Build A SaaS App In Ruby on Rails 7

Learn to build a production-ready Ruby on Rail 7 application



Rob Race

Build A SaaS App in Rails 7

Rob Race

This book is for sale at http://leanpub.com/buildasaasappinrails7

This version was published on 2023-09-15



This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2022 - 2023 Rob Race

Contents

1. Ruby on Rails and your First Application 1
- 1.1 Prerequisites 1
- 1.2 Rails Overview 2
  - 1.2.1 Rails 2
  - 1.2.2 Gems 3
- 1.3 A Sample App to Get You Started 4
  - 1.3.1 Rails Gem 4
  - 1.3.2 Installing Rails and Starting a New App 5
  - 1.3.3 Directory Structure 6
  - 1.3.4 Rails Server 7
  - 1.3.5 Rails Conventions 10
  - 1.3.6 Hello MVC 11
- 1.4 What did you learn in this chapter? 13
- 1.5 Exercises 14

2. Testing and You 15
- 2.1 Why Test 15
- 2.2 Testing Frameworks 16
- 2.3 Testing Types 18
  - 2.3.1 Model/Unit Tests 18
  - 2.3.2 View Tests 22
  - 2.3.3 System Tests 23
- 2.4 Other Testing Tools 24
- 2.5 CI and You 25
- 2.6 What did you learn in this chapter? 26

2.7 Exercises 27

3. Starting your SaaS App 28
3.1 `rails new`...for real this time 28
3.1.1 Install PG 28
3.1.2 `rails new` with pg 29
3.1.3 Database Configuration 29
3.2 Git gud enuf 30
3.2.1 Git Commands 31
3.2.2 Git Usage 32
3.3 Gemfile and preferred gems to start 34
3.4 Using TailwindCSS 40
3.4.1 First Controller, Views, and Tests 42
3.4.2 Application layout 46
3.4.3 Turbo 54
3.4.4 One more thing... 54
3.5 Exercises 55

4. Users are Everything 56
4.1 User Model 57
4.1.1 But first, UUID 58
4.1.2 Creating a Users Table 59
4.1.3 Installing Devise 63
4.2 Sign in, out, and up 68
4.3 User Management 102
4.3.1 User Invites 103
4.3.2 Managing the Users 105
4.3.3 Testing User Management 128
4.3.4 CanCanCan 146
4.4 What did you learn in this chapter? 157
4.5 Exercises 158

5. Business Models and Logic 159
5.0.1 `rails g scaffold` 161
5.0.2 Model 163
5.0.3 Controller 163

5.0.4 Views 166
5.1 Creating The Remaining Business Logic 172
5.1.1 Current Date 172
5.1.2 Teams using Has Many Through 186
5.1.3 Navigating better navigation 228
5.1.4 Tasks with Single Table Inheritance . . 235
5.2 What did you learn in this chapter? 283
5.3 Exercises 283

6. Service Objects 285
6.1 What is a Service Object? 285
6.1.1 So...a service Object? 286
6.2 Building and using a service object in this app . . 292
6.3 More uses for Service Objects 299
6.4 What did you learn in this chapter? 307
6.5 Exercise 307

7. Background Jobs, ActiveJob and Sidekiq 308
7.1 What is a Background Job, and why use one? . . 308
7.2 Right Tools for the "background" job 309
7.2.1 ActiveJob 310
7.2.2 Sidekiq 314
7.2.3 Sneakers 314
7.2.4 SuckerPunch 315
7.2.5 Other Notable Mentions 315
7.3 Using SideKiq with ActiveJob 316
7.4 Other ActiveJob tips and tricks 317
7.5 Scheduled Jobs 320
7.6 What did you learn in this chapter? 331
7.7 Exercises 332

8. Mailers and You 333
8.1 What is a Mailer? 333
8.2 Using a Mailer in your application 335
8.2.1 Adding a Welcome Email 335
8.2.2 Email Recap 345

8.2.3 Mail in Production 356
8.3 Receiving mail or replies 357
8.4 What did you learn in this chapter? 373
8.5 Exercises 374

9. HotWire 375
9.1 What is HotWire? 375
9.1.1 What is Turbo? 375
9.2 Using HotWire in your application 376
9.3 Streaming Updates 376
9.3.1 Streaming Standup Create 378
9.3.2 Streaming Standup Update 383
9.3.3 Streaming Standup Removal 385
9.3.4 Streaming Standup Changes onto Team pages 388
9.4 Turbo Frames 417
9.5 What did you learn in this chapter? 433

10. Stripe Payments and Subscriptions 434
10.1 Stripe 434
10.2 Stripe Gem 435
10.3 Products 437
10.4 New Account; New Subscription 443
10.4.1 Some Subscription Specs 456
10.5 Stripe's Billing Portal 461
10.5.1 Stripe Events 463
10.6 Checking for an Active Subscription 470
10.7 Limits 476
10.8 What did you learn in this chapter? 483
10.9 Exercises 483

11. Using Third-Party APIs in your Application 484
11.1 Github Integration 485
11.1.1 Github Gems 485
11.1.2 Implementing Account and Github API 486
11.1.3 Granting Access to Github via OAuth . 487

11.1.4 Fetching Data through Github 500
11.1.5 Accessing and Handling Github Webhooks 522
11.1.6 Displaying Event Data 535
11.1.7 Testing the Github Integration 546
11.2 Analytics: What are they good for? 576
11.2.1 Choices 577
11.2.2 Implementing and Sending events . . . 578
11.3 What did you learn in this chapter? 585
11.4 Exercises 585

12. Speeding up Rails 587
12.1 What Slows Rails Down? 587
12.1.1 N + 1 Queries 589
12.1.2 Queries without indices 591
12.1.3 Pagination 592
12.1.4 Gzip 601
12.1.5 Caching 602
12.1.6 How to Implement Caching 604
12.2 What did you learn in this chapter? 610

13. Deployments 612
13.1 There are many choices 612
13.2 Prepare application 612
13.3 Deploy with Heroku 614

14. A Few Other Rails Tools 617
14.1 Async Queries 617
14.2 Active Record Encryption 619

1. Ruby on Rails and your First Application

In this chapter, we will go over a little bit of the history of Rails and its benefit to you and some of the Rails basics. Ruby on Rails is a web framework that had its humble beginning in 2004. In an upcoming chapter, you will build your example SaaS app, an Agile/Scrum Daily Standup Tracker. You should be able to deviate from the example app at any time with your business idea. This book attempts to present concepts in a way that transcends any business logic or differentiations you may have.

1.1 Prerequisites

Knowing a few of the following concepts will make the best use of this book. You may be able to work your way through the book without all or any of them. Though you may benefit from having an understanding of the following

- **HTML and CSS**: You should have a basic understanding of HTML and CSS, and we are using the Bootstrap HTML/CSS framework. Bootstrap abstracts away the painful parts of HTML/CSS/layouts. Any knowledge of HTML/CSS will help you navigate your way through changes or any customization you would like to do.
- **SQL**: A basic understanding of databases and SQL will help you understand database interactions in Rails, including how you are querying the database or database abstractions.

- **Ruby**: This book will walk you through building an app with Rails. A basic understanding of Ruby (or another programming language) will help you get the most use of all the chapters in this book. In addition, you can grasp the examples throughout the book if you understand variables, conditionals, loops, and methods within the Ruby language.
- **Linux**: Having a Mac or Linux machine is the most accessible path to installing Ruby, Rails, and other packages used in developing an app. While it is possible to develop Rails on a Windows machine, it is not the most documented or utilized operating system for developers. Thus, using Windows means you may experience more setup issues and less help in the form of recorded fixes across the internet. However, I'll point out some excellent Windows-based variations via third-party blog posts as the need arises.

1.2 Rails Overview

1.2.1 Rails

David Heinemeier Hansson created Rails as he intended to create a simple *Convention over Configuration Framework*. Many web frameworks in the early 2000s would have extensive systems of configuration and needed many files to get your web applications up and running. David, finding this needless, designed Rails to have deep-rooted conventions making setting up and changing Rails apps a breeze.

Ruby on Rails is open source, meaning it is accessible to end-users and maintained by contributors worldwide. Powering Rails is the Ruby programming language. It is considered by many as one of the most pleasurable languages to program in, with its simplistic, expressive nature and its modularity.

Most times, you'll be wondering how it can be so easy — adding another resource from the database, through RESTful routes and templated views, for example. New feature additions to Rails have slowed down in recent years. This pace lends itself to Rails' now stable nature. The Rails core team is still keeping pace with other web frameworks and languages in areas such as WebSockets and javascript package management.

When you are stuck in Rails, you will be hard-pressed not to find an answer or support channel due to its longevity and large community. You'll often find an answer in the vast database of Questions and Answers on Stack Overflow. Otherwise, you can browse issue tracking in the Rails Github repositories. Lastly, you can find in-depth help with over ten years of blog posts and tutorials on copious blogging platforms or plenty of books! Most times, you can be on your way to solving your issue within seconds just by Googling the problem or exception message.

1.2.2 Gems

Gems are Ruby's self-contained packages of code. Gems are solving or collapsing logical programmatic problems. For instance, a gem called Devise helps create a near end-to-end authentication system. Another gem, Puma, provides you with a threaded HTTP server to serve your web application. Puma provides a local server as you develop and on a server in a production environment. There are thousands upon thousands of gems out there! Most gems are designed to help with most problems you encounter and have been vetted and used by millions in the open-source community.

1.3 A Sample App to Get You Started

1.3.1 Rails Gem

By default, not all systems have Rails installed. Most times, if a system does have a system-level version of Ruby, it is older than you would like for current development. Let's go over how you can install a recent version of Ruby, manage multiple Rails versions, and group gems to have a cleaner separation of gem versions per project.

The instructions for this tutorial address development on OSX and should work on almost any Unix-based system(OSX, Linux, etc.). I am deferring to a Rails 5 install blog post for Windows (*http://blog.teamtreehouse.com/installing-rails-5-windows*[1])

You can use many different tools to manage ruby versions, such as RVM, RBenv, or Chruby. I am still a fan of RVM after all these years. I find it easy for beginners to grok and simple to maneuver once you get it.

I recommend visiting the RMV website (*https://rvm.io/*[2]). Upon the browser rendering the RVM website, you find two simple bash commands you can use to install RVM. I opted not to print the commands here as 1) a digital security measure and 2) they may change as time goes on.

Once installed, we can begin installing the latest version of Ruby.

```
rvm list known
```

This command lists the current versions of Ruby available to install. There may be a lot on the list, but we can focus on the primary interpreter MRI plus the latest version(3.0.x as of writing). Now, the command to install:

[1] http://blog.teamtreehouse.com/installing-rails-5-windows
[2] https://rvm.io/

```
rvm install 3.0.4
```

Now sit back or grab a coffee as the installation can take anywhere from 1 to many minutes to complete. This wait is based on Ruby package availability, system speed, and many other factors.

Once the gems are installed, you can now set up a `gemset`. A gemset is a way to group installed gems into a particular RVM-based folder. Gemsets keep different projects and their various versions of gems from interfering with each other. Although with Bundler, this isn't a huge issue, it doesn't hurt to take the extra step and keep your gems tidy. Additionally, this command instructs RVM to use the recently installed version of Ruby at the same time:

```
rvm use 3.0.4@sample_app --create
```

The extra `--create` flag tells RVM to create the gemset if it doesn't exist(which it doesn't).

Now that we finally have our Ruby and Gemsets in order, we can install Rails. We are starting a new sample project to show you the basics of a new Rails app. Later we rerun this command for the SaaS app's creation. Please refer to the RVM documentation if you want to dig deeper into any other features of RVM past the basics in this chapter.

1.3.2 Installing Rails and Starting a New App

Now that we have Ruby installed and the gemset ready to go from the last portion of this chapter, we can now install Rails. We are installing version 7.0.2 and continue to use this version to keep results consistent throughout the book.

