In this chapter, you'll see:

- Replacing Rails' testing framework with RSpec
- Using Slim for HTML templates instead of ERB

CHAPTER 24

Customizing and Extending Rails

As you've come to learn, Rails provides an answer for almost every question you have about building a modern web application. It provides the basics for handling requests, accessing a database, writing user interfaces, and running tests. It does this by having a tightly integrated design, which is often referred to as Rails being "opinionated software."

This tight coupling comes at a price. If, for example, the way Rails manages CSS doesn't meet the needs of your project, you could be in trouble. Or if you prefer to write your tests in a different way, Rails doesn't give you a lot of options. Or does it? In the early days of Rails, customizing it was difficult or impossible. Starting with Rails 3, much effort was expended to make Rails more customizable. With Rails 7, developers have the flexibility to use the tools they prefer or that work the way they want to work. That's what we'll explore in this chapter.

We'll replace four parts of Rails in this chapter. First, we'll write a Web Component instead of using Stimulus. Then we'll see how to use RSpec instead of Rails' default testing library to write our tests. Next, we'll replace ERB for the alternative templating language Slim. Finally, we'll see how to manage CSS using Webpack instead of putting it in app/assets/stylesheets. This chapter will demonstrate another benefit to Rails, which is that you don't have to throw out the parts that work for you to use alternatives that work better. Let's get started.

Creating a Reusable Web Component

Web Components[1] are an industry standard way of extending HTML itself to implement custom behaviors and presentation.

1. https://developer.mozilla.org/en-US/docs/Web/Web_Components

You don't need to start from scratch when building a web component. You can build upon a rich ecosystem of npm[2] packages. We'll make use of lit.[3] We start by "pinning" it to our application so that it can be imported:

```
> bin/importmap pin lit
Pinning "lit" to https://.../index.js
Pinning "@lit/reactive-element" to https://.../reactive-element.js
Pinning "lit-element/lit-element.js" to https://.../lit-element.js
Pinning "lit-html" to https://.../lit-html.js
Pinning "lit-html/is-server.js" to https://.../is-server.js
```

Next, we'll write a web component. The following example renders the current time in blue. Create a directory named app/javascript/elements and create a file named current-time.js in that directory with the following contents:

```
import {html, css, LitElement} from 'lit';

class CurrentTime extends LitElement {
  static styles = css`span { color: blue }`;

  render() {
    return html`<span>${new Date().toLocaleTimeString()}</span>`;
  }
}

customElements.define('current-time', CurrentTime);
```

This code imports three properties from the lit package and defines a class that extends LitElement by defining a style that's scoped to this single element function that returns an HTML fragment. Finally, a new custom element is defined and associated with this class.

Next, we import this file into our application by adding a single line to app/javascript/application.js:

```
// Configure your import map in config/importmap.rb.
// Read more: https://github.com/rails/importmap-rails
import "@hotwired/turbo-rails"
import "controllers"

<!-- START_HIGHLIGHT -->
import "./elements/current-time.js"
<!-- END_HIGHLIGHT -->
```

With this in place, the current time can be added to any HTML template by adding the following HTML:

```
<current-time>
```

2. https://www.npmjs.com/
3. https://lit.dev/

This just scratches the surface of what can be done with Web Components. On the lit site you can find plenty of examples. A good place to start is on the page for Reactive Controllers,[4] which shows how you can add state and reactivity to a clock element.

Testing with RSpec

RSpec is an alternative to MiniTest, which Rails uses. It's different in almost every way, and many developers prefer it. Here's what one of our existing tests might look like written in RSpec:

```
RSpec.describe Cart do

  let(:cart)     { Cart.create }
  let(:book_one) { products(:ruby) }
  let(:book_two) { products(:two) }

  before do
    cart.add_product(book_one).save!
    cart.add_product(book_two).save!
  end

  it "can have multiple products added" do
    expect(cart.line_items.size).to eq(2)
  end

  it "calculates the total price of all products" do
    expect(cart.total_price).to eq(book_one.price + book_two.price)
  end
end
```

It almost looks like a different programming language! Developers who prefer RSpec like that the test reads like English: "Describe Cart, it can have multiple products added, expect cart.line_items.size to eq 2."

We're going to quickly go through how to write tests in RSpec without too much explanation. A great book for that is already available—*Effective Testing with RSpec 3 [MD17]*—so we'll learn just enough RSpec to see it working with Rails, which demonstrates Rails' configurability. Although many developers who use RSpec set it up from the start of a project, you don't have to. RSpec can be added at any time, and that's what we'll do here.

