book)
    allow(book).to receive(:update).and_return(true)
  end

  it "updates the book" do
    patch :update, :params => {
      :id => book.id,
      :book => { :name => "New Name" } }

    expect(book).to have_received(:update)
  end

  context "when the update succeds" do
    it "redirects to the book page" do
      patch :update, :params => {
        :id => book.id,
        :book => { :name => "New Name" } }

      expect(response).to redirect_to(book_path(book))
    end
  end

  context "when the update fails" do
    before do
      allow(book).to receive(:update).and_return(false)
    end

    it "renders the edit page again" do
      patch :update, :params => {
        :id => book.id,
        :book => { :name => "New Name" } }

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

At each step, we make sure to run our test suite, and verify that the controller is implemented correctly. In our case, Rails already generated a scaffold that matches our specification.

Remember, even if Rails auto-generated code for your feature, it is crucial to cover it with tests. They help us set up a fast feedback loop and make it easy to introduce changes in the behaviour.

Next, let's cover the #destroy method with unit specs.

```
# spec/controllers/books_controller_spec.rb

describe "DELETE #destroy" do
  let(:book) { FactoryBot.build_stubbed(:book) }

  before do
    allow(Book).to receive(:find).and_return(book)
    allow(book).to receive(:destroy)

    delete :destroy, :params => { :id => book.id }
  end

  it "deletes the book" do
    expect(book).to have_received(:destroy)
  end

  it "redirects to the index page" do
    expect(response).to redirect_to(books_path)
  end
end
```

When all unit tests are green, we can zoom out and return to our Cucumber feature. Everything should be passing at this point.

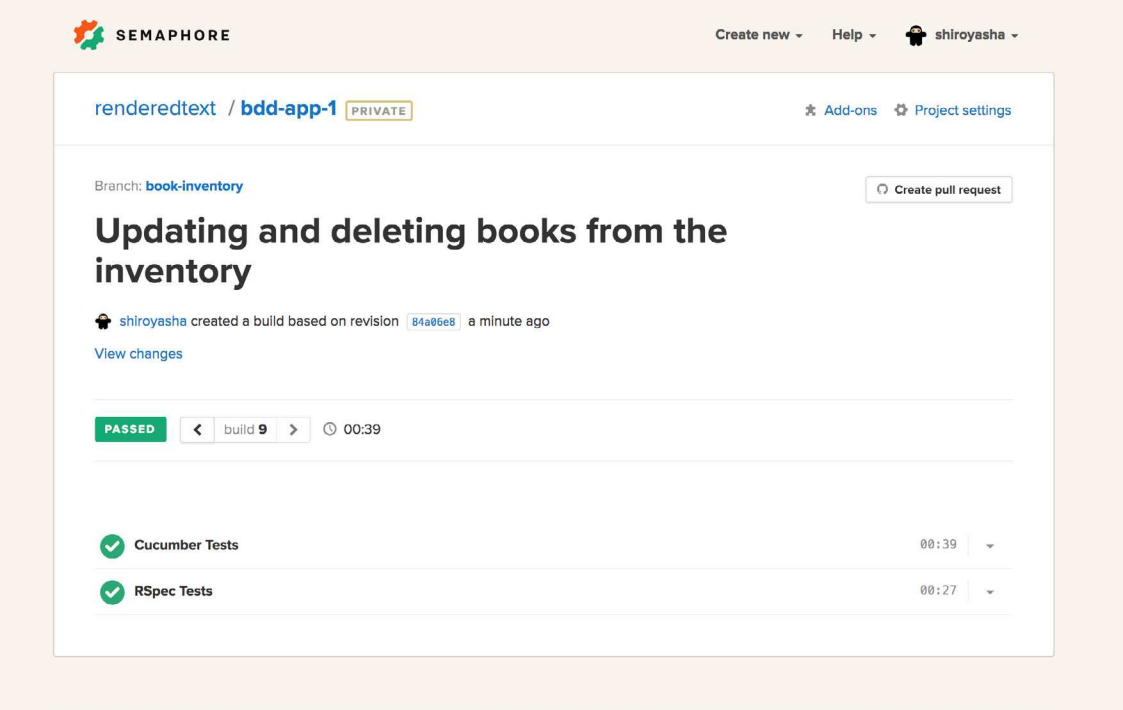
Let's commit and push:

```
git add .
git commit -m "Updating and deleting books from the inventory"
git push
```

Finishing a Feature and Merging into Master

All our initial goals are implemented. Now, we prepare a pull request and ask our colleagues for a review. This often forgotten step is crucial for delivering good software.

We will use the name of the feature, book inventory, to name our pull request.



When Semaphore reports that our build is green, we can safely merge our feature branch, and automatically deploy it without worrying that we'll break something in production.

Final Words

Congratulations, you've made it through the entire Rails Testing Handbook. We wrote this book with the goal to help you write clean code and build maintainable applications. Now it's up to you — go make something awesome!

Tell Us What You Think

We would absolutely love to hear your feedback. What did you get out of reading this book? How easy/hard was it to follow? What would you like to see us cover in another book?

Please write to us at learn@semaphoreci.com.

Share this Book

Please share this book with your colleagues, friends and anyone who you think might benefit from it.

Further Reading

1. RSpec documentation
2. Cucumber documentation
3. Capybara documentation
4. Semaphore Engineering Blog
5. Ruby Tutorials on Semaphore Community

About Semaphore

Semaphore helps you continuously test and deploy Ruby code at the push of a button. It lets you automatically parallelize your Ruby tests, get feedback right inside pull requests, and deploy more often in a unified workflow. Already trusted by thousands of businesses around the globe, Semaphore can help your team move faster too.

You can use a coupon code RUBYBOOK1 to get a $50 credit on a new Semaphore account.