It may not be all smooth sailing when installing Ruby or specific Ruby gems. For instance, when trying to install rails on ruby 3.0 on my system from a clean slate, I ran into OpenSSL issues. I also ran

into a problem with libraries needed for a gem called Nokogiri(used for HTML parsing in rails). It is somewhat impossible to predict issues you may see during installation with many Ruby versions, Rails versions, operating systems, and particular versions. Thus, I recommend you Google the error + the operating system + operating system version. You will usually have an answer within a couple of results and, most times, the first StackOverflow link you find.

Anyway, on with the installation:

```
gem install rails -v 7.0.2
```

Assuming your install finishes without error or after you solve all install issues, you can run the following command to create a new application.

```
rails new sample_app
```

After this command completes, your rails app is installed. Rails first installs the needed default gems, default commands, and file watchers. Also, note that the `rails new` command has many more flags and options, which you can read by typing the following.

```
rails new --help
```

1.3.3 Directory Structure

Rails come with a default directory structure to auto-load many files and folders. This organization of files may seem magical, but it is Rails' strong convention over configuration at work. This structure makes it easy to get started with a base application. In fact, after going over the directory structure, let's fire up the Rails application you created. Then even make a minor modification to get your own "Hello World" response!

Anyway, onwards and upwards. Let's go over the directories!

Folder	Use
app/	This is the primary parent folder of your application.
app/assets	App-specific stylesheets and images
app/channels	Files to handle WebSocket channel negotiations
app/controllers	Rails controller files
app/helpers	Helper methods for controllers and more often view files
app/jobs	Files for your background jobs
app/javascript	Where your app-specific javascript goes
app/mailers	This folder will hold mailer setup files
app/models	Model files that represent table database rows
app/views	This is where your templates and markup will live
bin/	This is where some rails command files exist
config/	This will house configuration files
db/	Database and database related files
lib/	Library module files that are not part of the app itself
log/	Rails creates log files, and they are in this folder
public/	Here, you can put static files such as favicons
test/(or spec/)	All your test folders and files
vendor/	Here, you can put third party javascript, stylesheets, or fonts

1.3.4 Rails Server

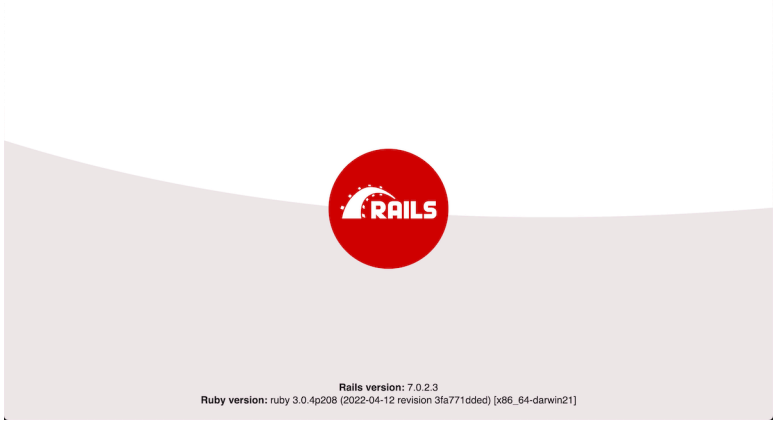
You can start a brand new project's application immediately with no further changes needed. This functionality is provided in many frameworks as a way to show you, yes, your install worked!

As a note, staying with an assumption of OSX in this book, you will use the Unix-based `cd` command to change directories. This

command should work in many other shells on Windows and virtually all Linux systems.

```
cd sample_app
bundle install
rails server
```

That command moves you into the applications directory and fires up Rails' development server on port 3000. Go ahead and go to *http://localhost:3000*[3], and you should see a welcome page



Let's not stop there, however! What is any programming book without a "Hello World" example? We will be changing two files to make this work. I will paste in the entirety of these two files, and you can use your text editor of choice(Sublime Text, Atom, vim, etc.).

[3] http://localhost:3000

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

  def awesome_action
    render html: "<h1>Hello, world!</h1>".html_safe
  end

end
```

The addition to the file is the method 'awesome_action'. When a request comes in the app that is directed to the awesome_action in the ApplicationController, it renders an HTML string that says "Hello, world!". The additional method at the end of the file ensures we pass the string through to the response with escaped HTML instead of the literal string. It is thus allowing the browser to display the text as a heading.

As a small note, actions are very rarely assigned or defined in the ApplicationController file. Instead, this controller is used for methods or functionality that extends to most other controllers. I will explain and show this in future chapters. So, how do we tell the Rails app to direct the request to the root address of our new awesome_action? Well, we edit the routes file:

```
Rails.application.routes.draw do

  root to: 'application#awesome_action'

end
```

The routes file handles directing incoming HTTPS requests. They are directed to the responsible controller and action based on path, content type, and HTTP method type. Further, in the book, you will see more Rails conventions to make this quick and painless for the usual REST routes and actions.

The router uses the `controller_name#action` with multiple methods and parameters. Setting the root route is no different. Here the route says, "set the root path to the ApplicationController and the awesome_action action". Then, using the previously explained controller code tells the browser to render our HTML message! So, if you head to *http://localhost:3000*[4] again, you should see our new message to the world!

1.3.5 Rails Conventions

Rails follows an MVC convention. This acronym stands for Model-View-Controller. A model is a Ruby class that interfaces with the database processes validations and can hold some extra business logic. A view is the markup file or template in Rails. The View is typically an HTML templating engine but can be other files such as javascript files with embedded Ruby code or JSON files. Last is the controller. The controller, as described earlier, holds the actions as an HTTP request routes through the application. Controllers are unique because they almost always interface with both a Model and a View in a Controller's action.

Models:
Models interface with the database through the ActiveRecord ORM(Object Relational Mapping). For example, you would have a Model class such as User, which represents the mapping between the databases' users table and the objects in Rails. Models also have a built-in validation framework where you can validate uniqueness, presence, regular expressions, and much more.

Each Model object represents a record in the database, for example.

[4] http://localhost:3000

```
User.find(42) #=> {id: 42, name: "John"}

User.create({name: "James"})
```

Rails models using ActiveRecord also have a ton of convenience through a convention. Such as referenced associations (belongs_to, has_many, etc.) and callbacks(before_create, after_save, etc.).

Controllers:
Controllers are vital but straightforward. As a request comes through the router, it is routed, and an action method inside a controller glues the request to the model (database), View (markup), and sometimes other business logic (background tasks, services, etc.).

Controllers are also responsible for handling abstract but needed logic such as sessions, authorizations, errors, or redirects. One important thing to note is the use of instance variables in actions. This detail allows values set in the controller to be accessed within the View. Meaning a database record looked up through a model instance gets saved to an instance variable that is accessible in the View.

View:
Views are pretty straightforward. Nearly every time you need to deal with a view, it is an HTML-based template. By default, Rails uses the ERB format for representing markup with embedded Ruby.

```
<h1><%= @user.name %></h1>
```

Yet, the view file can be other types of markup/template, such as Javascript, CSS, JSON, XML, CSV, and more!

1.3.6 Hello MVC

Let's combine the last two sections into a very MVC like `Hello World`. We will need a Model, a new Controller, and a View. We

can start with the model. For simplicity, we will create a plain old Ruby class that we can instantiate with a name attribute.

```
class User
  attr_reader :name
  def initialize(name='Rob')
    @name = name
  end
end
```

If you are unfamiliar with Ruby classes, `initialize` is called whenever a class instantiates a new object (e.g., `User.new`). While `attr_reader` (and the later `@name = name`) sets up a getter method (meaning you can use `user.name` to get the `name` value) and `name='Rob'` to set a default value. You can read a longer overview of Ruby classes here[5].

Next, we can create the new controller. Again, this looks very close to the sample action we added earlier but in a new file this time.

```
class StaticController < ApplicationController

  def hello_mvc
    @user = User.new
  end
end
```

The only new syntax here is the `@user = User.new` line that creates an object used in the View we are about to make. You may wonder why there is an `@` in front of the variable. In Rails, an instance variable (one prefixed with `@`) is available throughout the request and added to the View as a controller's action renders. Thus, we will be able to use `@user.name` in the View.

[5] http://rubylearning.com/satishtalim/writing_our_own_class_in_ruby.html

As mentioned before, Rails uses ERB for the View templating. It is straightforward to use by starting with standard HTML and adding Ruby. Here is the view of our sample app.

```
<h1>
  Hello <%= @user.name %>
</h1>
```

The last part of tying all of this together is to update the `routes.rb` file to use the new MVC code path we just created!

```
Rails.application.routes.draw do
  root to: 'static#hello_mvc'
end
```

If you still do not have your Rails server running, head back to your terminal, type `rails server`, and go to your localhost address. You should see `Hello Rob` in an `H1` tag. Since the model object takes a name parameter, you can change the line in the controller to any name you would like, such as `@user = User.new('Bob')`, reload the page in your browser, and see the new name.

Hopefully, this little example we just made showcases how easy it is to build with Rails. It gets even more comfortable as we go through the book and see how many Rails conventions make the process even quicker.

1.4 What did you learn in this chapter?

- You learned about the origins of Ruby on Rails and its intentions.
- You learned what gems are and why they are used.

- You created a sample app and then modified it to output a message you added.
- You learned about the basic conventions of a Rails app.

1.5 Exercises

You can find exercise answers with any package or bundle purchases of Build A SaaS App in Ruby on Rails 7[6]!

1. What command can you use to get `help` for all Rails CLI commands? When using help commands, what `rails server` flag would you use to change the port?
2. Awesome_action is not a great or descriptive name for the action. How would you update the action name and route?

[6] https://buildasaasappinrails.com

2. Testing and You

While requiring a little bit more cognitive load upfront and even loathed by some developers, testing often pays dividends. It will almost always have you saying, "I'm glad I wrote tests for this." at some point. TDD (Test Driven Development) is one approach that has grown in the Rails community. Although the community has not entirely accepted this approach, this method has you write your test(s) first and then write code to make your tests pass.

2.1 Why Test

Testing provides a few main benefits when done correctly. First, it helps you develop the behaviors within your app. For example, you may write a test in which your model validates the presence of an email. Then you realize you need to include validations for the password or add a "retype password" input box.

Another great reason to test is to protect yourself against regressions within your application as you're modifying code or adding new features. Testing also gives you confidence and assurance when refactoring classes, collapsing code, or removing extraneous code. If written tests pass before and after changes, you should be good to go! Another testing method is to write a test after finding an issue or bug in your application. Ensuring the fix works and does not plague your application in the future.

2.2 Testing Frameworks

Like many other parts of Rails, there are default gems and libraries and other optional alternatives, which at times are more used than the defaults.

RSpec vs. MiniTest

As of the latest version of Rails, MiniTest is the default testing framework included, which is an improvement on the original testing framework for Rails, Test::Unit. Personal preference can play a huge part in choosing many aspects of a Rails application, and testing is no exception.

The most substantial alternative to MiniTest is RSpec, which focuses on the readability and composability of tests. It uses words such as "describes", "context", "expect" and "it" when naming tests and test groups. For example:

```
describe "the test" do
  it "is a success" do
    expect(1).to eq(1)
  end
end
```

Instead, MiniTest uses the assert terminology (i.e., assert this equals that) found in other programmatic testing frameworks. I like RSpec. I find it easier to reason about, and it has a better command-line interface(CLI). Therefore, RSpec will be the testing framework of choice for this book.

Capybara System Tests vs. Integration Tests

Another unique alternative in the testing work is the addition of Capybara instead of using built-in Integration tests to test workflow. The built-in integration tests leave most developers desiring more for real-world browser workflows. This situation is

where Capybara comes in. Thus, allowing you to write feature tests taking you through pages of your choice within your application. Performing behaviors such as clicking, filling in a form or verifying that everything is behaving as expected.

Capybara plugs right into RSpec (and now is bundled into Rails System tests), allowing you to use the same syntax and phrasing to write tests. Here is an example directly from their docs!