Add rspec-rails to your Gemfile, putting it in the development and test groups:

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

4. https://lit.dev/docs/composition/controllers/

After you bundle install, a new generator will set up RSpec for you:

```
> bin/rails generate rspec:install
    create  .rspec
    create  spec
    create  spec/spec_helper.rb
    create  spec/rails_helper.rb
```

Verify the configuration is working by running the new task Rspec installed, spec:

```
> bin/rails spec
No examples found.

Finished in 0.00058 seconds (files took 0.11481 seconds to load)
0 examples, 0 failures
```

Let's reimplement the test for Cart as an RSpec test or *spec*. RSpec includes generators to create starter specs for us, similar to what Rails does with scaffolding. To create a model spec, use the spec:model generator:

```
> bin/rails generate spec:model Cart
      create  spec/models/cart_spec.rb
```

Now rerun spec, and we can see RSpec's generator has created a pending spec:

```
> bin/rails spec
Pending: (Failures listed here are expected and do not affect
          your suite's status)

  1) Cart add some examples to (or delete) spec/models/cart_spec.rb
     # Not yet implemented
     # ./spec/models/cart_spec.rb:4

Finished in 0.00284 seconds (files took 1.73 seconds to load)
1 example, 0 failures, 1 pending
```

To reimplement the test for Cart as a spec, let's first review the existing test:

rails7/depot_u/test/models/cart_test.rb

```
require 'test_helper'

class CartTest < ActiveSupport::TestCase
  def setup
    @cart  = Cart.create
    @book_one = products(:ruby)
    @book_two  = products(:two)
  end

  test "add unique products" do
    @cart.add_product(@book_one).save!
    @cart.add_product(@book_two).save!
    assert_equal 2, @cart.line_items.size
    assert_equal @book_one.price + @book_two.price, @cart.total_price
  end
```

```
  test "add duplicate product" do
    @cart.add_product(@book_one).save!
    @cart.add_product(@book_one).save!
    assert_equal 2*@book_one.price, @cart.total_price
    assert_equal 1, @cart.line_items.size
    assert_equal 2, @cart.line_items[0].quantity
  end
end
```

The setup creates a cart and fetches two products from the fixtures. It then tests the add_product() in two ways: by adding two distinct products and by adding the same product twice.

Let's start with the setup. By default, RSpec is configured to look in spec/fixtures for fixtures. This is correct for a project using RSpec from the start, but for us, the fixtures are in test/fixtures. Change this by editing spec/rails_helper.rb:

rails7/depot_xa/spec/rails_helper.rb

```
RSpec.configure do |config|
  # Remove this line if you're not using ActiveRecord or ActiveRecord fixtures
➤   config.fixture_path = "#{::Rails.root}/test/fixtures"
```

Back to the spec—its setup will need to create a Cart to use in our tests as well as fetch two products from fixtures. By default, fixtures aren't available in specs, but you can call fixtures() to make them available. Here's what the setup looks like:

rails7/depot_xa/spec/models/cart_spec.rb

```
require 'rails_helper'

RSpec.describe Cart, type: :model do

➤   fixtures :products
➤   subject(:cart) { Cart.new }
➤
➤   let(:book_one) { products(:ruby) }
➤   let(:book_two) { products(:two) }
```

This definitely doesn't look like our original test! The call to subject() declares the variable cart, which you'll use in the tests later. The calls to let() declare other variables that can be used in the tests. The reason for two methods that seemingly do the same thing is an RSpec convention. The object that's the focus of the test is declared with subject(). Ancillary data needed for the test is declared with let().

The tests themselves will also look different from their equivalents in a standard Rails test. For one thing, they aren't called tests but rather *examples*. Also, it's customary for each example to make only one assertion. The existing test of adding different products makes two assertions, so in the spec, that means two examples.

Assertions look different in RSpec as well:

```
expect(actual_value).to eq(expected_value)
```

Applying this to the two assertions around adding distinct items, we have two examples (we'll show you where this code goes in a moment):

```
it "has two line items" do
  expect(cart.line_items.size).to eq(2)
end
it "has a total price of the two items' price" do
  expect(cart.total_price).to eq(book_one.price + book_two.price)
end
```

These assertions won't succeed unless items are added to the cart first. That code *could* go inside each example, but RSpec allows you to extract duplicate setup code into a block using before():

```
before do
  cart.add_product(book_one).save!
  cart.add_product(book_two).save!
end
it "has two line items" do
  expect(cart.line_items.size).to eq(2)
end
it "has a total price of the two items' price" do
  expect(cart.total_price).to eq(book_one.price + book_two.price)
end
```

This setup is only relevant to some of the tests of the add_product() method, specifically the tests around adding different items. To test adding the same item twice, you'll need different setups. To make this happen, wrap the above code in a block using context(). context() takes a string that describes the context we're creating and acts as a scope for before() blocks. It's also customary to wrap all examples of the behavior of a method inside a block given to describe().

Given all that, here's what the first half of your spec should look like:

rails7/depot_xa/spec/models/cart_spec.rb

```
describe "#add_product" do
  context "adding unique products" do
    before do
      cart.add_product(book_one).save!
      cart.add_product(book_two).save!
    end

    it "has two line items" do
      expect(cart.line_items.size).to eq(2)
    end
```

```
    it "has a total price of the two items' price" do
      expect(cart.total_price).to eq(book_one.price + book_two.price)
    end
  end
```

Here's the second half of the spec, which tests the behavior of add_product() when adding the same item twice:

rails7/depot_xa/spec/models/cart_spec.rb

```
require 'rails_helper'

RSpec.describe Cart, type: :model do

  fixtures :products
  subject(:cart) { Cart.new }

  let(:book_one) { products(:ruby) }
  let(:book_two) { products(:two) }

  describe "#add_product" do
    context "adding unique products" do
      before do
        cart.add_product(book_one).save!
        cart.add_product(book_two).save!
      end

      it "has two line items" do
        expect(cart.line_items.size).to eq(2)
      end
      it "has a total price of the two items' price" do
        expect(cart.total_price).to eq(book_one.price + book_two.price)
      end
    end

    context "adding duplicate products" do
      before do
        cart.add_product(book_one).save!
        cart.add_product(book_one).save!
      end

      it "has one line item" do
        expect(cart.line_items.size).to eq(1)
      end
      it "has a line item with a quantity of 2" do
        expect(cart.line_items.first.quantity).to eq(2)
      end
      it "has a total price of twice the product's price" do
        expect(cart.total_price).to eq(book_one.price * 2)
      end
    end
  end

end
```

Running bin/rails spec, it should pass:

```
> bin/rails spec
.....

Finished in 0.11007 seconds (files took 1.72 seconds to load)
5 examples, 0 failures
```

A lot of code in this file isn't executing a test, but all the calls to describe(), context(), and it() aren't for naught. Passing SPEC_OPTS="--format=doc" to the spec task, the test output is formatted like the documentation of the Cart class:

```
> bin/rails spec SPEC_OPTS="--format=doc"

Cart
  #add_product
    adding unique products
      has two line items
      has a total price of the two items' price
    adding duplicate products
      has one line item
      has a line item with a quantity of 2
      has a total price of twice the product's price

Finished in 0.14865 seconds (files took 1.76 seconds to load)
5 examples, 0 failures
```

Also note that rspec-rails changes the Rails generators to create empty spec files in spec/ instead of test files in test/. This means that you use all the generators and scaffolding you're used to in your normal workflow without having to worry about the wrong type of test file being created.

If all of this seems strange to you, you're not alone. It *is* strange, and the reasons RSpec is designed this way, as well as why you might want to use it, are nuanced and beyond the scope of this book. The main point all this proves is that you can replace a major part of Rails with an alternative and still get all the benefits of the rest of Rails. It's also worth noting that RSpec is popular, and you're very likely to see it in the wild.

Let's learn more about Rails' configurability by replacing another major piece of Rails—ERB templates.

Creating HTML Templates with Slim

Slim is a templating language that can replace ERB.[5] It's designed to require much less code to achieve the same results, and it does this by using a nested structure instead of HTML tags. Consider this ERB:

5. http://slim-lang.com

```
<h2><%= t('.title') %></h2>
<table>
  <%= render(cart.line_items) %>

  <tr class="total_line">
    <td colspan="2">Total</td>
    <td class="total_cell"><%= number_to_currency(cart.total_price) %></td>
  </tr>

</table>
```

In Slim, it would look like so:

```
h2
  = t('.title')
table
  = render(cart.line_items)

  tr.total_line
    td.colspan=2
      Total
    td.total_cell
      = number_to_currency(cart.total_price)
```

Slim treats each line as an opening HTML tag, and anything indented under that line will be rendered inside that tag. Helper methods and instance variables can be accessed using =, like so:

```
ul
  li = link_to @product.name, product_path(@product)
```

To execute logic, such as looping over a collection, use -, like so:

```
ul
  - @products.each do |product|
    li
      - if product.available?
        = link_to product.name, product_path(product)
      - else
        = "#{product.name} out of stock"
```

The code after - is executed as Ruby, but note that no `end` keyword is needed—Slim inserts that for you.

Slim allows you to specify HTML classes by following a tag with a . and the class name:

```
h1.title This title has the "title" class!
```

And, in a final bit of ultracompactness, if you want to create a `div` with an HTML class on it, you can omit `div` entirely. This creates a `div` with the class `login-form` that contains two text inputs:

```
.login-form
  input type=text name=username
  input type=password name=password
```

Putting all this together, let's install Slim and reimplement the home page in app/views/store/index.html.erb using it. This will demonstrate how Rails allows us to completely replace its templating engine.