```
describe "the sign-in process", type: :feature do
  before :each do
    User.make(email: 'user@example.com', password: 'passw\
ord')
  end

  it "signs me in" do
    visit '/sessions/new'
    within("#session") do
      fill_in 'Email', with: 'user@example.com'
      fill_in 'Password', with: 'password'
    end
  click_button 'Sign in'
  expect(page).to have_content 'Success'
  end
end
```

This book will use Capybara through RSpec in this book for the reasons mentioned above.

FactoryBot

The last alternative I'll mention is the gem FactoryBot. The default method of test objects within Rails is fixtures. Fixture files are written in the YAML format, providing a necessary static mock object representation. Though fixtures get more burdensome, the model objects they represent get more complicated. Including their associations, they are proliferating. FactoryBot describes itself as

an API to create test objects. You programmatically use the API to create mock objects for your tests. You can provide a factory definition in Ruby, then use the API much like standard objects to `.build` or `.create`. FactoryBot also makes it super simple to include associations through the factory definitions. Thus, we will be using FactoryBot in this book.

2.3 Testing Types

There are three main types of tests in a typical Rails application: Model (Unit) tests, Controller tests, and View tests. Additionally, you can also use System Tests. These tests function as an end-to-end walkthrough test of your application. Each type of test has its place in the testing world and maybe equally (mostly) significant.

2.3.1 Model/Unit Tests

Model specs are most likely the simplest test types you deal with in Rails. With these tests, you are testing the regular CRUD (Create/Read/Update/Delete) operations on the model object. Additionally, they test validations and associations. As a result, model specs run super quickly and are relatively self-contained.

```
FactoryBot.define do
  factory :user do
    name { "Same Name" }
    email { "test@user.com" }
  end
end
```

```ruby
class User < ApplicationRecord

  validates :email, presence: true
  validates :email, uniqueness: true
  validates :name, presence: true

  ...
```

“‘ruby
describe User do

it “has a valid factory” do

expect(FactoryBot.build(:user).save).to be_true
end

it “is invalid without a name” do

expect(FactoryBot.build(:user, name: nil).save).to be_false
end

it “is invalid without a unique name” do

user = FactoryBot.create(:user)
expect(FactoryBot.build(:user, name: “Same Name”)).to be_false
end

it “is invalid without an email” do

expect(FactoryBot.build(:user, email: nil)).to be_false
end

end

```
For this example, there is an already valid user factory \
definition that is setting some default values. Additiona\
lly, this scenario tests that the User model can create a\
 user. Finally, the spec file validates user creation fai\
ls on a few key Rails Model validators on this User model\
.

Unit test is the general definition for testing one unit \
of work. In the case of Ruby, this is usually a class, wh\
ether it is a Service Object or a plain ole class, to pro\
vide some other functionality. An easy way to look at the\
 difference between Model and Unit tests is that every mo\
del test is a unit test, but not every unit test is a mod\
el.

### Controller Tests

Controller tests provide the functional approach to ensur\
ing all your conditional logic with tests based on reques\
t type, user type, success/failure of Models, and any oth\
er split logic. Controller specs are slightly slower than\
 Model specs due to their use of the request/response cyc\
le, though controller specs are still considerably faster\
 than feature specs. Thus, this is a great place to make \
sure the request/response cycle for a particular web requ\
est is "functioning" as it should!

"`ruby
describe UsersController do

  describe "GET index" do
    it "assigns @users" do
      user = FactoryBot.create(:user)
      get :index
      expect(assigns(:users)).to eq([user])
```

```
    end

    it "renders the index template" do
      get :index
      expect(response).to render_template("index")
    end

  end

  describe "GET #show" do
    it "renders the #show view" do
      get :show, {id:user.id}
      expect(response).to render_template("show")
      end
    end
  end

end
```

In this test file, you find that we are testing a page that would list the users. First, testing the action assigns the correct output. After creating a user and getting the `:index` action, the controller has an instance variable that equals an array of the one user from the test. The second test only tests that getting the `:index` action renders the index template.

These tests are not as involved and are abbreviated to give you a taste of how easy controller testing can be. For example, in a real-world Controller file, you would test every action, its behavior and rendering, and each branch of the action, such as the success or failure of a database transaction.

2.3.2 View Tests

View specs are often overlooked by developers who test all aspects of the visual representation in feature tests. System tests are meant to give you high-level behavior testing. However, they definitely should not be used to test every last detail of a server-rendered template. Instead, you often need to test different markup based on some if statement or logic that a view spec can do.

“‘ruby
FactoryBot.define do
factory :user do

name “Same Name”
email “*test@user.com*[1]” admin false
end
end

```erb
  <%- if @user.admin? %>

  <h1>Welcome back admin</h1>

  <%- else %>

  <h1>Welcome back pal</h1>

  <%- end %>
```

[1] mailto:test@user.com

```
describe "rendering homepage" do
  it "displays admin message" do
    assign(:user, FactoryBot.create(:user, admin: true))
    render :template => "home/index.html.erb"
    expect(rendered).to match /admin/
  end

  it "displays regular user message" do
    assign(:user, FactoryBot.create(:user))
    render :template => "home/index.html.erb"
    expect(rendered).to match /pal/
  end

end
```

In this example, we have a simple view template that renders a message in the markup. The noun is based on whether the user referenced in the user variable is an admin or not. Here, the view spec renders out the message with the user object created in each test (one admin, one not) and tests for the text with regular expressions.

View specs can sometimes be challenging to set up and test as abstractions are made within the application and view logic. Usually, view specs are few and far between but again can be important when there is a simple view with a bunch of simple branching logic.

2.3.3 System Tests

Finally, system tests (or may also be known as feature tests) are an excellent way to test your entire application at once(or significant parts of behavior). Using RSpec and Capybara, you can write simple everyday tests that walk through your app as if you were an end-user.

```
describe "Signing Up", :type => :feature do
  it "allows a user to sign up" do
    visit root_path
    expect(page).to have_content 'Sign Up'
    within("form#user_new") do
      fill_in 'Name', with: 'John'
      fill_in 'Email', with: 'user@example.com'
      fill_in 'Password', with: 'password'
      fill_in 'Retype Password', with: 'password'
    end
    click_button 'Sign Up'
    expect(page).to have_content 'Welcome to Sample App'
    expect(page).to have_content 'John'
  end
end
```

This test simulates a user visiting your application for the first time and then seeing a sign-up page, filling it out, and successfully accessing it after sign-up.

As you may know, System tests provide a fantastic way to test at a high level, just as a user would be interacting with your application. However, with that comes the cost of the fact feature tests run slower than all other types of tests. For that reason, it would be wise not to test every branch or possibility in your application and use a controller spec or other specs to test edge cases.

2.4 Other Testing Tools

Additionally, there are a few extra tools I would like to go over and will be included in your future application's codebase. These tools help with issues such as a test coverage completeness tool which consists of a local web tool to see analysis file by file and an instrument to mock external resources, such as third-party APIs.

Simplecov: We use this gem to calculate, display and investigate code coverage. This tool uses built-in Ruby coverage information and merges the results of different tools and test types of coverage. Afterward, it generates the data, saves it to a report, and automatically creates an HTML site for you to visit to look through the code coverage analysis.

VCR: VCR allows you to "record" HTTP transactions and save their responses locally. This tool lets you virtually "playback" the response from a file, making it easy and fast to run tests that interact with external resources. Along with the speed increases from no longer making an external request, it allows for offline testing and not worrying about a third-party change affecting your tests. However, it is recommended that you still perform some testing on the actual external resource's response.

Webmock: Webmock sits a layer higher than VCR and functions as the system to intercept the requests and return a mocked request. VCR will typically use Webmock "under the hood", to power the "cassette" based approach.

2.5 CI and You

The last thing to discuss in this chapter is Continuous Integration. You continuously merge changes into the main branch, where various pipelines work on the newly merged code. In most cases, this is a way to merge changes into one central branch. Then run the tests on an external system and use the test results to choose another action, such as deploying or requesting a fix to the codebase.

There are a few different tools one could use for CI. The first that may be recommended is *Jenkins*[2]. The great thing about Jenkins is that it is an open-source and free project. However, there is no hosted solution available through their site. This limitation means

[2] http://jenkins.io/

you will either have to build and maintain your Jenkins server or seek a service that would do so for you.

There is good news for those who do not want to keep their server running continuous integration. Quite a few services run CI services and have integrations with all major repository services(GitHub, Bitbucket, etc.). The integrations allow a simple push to your repository that trigger your tests to run automatically on their service. Followed by a report with the results in any response medium you would like. Such as email, chat services, and others.

Without any affiliation, the server I usually use for my applications is *CircleCI*[3]. I find their ease of use and their pricing (free for most of my purposes) to suffice for my needs well. Other choices would be *TravisCI*[4] and *SemaphoreCI*[5]. Again, I would recommend checking these products out and choosing the one that fits you best.

2.6 What did you learn in this chapter?

- You learned about the importance of testing your Rails application.
- You learned the alternatives for testing frameworks.
- You learned about the different types of tests and examples of each test type.
- You learned about additional test tools.
- You learned about Continuous Integration, where it fits into your testing workflow and services to run the CI.

[3] http://circleci.com/
[4] https://travis-ci.org/
[5] https://semaphoreci.com/

2.7 Exercises

You can find exercise answers with any package or bundle purchases of Build A SaaS App in Ruby on Rails 7[6]!

1. Suppose your main dashboard page showed different charts for each of the four roles your application allows a user to belong to. What type of tests could you use to test that the correct charts showed up on the page?
2. Since there is test type overlap, what would the preferred test type be here and why?

[6] https://buildasaasappinrails.com

3. Starting your SaaS App

The time has come! We begin building a SaaS application right now. You are creating an Agile/Scrum Daily Standup Tracking application for this book. This sort of application will allow you to focus on the basics and some in-depth topics throughout the book. First, you will start a new Rails application. Then continue to build upon the application until you deploy the application live!

You have two approaches to building an application from this book. First, you can make the same application as the book, step by step. Then, use that knowledge to create another application with your business concept and logic. Otherwise, you can deviate immediately with your business idea and use the same techniques.

3.1 `rails new`...for real this time

3.1.1 Install PG

When building a production-ready application with Rails, the most popular database to use is PostgreSQL. However, Rails via ActiveRecord supports a few databases out of the box. These databases include PostgreSQL, MySQL, and SQLite3. In addition, Rails can support other databases through gems and their ORM.

Since you are most likely using both PostgreSQL and MacOS(OSX), I recommend just installing Postgres.app. Head to the postgres.app[1] site. Then download and follow the simple instructions to have a PostgreSQL server and library installed on your box.

[1] https://postgresapp.com/

3.1.2 `rails new` with pg

By default, Rails uses SQLite3 for development databases. This database type is used to ease installation and development. However, since you are building an app that you intend to deploy, you will use PostgreSQL in all of your environments. Changing the default database type or other default settings is pretty straightforward when initializing a new Rails app.

```
rails new standup_app -T --database=postgresql
```

After a minute or two, Rails will have installed all the required files for the base application. Rails then installed the base gems and ran a few other shell commands to get your application started.

3.1.3 Database Configuration

If everything in the install went correctly, you would be able to use a Rails command to create the database on your local PostgreSQL instance.

To be safe, take a look at the database configuration file and see what is there. The database configuration file `config/database.yml` comes with information about installing the pg gem. Along with other option settings, you may change if you deviate with your PostgreSQL setup. Here is the relevant development connection information:

```
default: &default
  adapter: postgresql
  encoding: unicode
  pool: <%= ENV.fetch("RAILS_MAX_THREADS") { 5 } %>

development:
  <<: *default
  database: standup_app_development
```

Here Rails sets up a shared configuration between all environments setting the adapter type, encoding, and connection pool settings. None of this is relevant to the default usage of Rails, but you may need to change this at some point in your application's lifetime.

The development block specifies it will use the default settings and a database_name of `standup_app_development`. The default naming for Rails databases is `appname_environment`. Meaning your test database name would be `standup_app_test` and so on.

Now, let's change our directory to that of the Rails app and create the database for Rails to use.

```
cd standup_app
rails db:create
```

This spot is a perfect place to pause, take a breather, and celebrate that you have taken your first step in creating your *real* Rails application. In the next section, we will go over the next critical skill, source control.

3.2 Git gud enuf

The most shared and widespread way to manage your source code is Git. Git is a version control system that allows you to take

snapshots of your codebase. Also, it lets you branch your codebase and traverse the git history if you need to move back to a previous state. Understanding git may take a bit more explaining than this book could offer. I recommend reading the following chapter from git-SCM's HTML ebook[2]. If you are on macOS(OSX), you can make sure you have the git command line binary by having XCode's Command line tools. This is done by using the simple command:

```
Xcode-select --install
```

On other systems, you can use the following link to install Git [Git Install Instructions][b8c5cb2c].

[b8c5cb2c]: https://git-scm.com/book/en/v2/Getting-Started-Installing-Git "Git Install Instructions"

3.2.1 Git Commands

Now that you have Git installed, you can go over a few basic commands:

```
git init
```

Will create an empty Git repository or reinitialize an existing one.

```
git clone
```

Will copy an existing repo(usually remote) into a new directory.

```
git status
```

Will show the working tree status. Meaning you can see which files/folders have been added or modified

[2] https://git-scm.com/book/en/v2/Getting-Started-Git-Basics

```
git add
```

Will add files to your working index/branch to be committed.

```
git rm
```

Will remove a file from index/branch that would have been staged for commit.

```
git checkout
```

Allows you to switch between branches

```
git pull
```

Allows you to pull and merge changes from a remote branch into your local branch.

```
git push
```

Allows you to push committed changes from your local branch into a remote branch.

3.2.2 Git Usage

Now that you have a few commands under your belt, you can put them to actual use with your new application.

```
git init
git add .
git commit -am "Initial commit"
```

Here you are initializing a new git repository in the current folder and adding all files to be added by passing a `.` (which means current directory in most command-line systems), then committing all (`-a` flag) the changes, with a message(`-m` flag). So saying, you'd like to save the current progress of the whole folder to Git.

At this point, it is recommended you set up a remote repository host on a third-party git hosting system. There are three main places you can do this. GitHub[3], Gitlab[4] and BitBucket[5].

GitHub has been the leading solution in the Git hosting space for years and now offers free private repository hosting. Bitbucket and Gitlab have been offering free plans for much longer, however.

You may be asking, "What is a private repository?". When hosting your git repository, you have an option to make it private. This distinguishment hides the repository from everyone except yourself and any collaborators you add. On the other hand, a public repository is indexed by search engines and visible to anyone on the internet to see. You want to host a private repository for this book, and the assumption is that you want to create an application whose source code is not visible to the internet.

Github does offer the ability to host unlimited private and public repositories. There is another option called BitBucket. BitBucket is part of the Atlassian umbrella of products. Atlassian houses other tools such as JIRA, Confluence, and others. BitBucket offers a free plan that includes private repository hosting.

Additionally, Gitlab has a pretty robust internal CI/pipeline system that could replace CircleCi or others in your pipeline if you choose to set up such things.

Once you select a git host, the service will guide you through account setup. Then, set up a remote repository, granting your system access to connect, and given a command to run to connect

[3] https://github.com
[4] https://gitlab.com
[5] https://bitbucket.com

your local git repository to the remote repository. That command generally looks like:

```
git remote add origin git@github.com:username/repo_name.g\
it
```

Lastly, you will run a push command to push the current git repository to your remote repository:

```
git push -u origin main
```

Perfect, your codebase is now replicated on a remote git repository. This repository will be used as a backup, a central place to pull the repository onto other machines you may have, and be used in application deployments.

As we go further in the book, we will cover `git` branches, `git reset`, and other `git` commands as they apply to your process.

3.3 Gemfile and preferred gems to start

A Gemfile is a file used by Bundler (Rails' gem manager) to determine which gems to include in the application. It also determines what version of a gem to use. This book is intended to take you straight to the most practical and efficient way to create a SaaS application with Rails. Thus, you are using particular gems right out of the gate. The Gemfile in this book includes default gems, removes a few gems, and gems you will use in your application.

```
source "https://rubygems.org"
git_source(:github) { |repo| "https://github.com/#{repo}.\
git" }

ruby "3.0.4"

# Bundle edge Rails instead: gem "rails", github: "rails/\
rails", branch: "main"
gem "rails", "~> 7.0.2", ">= 7.0.2.3"

# The original asset pipeline for Rails [https://github.c\
om/rails/sprockets-rails]
gem "sprockets-rails"

# Use postgresql as the database for Active Record
gem "pg", "~> 1.1"

# Use the Puma web server [https://github.com/puma/puma]
gem "puma", "~> 5.0"

# Use JavaScript with ESM import maps [https://github.com\
/rails/importmap-rails]
gem "importmap-rails"

# Hotwire's SPA-like page accelerator [https://turbo.hotw\
ired.dev]
gem "turbo-rails"

# Hotwire's modest JavaScript framework [https://stimulus\
.hotwired.dev]
gem "stimulus-rails"

# Build JSON APIs with ease [https://github.com/rails/jbu\
ilder]
gem "jbuilder"
```

```
# Use Redis adapter to run Action Cable in production
gem "redis", "~> 4.0"

# Use Kredis to get higher-level data types in Redis [htt\
ps://github.com/rails/kredis]
# gem "kredis"

# Use Active Model has_secure_password [https://guides.ru\
byonrails.org/active_model_basics.html#securepassword]
# gem "bcrypt", "~> 3.1.7"

# Windows does not include zoneinfo files, so bundle the \
tzinfo-data gem
gem "tzinfo-data", platforms: %i[ mingw mswin x64_mingw j\
ruby ]

# Reduces boot times through caching; required in config/\
boot.rb
gem "bootsnap", require: false

# Use Sass to process CSS
# gem "sassc-rails"

# Use Active Storage variants [https://guides.rubyonrails\
.org/active_storage_overview.html#transforming-images]
# gem "image_processing", "~> 1.2"

gem 'local_time'

gem 'devise'
gem 'devise_invitable'
gem 'rolify'
gem 'cancancan'
```

```
gem 'immutable-struct'

gem 'sidekiq'
gem 'sinatra', require: nil

gem 'gravatar_image_tag'

gem 'money-rails'

gem 'slack-notifier'


group :development, :test do
  # See https://guides.rubyonrails.org/debugging_rails_ap\
plications.html#debugging-with-the-debug-gem
  gem "debug", platforms: %i[ mri mingw x64_mingw ]
  gem 'rspec-rails'
  gem 'rails-controller-testing'
  gem 'capybara'
  gem 'simplecov', :require => false, :group => :test
  gem "factory_bot_rails"
end

group :development do
  # Use console on exceptions pages [https://github.com/r\
ails/web-console]
  gem "web-console"

  # Add speed badges [https://github.com/MiniProfiler/rac\
k-mini-profiler]
  # gem "rack-mini-profiler"

  # Speed up commands on slow machines / big apps [https:\
//github.com/rails/spring]
  # gem "spring"
```

```
  gem "letter_opener"
  gem 'foreman'
end

gem 'rack-cors', :require => 'rack/cors'
```

You can run `bundle install` to install the gems specified in the Gemfile. However, if you run into a version issue when running that, you may need to run `bundle update` to resolve version differences. Bundler often tells you to run the update command if versions mismatch. Unfortunately, Gemfiles and their dependencies can be fickle. The included Gemfile is working at the point and time this book is being written. However, if you are copying code out of the included git repository, by copying the `Gemfile.lock`, you are guaranteed to get the working versions from the complete application.

Now let's go over some of the specific Gems added to our Gemfile:

- **local_time**[6] - This is a gem that includes a view helper and javascript helper to display relative time (i.e. 'updated 2 minutes ago')
- **devise**[7] **and devise_invitable**[8] - Devise is the authentication library most used in Rails for the past few years. While there are some up-and-coming libraries to provide authentication services, the support around Devise and its ease of use makes it great to move quickly on your SaaS application. Devise Invitable is an add-on gem that will allow you to have an invitation system to have users invite other users to your application.
- **rolify**[9] **and cancancan**[10] - Rolify allows us to assign roles

[6] https://github.com/basecamp/local_time
[7] https://github.com/plataformatec/devise
[8] https://github.com/scambra/devise_invitable
[9] https://github.com/RolifyCommunity/rolify
[10] https://github.com/CanCanCommunity/cancancan

such as User or Admin, then CanCanCan will enable us to take those roles and apply authorization policies.

- **Sidekiq**[11] - This gem allows us to run background jobs backed by a Redis queue. Instead of running tasks like sending emails on the main request loop, we can drop it into a Sidekiq queue to run in the background.
- **money-rails**[12] - This will help standardize around common issues found when calculating money and provide a few template helpers.
- **slack-notifier**[13] - Simple Slack API integration tool to quickly send a message to a Slack Webhook. We'll use this to send notifications to your Slack when a new user signs up or in other areas.
- **rspec-rails**[14], **factory_bot_rails**[15], **simplecov**[16] - These gems include the testing functionality discussed in Chapter 2.
- **letter_opener**[17] - This gem will open mail in the browser instead of attempting to send mail in the development environment.
- **rack-cors**[18] - This gem will help set up CORS rules. This functionality will be used if accessing JSON requests from domains that are not the current Rails app configured domain name.

This bulleted list is not an exhaustive list of all the gems used throughout your application. However, it *is* a list of the gems that help you get your application built and ready for customers.

Before moving on, let's run the following command to do some post-bundler initializing:

[11] https://github.com/mperham/sidekiq
[12] https://github.com/RubyMoney/money-rails
[13] https://github.com/stevenosloan/slack-notifier
[14] https://github.com/rspec/rspec-rails
[15] https://github.com/thoughtbot/factory_bot_rails
[16] https://github.com/colszowka/simplecov
[17] https://github.com/ryanb/letter_opener
[18] https://github.com/cyu/rack-cors

```
rails generate rspec:install
```

3.4 Using TailwindCSS

TailwindCSS[19] is a utility first CSS framework that simplifies the craft of building by the use of class names. These names are displayed and used in the markup to create a semantic HTML web page. In both theory and practice, this will allow you to reduce the time you spend in your CSS markup but design within the system's constraints, leaving it more manageable to maintain design and markup in your application.

Additionally, along with the design patterns built-in (Flexbox, responsive design, etc.), you will also find that TailwindCSS purges any unused CSS giving the smallest possible CSS needed to render your designed markup.

Lastly, the Rails team has committed to working with TailwindCSS, and it is easily bundled into Rails by default, which means getting started is just a few commands away!

First, a command to add the gem and let Bundler install the gem:

```
bin/bundle add tailwindcss-rails
```

Then, you can run the actual TailwindCSS setup:

[19] https://tailwindcss.com/

```
bin/rails tailwindcss:install
Add Tailwindcss include tags and container element in app\
lication layout
      insert  app/views/layouts/application.html.erb
      insert  app/views/layouts/application.html.erb
      insert  app/views/layouts/application.html.erb
Build into app/assets/builds
      create  app/assets/builds
      create  app/assets/builds/.keep
      append  app/assets/config/manifest.js
      append  .gitignore
Add default config/tailwindcss.config.js
      create  config/tailwind.config.js
Add default app/assets/stylesheets/application.tailwind.c\
ss
      create  app/assets/stylesheets/application.tailwind\
.css
Add default Procfile.dev
      create  Procfile.dev
Ensure foreman is installed
         run  gem install foreman from "."
Successfully installed foreman-0.87.2
Parsing documentation for foreman-0.87.2
Installing ri documentation for foreman-0.87.2
Done installing documentation for foreman after 0 seconds
1 gem installed
Add bin/dev to start foreman
      create  bin/dev
Compile initial Tailwind build
         run  rails tailwindcss:build from "."
```

Looking at the output, you can see that the command included the needed markup, build tools, configuration, and finally added `Foreman` to manage the multiple processes.

Specifically, `Foreman`, `bin/dev`, and the configuration file added

(`Procfile.dev`) make it as simple as one command to run two processes (and more later on!). If you open the Procfile, you can see how the processes are named and run:

```
web: bin/rails server -p 3000
css: bin/rails tailwindcss:watch
```

3.4.1 First Controller, Views, and Tests

Before implementing the layout, we need to apply a route and controller to bypass the default welcome page. We also will be using this controller in the application as we go along.

Let's also pause to mention that Rails comes with built-in command line generators to generate controllers, models, and mailers. As well as generating database migration files or even generating a scaffold of all of those combined. In our case, let's focus on the controller generator. The general format for this generator is:

```
rails generate controller Activity mine feed
      create  app/controllers/activity_controller.rb
       route  get 'activity/mine'
              get 'activity/feed'
      invoke  tailwindcss
      create    app/views/activity
      create    app/views/activity/mine.html.erb
      create    app/views/activity/feed.html.erb
      invoke  rspec
      create    spec/requests/activity_spec.rb
      create    spec/views/activity
      create    spec/views/activity/mine.html.tailwindcss\
_spec.rb
      create    spec/views/activity/feed.html.tailwindcss\
_spec.rb
      invoke  helper
```

```
      create    app/helpers/activity_helper.rb
      invoke    rspec
      create      spec/helpers/activity_helper_spec.rb
```

You may notice `tailwindcss_spec` in the view specs being generated. This naming seems to be a bug (at the time of writing) with the naming, but it does not affect the running or use of the specs themselves.

Let's make one quick edit to the route file to define a default route, adding `root to: 'activity#mine'` to the bottom of the route file, as follows:

```
Rails.application.routes.draw do
  get 'activity/mine'

  get 'activity/feed'

  root to: 'activity#mine'
end
```

As you saw, there were a few test files automatically created by the generator. Let's run the RSpec command to see what happens now:

```
rspec
*..**

Pending: (Failures listed here are expected and do not af\
fect your suites status)

  1) ActivityHelper add some examples to (or delete) /sta\
ndup_app/spec/helpers/activity_helper_spec.rb
     # Not yet implemented
     # ./spec/helpers/activity_helper_spec.rb:14
```

```
  2) activity/feed.html.tailwindcss add some examples to \
(or delete) /standup_app/spec/views/activity/feed.html.ta\
ilwindcss_spec.rb
     # Not yet implemented
     # ./spec/views/activity/feed.html.tailwindcss_spec.r\
b:4

  3) activity/mine.html.tailwindcss add some examples to \
(or delete) /standup_app/spec/views/activity/mine.html.ta\
ilwindcss_spec.rb
     # Not yet implemented
     # ./spec/views/activity/mine.html.tailwindcss_spec.r\
b:4


Finished in 0.34866 seconds (files took 2 seconds to load)
5 examples, 0 failures, 3 pending
```

It also created a helper test file with a pending stub, which we delete. Lastly, it created pending test stubs for each of our view files. This pause would be a good time to write some simple tests there to get some practice.

First, let's go ahead and edit the `feed` and `mine` specs (which I went ahead and renamed, but you are free to leave the names as they were generated).

```
require 'rails_helper'

RSpec.describe "activity/feed.html.erb", type: :view do
  it "renders the word feed" do
    render template: 'activity/feed'
    expect(rendered).to match(/feed/)
  end
end


require 'rails_helper'

RSpec.describe "activity/mine.html.erb", type: :view do
  it "renders the word feed" do
    render template: 'activity/mine'
    expect(rendered).to match(/mine/)
  end
end
```

Run the `rspec` command to now run the tests:

```
rspec
....

Finished in 0.12135 seconds (files took 1.72 seconds to l\
oad)
4 examples, 0 failures
```

As you see, we are good to go. There are four passing specs and no pending or unneeded specs.

Now that you have some testing out of the way let's start to set up your layout.

3.4.2 Application layout

First, we can modify the main layout file to include the necessary elements and classes to wrap the content provided by routed templates.

```
<!DOCTYPE html>
  <html>
    <head>
    <title>StandupApp</title>
    <meta name="viewport" content="width=device-width,ini\
tial-scale=1">
    <%= csrf_meta_tags %>
    <%= csp_meta_tag %>
    <%= stylesheet_link_tag "tailwind", "inter-font", "da\
ta-turbo-track": "reload" %>

    <%= stylesheet_link_tag "application", "data-turbo-tr\
ack": "reload" %>
    <%= javascript_importmap_tags %>
  </head>

  <body class="bg-gray-800 font-sans leading-normal track\
ing-normal mt-12">

    <%= render partial: 'layouts/components/header' %>
    <div class="flex flex-col md:flex-row">
      <%= render partial: 'layouts/components/sidebar' %>
      <div class="main-content flex-1 bg-gray-100 mt-12 m\
d:mt-2 pb-24 md:pb-5">

        <div class="flex flex-wrap p-3">
          <%= yield %>
        </div>
        <%= render partial: 'layouts/components/footer' %>
      </div>
```

```
    </div>
  </body>
</html>
```

The details of this file are not all pertinent to the addition of the TailwindCSS, so we'll only touch on those that matter. First, classes are added to the body element, `g-gray-800 font-sans leading-normal tracking-normal mt-12`. Next, those `render partial:` calls allow us to add markup from other template files in that position. In this case, it adds various larger parts of the application template, such as the header, sidebar, or footer. Plus, it keeps the layout to only the most necessary foundational elements to build the page structure. Then, `<%= yield %>` is the Rails template method for adding the template from the controller and action called.

Since we are using partial templates, we create the header file next.



One thing to note here is using the underscore in the template file's name. In Rails, partial template files are designated with an underscore at the beginning of the file name. What are partials? Partials are simply a way to break down templates into smaller chunks. This approach comes into far greater play in situations such as looping over a collection of items. You would make the template chunk to display item information from a partial. For example, the header template is titled `_header.html.erb`.

```
<header>
  <!--Nav-->
  <nav aria-label="menu nav" class="bg-gray-800 pt-2 md:p\
t-1 pb-1 px-1 mt-0 h-auto fixed w-full z-20 top-0">

    <div class="flex flex-wrap items-center">
      <div class="flex flex-shrink md:w-1/3 justify-cente\
r md:justify-start text-white">
        <a href="#" aria-label="Home">
          <span class="text-xl pl-2">Standup App</span>
        </a>
      </div>

      <div class="flex flex-1 md:w-1/3 justify-center md:\
justify-start text-white px-2">
        <span class="relative w-full">
          <input aria-label="search" type="search" id="se\
arch" placeholder="Search" class="w-full bg-gray-900 text\
-white transition border border-transparent focus:outline\
-none focus:border-gray-400 rounded py-3 px-2 pl-10 appea\
rance-none leading-normal">
          <div class="absolute search-icon" style="top: 1\
rem; left: .8rem;">
            <svg class="fill-current pointer-events-none \
text-white w-4 h-4" xmlns="http://www.w3.org/2000/svg" vi\
ewBox="0 0 20 20">
              <path d="M12.9 14.32a8 8 0 1 1 1.41-1.41l5.\
35 5.33-1.42 1.42-5.33-5.34zM8 14A6 6 0 1 0 8 2a6 6 0 0 0\
 0 12z"></path>
            </svg>
          </div>
        </span>
      </div>

      <div class="flex w-full pt-2 content-center justify\
```

```
-between md:w-1/3 md:justify-end">
        <ul class="list-reset flex justify-between flex-1\
 md:flex-none items-center">
          <li class="flex-1 md:flex-none md:mr-3">
            <a class="inline-block py-2 px-4 text-white n\
o-underline" href="#">Active</a>
          </li>
          <li class="flex-1 md:flex-none md:mr-3">
            <a class="inline-block text-gray-400 no-under\
line hover:text-gray-200 hover:text-underline py-2 px-4" \
href="#">link</a>
          </li>
          <li class="flex-1 md:flex-none md:mr-3">
            <div data-controller="dropdown" class="relati\
ve inline-block">
              <button data-action="dropdown#toggle click@\
window->dropdown#hide" class="drop-button text-white py-2\
 px-2"> <span class="pr-2"></span> Hi, User <svg class="h\
-3 fill-current inline" xmlns="http://www.w3.org/2000/svg\
" viewBox="0 0 20 20">
                <path d="M9.293 12.95l.707.707L15.657 8l-\
1.414-1.414L10 10.828 5.757 6.586 4.343 8z" /></svg></but\
ton>
              <div data-dropdown-target="menu" class="hid\
den absolute bg-gray-800 text-white right-0 mt-3 p-3 over\
flow-auto z-30" data-transition-enter-from="opacity-0 sca\
le-95" data-transition-enter-to="opacity-100 scale-100" d\
ata-transition-leave-from="opacity-100 scale-100" data-tr\
ansition-leave-to="opacity-0 scale-95" id="nav-dropdown">
                <a href="#" data-action="dropdown#toggle"\
 class="p-2 hover:bg-gray-800 text-white text-sm no-under\
line hover:no-underline block"><i class="fa fa-user fa-fw\
"></i> Profile</a>
                <a href="#" data-action="dropdown#toggle"\
 class="p-2 hover:bg-gray-800 text-white text-sm no-under\
```

```
line hover:no-underline block"><i class="fa fa-cog fa-fw"\
></i> Settings</a>
                <div class="border border-gray-800"></div>
                <a href="#" data-action="dropdown#toggle"\
 class="p-2 hover:bg-gray-800 text-white text-sm no-under\
line hover:no-underline block"><i class="fas fa-sign-out-\
alt fa-fw"></i> Log Out</a>
              </div>
            </div>
          </li>
        </ul>
      </div>
    </div>

  </nav>
</header>
```

This template contains the structure to have the app title/logo, dropdowns, and any additional links you may need for navigation. To get the app template structure setup, we can leave some of the demo data and elements to show how the dropdowns would look and work.

We need to fix here that, unlike other CSS frameworks, Tailwind-CSS does not come with any built-in components. To make your way past this problem, you need to purchase their premium component library called TailwindUI or build these features yourself. Fortunately, such a feature is already created by using a Stimulus library and adjusting the data attributes as you saw above in the markup.



Stimulus is part of the HotWire family, just like the Turbo framework explained later in the chapter.

First, you will need to install the Stimulus library, and we can do

that with `importmap-rails`



What are Importmaps, and what do they have to do with rails? From their repo's description, "Import maps let you import JavaScript modules using logical names that map to versioned/digested files – directly from the browser. So you can build modern JavaScript applications using JavaScript libraries made for ES modules (ESM) without the need for transpiling or bundling. This frees you from needing Webpack, Yarn, npm, or any other part of the JavaScript toolchain. All you need is the asset pipeline that's already included in Rails.". Simply put, it allows quick use of NPM libraries without going through bigger node build systems like Webpack, etc.

Use this command to install `stimulus-dropdown`:

```
bin/importmap pin stimulus-dropdown --download
Pinning "stimulus-dropdown" to vendor/javascript/stimulus\
-dropdown.js via download from https://ga.jspm.io/npm:sti\
mulus-dropdown@2.0.0/dist/stimulus-dropdown.es.js
Pinning "@hotwired/stimulus" to vendor/javascript/@hotwir\
ed/stimulus.js via download from https://ga.jspm.io/npm:@\
hotwired/stimulus@3.0.1/dist/stimulus.js
Pinning "hotkeys-js" to vendor/javascript/hotkeys-js.js v\
ia download from https://ga.jspm.io/npm:hotkeys-js@3.8.9/\
dist/hotkeys.esm.js
Pinning "stimulus-use" to vendor/javascript/stimulus-use.\
js via download from https://ga.jspm.io/npm:stimulus-use@\
0.50.0-2/dist/index.js
```

This command will download the library and needed dependencies to your application and allow you to import it in your javascript files, which happens to be the next step. Edit the top of your `app/javascript/controllers/application.js` to look like this:

```
import { Application } from "@hotwired/stimulus"
import Dropdown from 'stimulus-dropdown'

const application = Application.start()
application.register('dropdown', Dropdown)
```

With these additions coupled with the data attributes in the header markup, you should have a functioning dropdown in the navigation portion of the page. Now, onto the sidebar.

The following partial to create is the `_sidebar` template:

```
<div class="bg-gray-800 shadow-xl h-16 fixed bottom-0 mt-\
12 md:relative md:h-screen z-10 w-full md:w-48">

  <div class="md:mt-12 md:w-48 md:fixed md:left-0 md:top-\
0 content-center md:content-start text-left justify-betwe\
en">
    <ul class="list-reset flex flex-row md:flex-col py-0 \
md:py-3 px-1 md:px-2 text-center md:text-left">
      <li class="mr-3 flex-1">
        <a href="#" class="block py-1 md:py-3 pl-1 align-\
middle text-white no-underline hover:text-white border-b-\
2 border-gray-800 hover:border-pink-500">
          <i class="fas fa-tasks pr-0 md:pr-3"></i><span \
class="pb-1 md:pb-0 text-xs md:text-base text-gray-600 md\
:text-gray-400 block md:inline-block">Tasks</span>
        </a>
      </li>
      <li class="mr-3 flex-1">
        <a href="#" class="block py-1 md:py-3 pl-1 align-\
middle text-white no-underline hover:text-white border-b-\
2 border-gray-800 hover:border-purple-500">
          <i class="fa fa-envelope pr-0 md:pr-3"></i><spa\
n class="pb-1 md:pb-0 text-xs md:text-base text-gray-600 \
md:text-gray-400 block md:inline-block">Messages</span>
```

```
        </a>
      </li>
      <li class="mr-3 flex-1">
        <a href="#" class="block py-1 md:py-3 pl-1 align-\
middle text-white no-underline hover:text-white border-b-\
2 border-blue-600">
          <i class="fas fa-chart-area pr-0 md:pr-3 text-b\
lue-600"></i><span class="pb-1 md:pb-0 text-xs md:text-ba\
se text-white md:text-white block md:inline-block">Analyt\
ics</span>
        </a>
      </li>
      <li class="mr-3 flex-1">
        <a href="#" class="block py-1 md:py-3 pl-0 md:pl-\
1 align-middle text-white no-underline hover:text-white b\
order-b-2 border-gray-800 hover:border-red-500">
          <i class="fa fa-wallet pr-0 md:pr-3"></i><span \
class="pb-1 md:pb-0 text-xs md:text-base text-gray-600 md\
:text-gray-400 block md:inline-block">Payments</span>
        </a>
      </li>
    </ul>
  </div>
</div>
```

As you see, the sidebar partial is a bit smaller and more straightforward. In addition, it contains links to other parts of the application that we change later.

The last major piece of the layout is the footer. It is simple, and it is short. Again, it is another partial. Here is the ERB markup:

```
<footer class="w-full fixed bottom-0 p-1 pl-16 bg-gray-30\
0 text-gray-600">
  © 2022 Copyright: Your Company
</footer>
```

3.4.3 Turbo

There is one more thing to touch on before ending this chapter, and it is important enough to have its section before getting into some more in-depth topics. Turbo, created by Basecamp(partly owned by the creator of Ruby on Rails), is part of the HotWire family and is a default Rails javascript library whose main job is to make your application faster to navigate.

Turbo achieves this by hijacking the links within a page and replacing only the markup that changes based on the route/controller/action. This behavior can be noteworthy because it lets you continue to build a Rails application with few changes instead of learning how to implement the "Front End" of the application in an entirely different javascript framework or library(EmberJS, ReactJS, VueJS, etc.).

The caveat to not changing any code is that Turbo creates listeners to specific Turbo javascript events. This causes an issue with other libraries' standard jQuery-ready events. However, the fix at most times is to make sure you are binding the libraries listeners within a Turbo's version of the DOM being ready (which is `document.ready` if you are already jQuery inclined)

3.4.4 One more thing...

With all your progress in this chapter, it would be an excellent time to commit your changes. Use git and push the changes to your remote repository. Here are the commands you could use if you haven't been committing more frequently already:

```
git add .
git commit -am "Added an application layout"
git push
```

##What did you learn in this chapter?

- You learned how to start a new Rails app with non-default settings. Including starting without including the default test framework and using PostgreSQL as a database.
- You learned some basics about git and remote git repositories.
- You learned about some of the more valuable gems when building a purposeful Rails application
- You generated a controller, actions, and views from the command line.
- You wrote helpful view tests.
- You implemented TailwindCSS and a theme to your application, which included markup, javascript, and stylesheets.
- You learned a basic amount about Turbolinks.

3.5 Exercises

You can find exercise answers with any package or bundle purchases of Build A SaaS App in Ruby on Rails 7[20]!

1. While it may not always be the most preferred approach, you can use rails to serve static pages. Create a Controller, Route, and View template for a static page that will display some text on how to reach you for support.
2. While the Rails app you will be creating in this book had only one change in the `rails new` command, it would benefit you to look at the help text for the commands you ran in this chapter. For example, run `rails new --help` and `rails generate controller --help`.

[20] https://buildasaasappinrails.com

4. Users are Everything

In the last chapter, you set up your application to build upon it rapidly:
You have an application that interfaces with PostgreSQL.
You implemented a layout with a Bootstrap theme that provides a robust admin style for your application.
You added all the valuable gems a production-ready application could need.

The most vital part of any application is its users. Unfortunately, though, it isn't as simple as just having a database table called `Users`.

Almost any application needs an authentication system. While some developers may try to "roll their own" authentication, you will use one of the most widely known gems for such authentication, Devise. Devise provides the system to handle authentication sessions and cookies. Additionally, it includes middleware, Controllers, and Views by default. The impressive part is when you need to customize the login system, you can copy the files into your application and make edits as you see fit.

In the spirit of extending Devise, you can use the `devise_invitable` gem to implement the functionality of users in your application inviting other users. This feature is crucial for allowing your customers to manage their users.

User objects won't only house simple attributes such as names or time zones but handle role attributes using Rolify. With roles, the users can now be utilized in an authorization framework to set up access control throughout your application.

4.1 User Model

Here is an excellent chance for you to get used to creating a new branch in git. This functionality allows you to make a copy of the main branch but have confidence you can make changes. Using branches for new features and functionality becomes vital when you have an app up and running. For example, suppose you wanted to make significant changes to your user panel but also need to make a bug fix. In this case, your new feature would have its branch allowing you to fix the bug on the main branch and push the fix to your live production server(s).

```
git checkout -b users
```

With that out of the way, you can now finally start setting up the user model and everything that goes with it. But, first, let's go over models in a bit more depth.

Devise handles the email field and many of Devise's other attributes around passwords, tokens, and metadata.

Instead, you can start a model with other attributes you can use in the application. For example, you add a name, time zone, and hash_id (to obfuscate the primary id key from the URL).

In Rails, you modify the database schema through a process called migrations. A Migration is a DSL[1] (domain-specific language, or in short, an even more terse sublanguage for a particular domain area) for creating, modifying, or deleting tables or table schemas, removing the need to do all of this through raw SQL.

Migrations are versioned with timestamps to allow you to replay these database changes in new environments or machines. This tool also allows the Rails command-line tools to know which migration files to run at a given instance. Speaking of the command line, Rails comes with a command-line interface for generating migrations.

[1] https://en.wikipedia.org/wiki/Domain-specific_language

4.1.1 But first, UUID

What is UUID? A UUID is a universally unique identifier or a long string of cryptographically unique characters. Why? One of the attack vectors of many web applications is incrementing integers as a primary key used in a URL. Thus, an attacker could try to test security by accessing endpoints by incrementing the ID in the URL. While we will use roles and permissions to secure such things from happening, security through obfuscation is yet one other layer of protection to add. Additionally, PostgreSQL offers a built-in library and tool to generate UUIDs, and Rails allows you to plug UUIDs in as primary keys very quickly. Let's make the changes necessary to have this work.

First, we will add a migration to turn on this functionality in PostgreSQL.

```
rails g migration enable_extension_for_uuid
```

```
class EnableExtensionForUuid < ActiveRecord::Migration[7.\
0]
  def change
    enable_extension 'pgcrypto' unless extension_enabled?\
('pgcrypto')
  end
end
```

After saving that file, run `rails db:migrate` to migrate the database. We can take one more step here to make sure Rails knows to use UUID as the primary key for each table you create. This change is accomplished by adding an initializer file that adds a setting for generators that modify tables.



What is an `initializer`?

An initializer is a file in the `config/initializers` directory, which will be run once at application boot-up. This directory is a great area to add application settings, third-party API keys, or anything you need to load once before the application runs. Initializers will run alphabetically in the initializers folder unless you use specific syntax. That syntax specifies if it should be inserted before or after another initializer or class.

To add this application-wide setting, add the following initializer:

```
Rails.application.config.generators do |g|
  g.orm :active_record, primary_key_type: :uuid
end
```

4.1.2 Creating a Users Table

Now, you can create the model, tests, and more with a command for creating a model.

```
rails g model User name:string
```

Before running this, let's dissect the syntax here. First, `rails g` is a shortcut for `rails generate`. This shortcut can be used in any of our generation commands. Earlier, you created a controller, but you can generate models, routes, background jobs, mailers, etc. You can even generate a fair number of those at once using a scaffold. A Scaffold will produce a complete RESTful route collection, a controller with RESTful actions, a base model with specified attributes, etc. Scaffolds are very powerful but often need tweaking after the fact to achieve the results you are looking for.

Next is the `model`, which is the type of resource to generate. Which is followed by the Model name `User`. Notice here that `User` is a capitalized singular version of the Model noun. Rails will understand this and create other resources such as the table name with the plural version to match the framework's conventions. This convention is just another place Rails is making your life easier.

The last part of the command is `name:string`. After your resource name, you can list the attributes followed by their types. The string part can be omitted for strings since this is the default type for attributes.

Let's look at generated files:

```
class CreateUsers < ActiveRecord::Migration[6.0]
  def change
    create_table :users, id: :uuid do |t|
      t.string :name

      t.timestamps
    end
  end
end
```



Missing a `UUID`?

If your migration doesn't look like the file above and specifically the `id: uuid` is missing, the Spring tool could be messing things up here. Spring preloads the rails app to make commands that access the application (generators, servers, RSpec) use the preloaded version. Unfortunately, sometimes, it doesn't update the application when it should. So, if you are experiencing a head-scratching problem like the above, it is time to run `spring stop` in your rails console. This command will make Spring stop preloading and reload the application on the following Rails-based command. Finally, add in the `, id: :uuid` manually to ensure it is included in this table.

Most Rails migrations will include one change method, which will be used for both migrating and rolling back a migration. In a few cases, you will see `up` and `down` methods if the behavior is different between migrating and rollbacks.

Next is the basic User model:

```
class User < ApplicationRecord
end
```

This syntax instantiates a `User` class, extending the `ApplicationRecord` class. `ApplicationRecord` was new in Rails 5, and is a fundamental class for shared model code. ApplicationRecord extends ActiveRecord::Base itself, which gives a Rails model its "magic".

Rails also generates a model spec that contains a default pending test to be written. Tests will be added to the file shortly when more particular functionality is added to the `User` object.

Lastly, a FactoryBot factory file for the user is created. It was set up automatically based on the attributes and their types. In this case, just `name` and a string.

```
FactoryBot.define do
  factory :user do
    name {"MyString"}
  end
end
```

It would be best if you now run the migration with the following command:

```
rails db:migrate
```

Running the `db:migrate` command tells your Rails application to apply any changes from migration files from the last time migrations were run. If you need to revert the change that command just made, you can run this quick command:

```
rails db:rollback
```

There is one issue with UUID that crops up by default. Rails orders by primary key, and by default, that primary key is an auto-incrementing integer. Thus, Rails can perform a necessary sort on a number. However, a UUID does not offer any built-in time base generation in the UUID that it creates. Hence, when it sorts by `id`, in this case, strings, they will not be in the order made. Fear not, though, there is a fix for this. Starting in Rails 7.0, you can set an implicit sort column on a model, and with the ApplicationRecord parent class, we can add it to all models with one line of code!