First, install slim-rails by adding it to the Gemfile:

```
gem 'slim-rails'
```

After you bundle install, your Rails app will now render files ending in .slim as a Slim template. We can see this by removing app/views/store/index.html.erb and creating app/views/stores/index.slim like so:

rails7/depot_xb/app/views/store/index.slim

```
- if notice
  aside#notice = notice

h1 = t('.title_html')

ul.catalog
  - cache @products do
    - @products.each do |product|
      - cache product do
        li
          = image_tag(product.image_url)
          h2 = product.title
          p = sanitize(product.description)
          .price
            = number_to_currency(product.price)
            = button_to t('.add_html'),
                line_items_path(product_id: product, locale: I18n.locale),
                remote: true
```

Restart your server if you have it running, and you should see the home page render the same as before.

In addition to being able to render Slim, installing slim-rails changes Rails generators to create Slim files instead of ERB, so all of the scaffolding and other generators you're used to will now produce Slim templates automatically. You can even convert your existing ERB templates to Slim by using the erb2slim command, available by installing the html2slim RubyGem.[6]

6. https://github.com/slim-template/html2slim

Customizing Rails in Other Ways

Customizing the edges of Rails, like you did in the preceding section with CSS, HTML templates, and tests, tends to be more straightforward, and more options are out there for you. Customizing Rails' internals is more difficult. If you want, you can remove Active Record entirely and use libraries like Sequel or ROM,[7,8] but you'd be giving up a lot—Active Record is tightly coupled with many parts of Rails.

Tight coupling is usually viewed as a problem, but it's this coupling that allows you to be so productive using Rails. The more you change your Rails app into a loosely coupled assembly of unrelated libraries, the more work you have to do getting the pieces to talk to each other. Finding the right balance is up to you, your team, or your project.

The Rails ecosystem is also filled with plugins and enhancements to address common needs that aren't common enough to be added to Rails itself. For example, Kaminari provides pagination for when you need to let a user browse hundreds or thousands of records.[9] Ransack and Searchkick provide advanced ways of searching your database with Active Record.[10,11] CarrierWave makes uploading files to your Rails app much more straightforward than hand-rolling it yourself.[12]

And if you want to analyze and improve the code inside your Rails app, RuboCop can check that you're using a consistent style,[13] while Brakeman can check for common security vulnerabilities.[14]

These extras are the tip of the iceberg. The community of extensions and plugins for Rails is yet another benefit to building your next web application with Rails.

Where to Go from Here

Congratulations! We've covered a lot of ground together.

7. http://sequel.jeremyevans.net/
8. http://rom-rb.org/
9. https://github.com/kaminari/kaminari
10. https://github.com/activerecord-hackery/ransack
11. https://github.com/ankane/searchkick
12. https://github.com/carrierwaveuploader/carrierwave
13. https://github.com/bbatsov/rubocop
14. https://github.com/presidentbeef/brakeman

In Part I, you installed Rails, verified the installation using a basic application, got exposed to the architecture of Rails, and got acquainted (or maybe reacquainted) with the Ruby language.

In Part II, you iteratively built an application and built up test cases along the way. We designed this application to touch on all aspects of Rails that every developer needs to be aware of.

Whereas Parts I and II of this book each served a single purpose, Part III served a dual role.

For some of you, Part III methodically filled in the gaps and covered enough for you to get real work done. For others, these will be the first steps of a much longer journey.

For most of you, the real value is a bit of both. A firm foundation is required for you to be able to explore further. And that's why we started this part with a chapter that not only covered the convention and configuration of Rails but also covered the generation of documentation.

Then we proceeded to devote a chapter each to the model, view, and controller, which are the backbone of the Rails architecture. We covered topics ranging from database relationships to the REST architecture to HTML forms and helpers.

We covered migration as an essential maintenance tool for the deployed application's database.

Finally, we split Rails apart and explored the concept of gems from a number of perspectives, from making use of individual Rails components separately to making full use of the foundation upon which Rails is built and, finally, to building and extending the framework to suit your needs.

At this point, you have the necessary context and background to more deeply explore whatever areas suit your fancy or are needed to solve that vexing problem you face. We recommend you start by visiting the Ruby on Rails site and exploring each of the links across the top of that page.[15] Some of this will be quick refreshers of materials presented in this book, but you'll also find plenty of links to current information on how to report problems, learn more, and keep up-to-date.

Additionally, please continue to contribute to the forums mentioned in the book's introduction.

15. http://rubyonrails.org/

Pragmatic Bookshelf has more books on Ruby and Rails subjects as well as plenty of related categories that go beyond Ruby and Rails, such as technical practices; testing, design, and cloud computing; and tools, frameworks, and languages.

You can find these and many other categories at http://www.pragprog.com/.

We hope you've enjoyed learning about Ruby on Rails as much as we've enjoyed writing this book!