```
class ApplicationRecord < ActiveRecord::Base
  primary_abstract_class
  self.implicit_order_column = "created_at" # new setting\
 here
end
```

As you can see, we can now have the order correct and match default Rails behavior when sorting records on a table.

With the model, table, and one column in place, let's write another migration to add the other column you need. You will be adding the column `time_zone`. The time zone column should be self-explanatory.

```
rails g migration AddTimeZoneToUsers time_zone:string
rails db:migrate
```

4.1.3 Installing Devise

With all of your non-Devise attributes in place, it is time to go through the installation process for Devise. First, Devise will have you run a command to handle the general install and then apply the Devise logic to a model (Users in your case). Finally, you will execute the optional view generation command so you can customize the look of your Devise-based forms.

```
rails generate devise:install
rails generate devise User
rails db:migrate
rails generate devise:views
```

To get an idea of the table columns and model attributes Devise had added, you can look at the migration file Devise created and Rails then migrated:

```
# frozen_string_literal: true

class AddDeviseToUsers < ActiveRecord::Migration[7.0]
  def self.up
    change_table :users do |t|
      ## Database authenticatable
      t.string :email,              null: false, default:\
 ""
      t.string :encrypted_password, null: false, default:\
 ""

      ## Recoverable
      t.string   :reset_password_token
      t.datetime :reset_password_sent_at

      ## Rememberable
      t.datetime :remember_created_at

      ## Trackable
      # t.integer  :sign_in_count, default: 0, null: false
      # t.datetime :current_sign_in_at
      # t.datetime :last_sign_in_at
      # t.string   :current_sign_in_ip
      # t.string   :last_sign_in_ip

      ## Confirmable
      # t.string   :confirmation_token
      # t.datetime :confirmed_at
      # t.datetime :confirmation_sent_at
      # t.string   :unconfirmed_email # Only if using rec\
onfirmable

      ## Lockable
      # t.integer  :failed_attempts, default: 0, null: fa\
lse # Only if lock strategy is :failed_attempts
```

```
      # t.string   :unlock_token # Only if unlock strateg\
y is :email or :both
      # t.datetime :locked_at


      # Uncomment below if timestamps were not included i\
n your original model.
      # t.timestamps null: false
    end

    add_index :users, :email,                unique: true
    add_index :users, :reset_password_token, unique: true
    # add_index :users, :confirmation_token,   unique: tr\
ue
    # add_index :users, :unlock_token,         unique: tr\
ue
  end

  def self.down
    # By default, we don't want to make any assumption ab\
out how to roll back a migration when your
    # model already existed. Please edit below which fiel\
ds you would like to remove in this migration.
    raise ActiveRecord::IrreversibleMigration
  end
end
```

As you can see, it adds some pretty essential fields such as email and password to provide basic login functionality but also adds columns that will deal with password reset functionality and more.

Next, you can take a quick peek at the changes to the `User` model file:

```
class User < ApplicationRecord
  # Include default devise modules. Others available are:
  # :confirmable, :lockable, :timeoutable, :trackable and\
 :omniauthable
  devise :database_authenticatable, :registerable,
         :recoverable, :rememberable, :validatable
end
```

Now the `User` model has the Devise modules included. There are no additional changes to make, but you can use this file as a reference to dig into Devise and want to use more of its extended functionality.

With Devise installed, you can also write some unit tests for the `User` model. You will prove that creating a `User` works with Devise connected through a Factory. First edit the `spec/factories/users.rb` file:

```
FactoryBot.define do
  factory :user do
    name {"MyString"}
    email {"email.test@mail.com"}
    password {"123ewq"}
  end
end
```

Back to the `spec/models/user_spec.rb` file. It should look like this after adding those tests:

```
require 'rails_helper'

RSpec.describe User, type: :model do
  context "valid Factory" do
    it "has a valid factory" do
      expect(FactoryBot.build(:user)).to be_valid
    end
  end
end
```

This test may not look like much, but it sets the table for testing models in the future. I intentionally am leaving out Model specs for the email and password as they are already tested through the Devise gem. However, the authentication system will still be tested later using feature tests.

While you are in the testing mode, you can now set up another one of the testing tools from Chapter 2. Next, you will set up SimpleCov.

Add the following to the top of `spec/spec_helper.rb`

```
require 'simplecov'
SimpleCov.start 'rails'
```

Add the word `coverage` to the last line in `.gitignore` so your coverage folder does not get tracked in your git repository. Then run `rspec`:

```
rspec
.....

Finished in 0.21772 seconds (files took 7.42 seconds to l\
oad)
5 examples, 0 failures

Coverage report generated for RSpec to standup_app/covera\
ge. 11 / 25 LOC (44.0%) covered.
```

The coverage may seem low as a few default generated application-level metaclasses have lines but are not tested. However, SimpleCov will check these files when some of their children's classes are created and tested.

Here is an excellent spot to take a break and commit your code to your git repository. Here are the commands once again to do so:

```
git add .
git commit -am "Devise Install"
git push
```

4.2 Sign in, out, and up

While the particular view files for the Sign-up and Sign-in pages were generated automatically, Rails would yield them into the already existing application layout without any further changes. This layout is meant for a logged-in user. In this section, you will create an alternative template. Then, set up a condition in `ApplicationController` to switch the layouts based on the visitor's request. Finally, we'll add a bit of the theme's style to the pre-created Devise pages.

First, you will create the new layout in a new file `app/views/layouts/devise.html.erb`

```
<!DOCTYPE html>
  <html class="h-full">
    <head>
    <title>StandupApp</title>
    <meta name="viewport" content="width=device-width,ini\
tial-scale=1">
    <%= csrf_meta_tags %>
    <%= csp_meta_tag %>
    <%= stylesheet_link_tag "tailwind", "inter-font", "da\
ta-turbo-track": "reload" %>

    <%= stylesheet_link_tag "application", "data-turbo-tr\
ack": "reload" %>
    <%= javascript_importmap_tags %>
  </head>

  <body class="bg-gray-200 h-full">
    <div class="min-h-full flex items-center justify-cent\
er">
      <div class="bg-neutral-50 shadow-md rounded-sm bord\
er m-auto min-h-full flex items-center justify-center py-\
12 px-4 sm:px-6 lg:px-8">
        <div class="max-w-md w-full space-y-8">
          <%= yield %>
        </div>
      </div>
    </div>
  </body>
</html>
```

You may pick up the pattern here from the application layout. This layout sets up the static portion of the layout and specifies where Rails will = `yield` the action's view into the page.

Next, you will modify the `apps/controllers/application_controller.rb` to pick the correct layout dynamically.

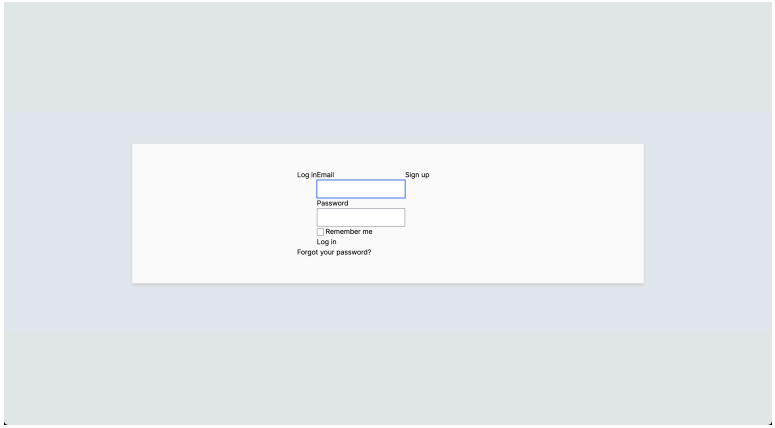
```
class ApplicationController < ActionController::Base

  before_action :authenticate_user!
  layout :layout_by_resource

  protected
  def layout_by_resource
    if devise_controller?
      "devise"
    else
      "application"
    end
  end
end
```

Going over specific lines on this file:

- Line 3: This `before_action` syntax triggers Devise to allow only authenticated users to access any action within the application. In future Chapters, you will learn how to exclude controllers or specific actions from authentication.
- Line 4: This is setting the layout to the result of the `layout_by_resource` method.
- Lines 6-13: This method branches conditionally on the boolean (true-false) response to the `devise_controller?` method. Devise automatically provides that method and tells you if the controller being used is a Devise controller or not. If yes, use the `devise` layout, otherwise (else) use the standard application layout

If you were to head to http://localhost:3000/users/sign_in[2] right now, you would see the Sign In page rendered automatically and in the `devise` layout.

[2] http://localhost:3000/users/sign_in



4.2.0.1 Sign In

Great! Now it's time to get that page styled with the correct markup and attributes! The best way to show you the difference would be to show you the markup before and after, as there are many changes.

```
<h2>Log in</h2>

<%= form_for(resource, as: resource_name, url: session_pa\
th(resource_name)) do |f| %>
  <div class="field">
    <%= f.label :email %><br />
    <%= f.email_field :email, autofocus: true, autocomple\
te: "email" %>
  </div>

  <div class="field">
    <%= f.label :password %><br />
    <%= f.password_field :password, autocomplete: "curren\
t-password" %>
  </div>
```

```
  <% if devise_mapping.rememberable? %>
    <div class="field">
      <%= f.check_box :remember_me %>
      <%= f.label :remember_me %>
    </div>
  <% end %>

  <div class="actions">
    <%= f.submit "Log in" %>
  </div>
<% end %>

<%= render "devise/shared/links" %>
```

..and after:

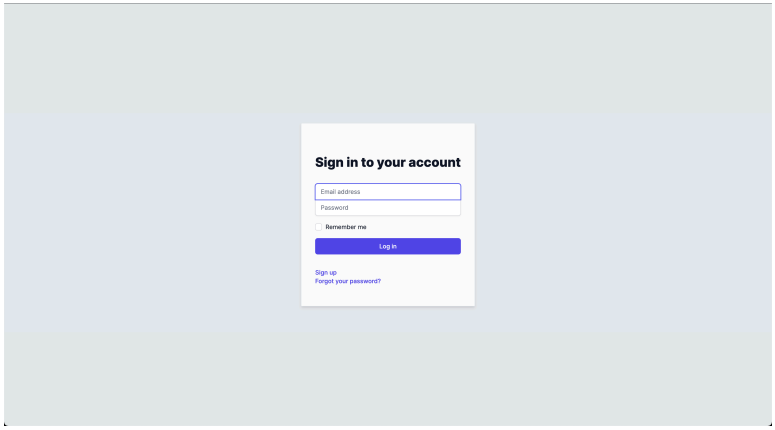
```
<h2 class="mt-6 text-center text-3xl font-extrabold text-\
gray-900"> Sign in to your account </h2>
<%= form_for(resource, as: resource_name, url: session_pa\
th(resource_name), html: {data: {turbo: false}, class: "m\
t-8 space-y-6"}) do |f| %>
  <div class="rounded-md shadow-sm -space-y-px">
    <div>
      <%= f.label :email, "Email address", class: "sr-onl\
y" %>
      <%= f.email_field :email, autofocus: true, autocomp\
lete: "email", class: "appearance-none rounded-none relat\
ive block w-full px-3 py-2 border border-gray-300 placeho\
lder-gray-500 text-gray-900 rounded-t-md focus:outline-no\
ne focus:ring-indigo-500 focus:border-indigo-500 focus:z-\
10 sm:text-sm", placeholder: "Email address" %>
    </div>
    <div>
      <%= f.label :password, "Password", class: "sr-only"\
 %>
```

```
      <%= f.password_field :password, autocomplete: "curr\
ent-password", class: "appearance-none rounded-none relat\
ive block w-full px-3 py-2 border border-gray-300 placeho\
lder-gray-500 text-gray-900 rounded-b-md focus:outline-no\
ne focus:ring-indigo-500 focus:border-indigo-500 focus:z-\
10 sm:text-sm", placeholder: 'Password' %>
    </div>
  </div>

  <div class="flex items-center justify-between py-4">
    <% if devise_mapping.rememberable? %>
      <div class="flex items-center">
        <%= f.check_box :remember_me, class: "h-4 w-4 tex\
t-indigo-600 focus:ring-indigo-500 border-gray-300 rounde\
d" %>
        <%= f.label :remember_me, class: "ml-2 block text\
-sm text-gray-900" %>
      </div>
    <% end %>


  </div>

  <div>
    <%= f.submit "Log in", class: "group relative w-full \
flex justify-center py-2 px-4 border border-transparent t\
ext-sm font-medium rounded-md text-white bg-indigo-600 ho\
ver:bg-indigo-700 focus:outline-none focus:ring-2 focus:r\
ing-offset-2 focus:ring-indigo-500" %>
  </div>
<% end %>
<div class="text-sm">
  <%= render "devise/shared/links" %>
</div>
```

Most of the changes are TailwindCSS classes and form CSS classes

to clean up the page's look.

Refreshing http://localhost:3000/users/sign_in[3] should now bring up a page that looks more like:



Styled Sign In Page

4.2.0.2 Sign Up

Perfect! Well, kind of…

You can sign in as a user if the user exists. If you click on the `Sign Up` link for the login page right now, it takes you to an unstyled default Devise page. Let's go ahead and reconstruct the template to work within the TailwindCSS.

[3] http://localhost:3000/users/sign_in

```
<h2>Sign up</h2>

<%= form_for(resource, as: resource_name, url: registrati\
on_path(resource_name)) do |f| %>
  <%= render "devise/shared/error_messages", resource: re\
source %>

  <div class="field">
    <%= f.label :email %><br />
    <%= f.email_field :email, autofocus: true, autocomple\
te: "email" %>
  </div>

  <div class="field">
    <%= f.label :password %>
    <% if @minimum_password_length %>
    <em>(<%= @minimum_password_length %> characters minim\
um)</em>
    <% end %><br />
    <%= f.password_field :password, autocomplete: "new-pa\
ssword" %>
  </div>

  <div class="field">
    <%= f.label :password_confirmation %><br />
    <%= f.password_field :password_confirmation, autocomp\
lete: "new-password" %>
  </div>

  <div class="actions">
    <%= f.submit "Sign up" %>
  </div>
<% end %>

<%= render "devise/shared/links" %>
```

..and after:

```
<h2 class="mt-6 text-center text-3xl font-extrabold text-\
gray-900"> Sign up an account </h2>
<%= form_for(resource, as: resource_name, url: registrati\
on_path(resource_name), html: {data: {turbo: false}, clas\
s: "mt-8 space-y-6"}) do |f| %>
  <%= render "devise/shared/error_messages", resource: re\
source %>
  <div class="rounded-md shadow-sm -space-y-px">
    <div>
      <%= f.label :email, "Email address", class: "sr-onl\
y" %>
      <%= f.email_field :email, autofocus: true, autocomp\
lete: "email", class: "appearance-none rounded-none relat\
ive block w-full px-3 py-2 border border-gray-300 placeho\
lder-gray-500 text-gray-900 rounded-t-md focus:outline-no\
ne focus:ring-indigo-500 focus:border-indigo-500 focus:z-\
10 sm:text-sm", placeholder: "Email address" %>
    </div>
    <div>
      <%= f.label :password, "Password", class: "sr-only"\
 %>
      <%= f.password_field :password, autocomplete: "new-\
password", class: "appearance-none rounded-none relative \
block w-full px-3 py-2 border border-gray-300 placeholder\
-gray-500 text-gray-900 focus:outline-none focus:ring-ind\
igo-500 focus:border-indigo-500 focus:z-10 sm:text-sm", p\
laceholder: 'Password' %>
      <%= f.label :password_confirmation, "Confirm Passwo\
rd", class: "sr-only" %>
      <%= f.password_field :password_confirmation, autoco\
mplete: "new-password", class: "appearance-none rounded-n\
one relative block w-full px-3 py-2 border border-gray-30\
0 placeholder-gray-500 text-gray-900 rounded-b-md focus:o\
utline-none focus:ring-indigo-500 focus:border-indigo-500\
```

```
 focus:z-10 sm:text-sm", placeholder: 'Confirm Password' \
%>
    </div>
  </div>
  <% if @minimum_password_length %>
    <em>(<%= @minimum_password_length %> characters minim\
um)</em>
  <% end %><br>
  <div>
    <%= f.submit "Sign Up", class: "group relative w-full\
 flex justify-center py-2 px-4 border border-transparent \
text-sm font-medium rounded-md text-white bg-indigo-600 h\
over:bg-indigo-700 focus:outline-none focus:ring-2 focus:\
ring-offset-2 focus:ring-indigo-500" %>
  </div>
<% end %>
<div class="text-sm">
  <%= render "devise/shared/links" %>
</div>
```

4.2.0.3 Flash Messages

Now, you may have noticed a problem with your sign-in form. If you tried to log in, you would fail due to not having any users created yet. You would be redirected back to the sign-in form with no indication anything had happened. Even though Devise is sending a failure message, there is nothing in the application layout displaying this message.

The following approach will allow you to capture and display any form's response message and display it in the application. There will be three parts to the process. First, an application helper will handle parsing the Rails Session for Flash messages. There will be an added line to the application layout and a snippet of javascript to make the message interactive.

First the `application_helper`:

```
module ApplicationHelper
  def tailwind_color_class_for flash_type
    {
      success: "green",
      error: "red",
      alert: "orange",
      notice: "yellow"
    }[flash_type.to_sym] || flash_type.to_s
  end

  def flash_messages(_opts = {})
    # return if flash[0][0] == :form_error
    flash.each do |msg_type, message|
      next if msg_type == :form_error
      concat(content_tag(:div, message.html_safe, {
        class: "alert transition-all duration-1000 ease-i\
n-out p-4 m-4 mt-5 w-3/5 bg-#{tailwind_color_class_for(ms\
g_type)}-100 border border-#{tailwind_color_class_for(msg\
_type)}-400 text-#{tailwind_color_class_for(msg_type)}-70\
0 rounded-lg",
        role: msg_type}
      ) do
        concat content_tag(:span, message.html_safe, { cl\
ass: "block sm:inline"})
      end)
    end
    nil
  end
end
```

Mainly, this file will parse the type of flash_messages and then build the markup to be inserted dynamically.

Next, create a new file in the new `app/javascript/lib/` folder called `flash.js`.

```
document.addEventListener('turbo:load', function () {

  const alert = document.querySelector('.alert');

  if (alert) {
    setTimeout(() => {
      alert.classList.remove('opacity-100');
      alert.classList.add('opacity-0');
    }, 5000)
    setTimeout(() => {
      alert.style.display = 'none';
    }, 6000)
  }

})
```

To include this file in Rails 7, you will need to add it to the `importmaps` configuration, then import it inside your `application.js`. First, add the following line to `config/importmap.rb`:

```
pin_all_from "app/javascript/lib", under: "lib"
```

Plus, add it to the `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"
import "lib/flash" // new line is here
```

Lastly, we will need to add it to the layout templates and edit one particular TailwindCSS configuration to make sure Tailwind does not skip adding the classes to the compiled CSS (as it normally will purge any unused CSS from its library to save on CSS space). first, the layouts:

```
<!DOCTYPE html>
  <html>
    <head>
    <title>StandupApp</title>
    <meta name="viewport" content="width=device-width,ini\
tial-scale=1">
    <%= csrf_meta_tags %>
    <%= csp_meta_tag %>
    <%= stylesheet_link_tag "tailwind", "inter-font", "da\
ta-turbo-track": "reload" %>

    <%= stylesheet_link_tag "application", "data-turbo-tr\
ack": "reload" %>
    <%= javascript_importmap_tags %>
  </head>

  <body class="bg-gray-800 font-sans leading-normal track\
ing-normal mt-12">

    <%= render partial: 'layouts/components/header' %>
    <div class="flex flex-col md:flex-row">
      <%= render partial: 'layouts/components/sidebar' %>
      <div class="main-content flex-1 bg-gray-100 mt-12 m\
d:mt-2 pb-24 md:pb-5">
        <div class="flex items-center justify-center">
          <%= flash_messages %>
        </div>
        <div class="flex flex-wrap p-3">
          <%= yield %>
        </div>
        <%= render partial: 'layouts/components/footer' %>
      </div>
    </div>
  </body>
</html>
```

and

```
<!DOCTYPE html>
  <html class="h-full">
    <head>
    <title>StandupApp</title>
    <meta name="viewport" content="width=device-width,ini\
tial-scale=1">
    <%= csrf_meta_tags %>
    <%= csp_meta_tag %>
    <%= stylesheet_link_tag "tailwind", "inter-font", "da\
ta-turbo-track": "reload" %>

    <%= stylesheet_link_tag "application", "data-turbo-tr\
ack": "reload" %>
    <%= javascript_importmap_tags %>
  </head>

  <body class="bg-gray-200 h-full">
    <div class="flex items-center justify-center">
      <%= flash_messages %>
    </div>
    <div class="min-h-full flex items-center justify-cent\
er">
      <div class="bg-neutral-50 shadow-md rounded-sm bord\
er m-auto min-h-full flex items-center justify-center py-\
12 px-4 sm:px-6 lg:px-8">
        <div class="max-w-md w-full space-y-8">
          <%= yield %>
        </div>
      </div>
    </div>
  </body>
</html>
```

Lastly, you will want to add a `safelist` to the

`app/config/tailwind.config.js` to look like this:

```
const defaultTheme = require('tailwindcss/defaultTheme')

module.exports = {
  safelist: [
              "bg-orange-100", "border-orange-400", "text\
-orange-700",
              "bg-red-100", "border-red-400", "text-red-7\
00",
              "bg-yellow-100", "border-yellow-400", "text\
-yellow-700",
              "bg-green-100", "border-green-400", "text-g\
reen-700",
            ],
  content: [
    './app/helpers/**/*.rb',
    './app/javascript/**/*.js',
    './app/views/**/*.{erb,haml,html,slim}'
  ],
  theme: {
    extend: {
      fontFamily: {
        sans: ['Inter var', ...defaultTheme.fontFamily.sa\
ns],
      },
    },
  },
  plugins: [
    require('@tailwindcss/forms'),
    require('@tailwindcss/aspect-ratio'),
    require('@tailwindcss/typography')
  ]
}
```

Now when you go to http://localhost:3000/users/sign_in[4] and submit the form, you will see the flash notice on the screen:

So let's stop and commit the changes to git.

```
git add .
git commit -am "Devise Sign Up and error helper"
git push
```

4.2.0.4 Accounts

Before digging into the inviting-user section, you are presented with your first model relationship in your Rails application. There are multitudes of different Model relationships in Rails to go along with the table relationships you would find in a relational database. The main two that are most used are `:has_many` and `:belongs_to`. However, there are also `:has_one` and `:has_many :through` relationships that help ActiveRecord deal with nuances in relational data structures.

In the case of the application you're building in this book, the `User` model will relate to many other models. For example, it will belong to an `Account`. Likewise, `Users` will have many `Tasks` and `Roadblocks`. You will even see how a `User` can have many `Teams` through an intermediate model called `TeamMemberships`.

For the purpose of this section and continuing to build your User portion of the application, you will now work on the `Account` model.

However, you can't create a migration for the `User` model as the generated migration file will try and set up a foreign key to an accounts table that doesn't exist. So the next step will be setting up the 'Account' model.

[4] http://localhost:3000/users/sign_in

```
rails g model Account name addr1 addr2 city state zip cou\
ntry settings:jsonb
```

You may have noticed a new type. In PostgreSQL, you can actually save JSON as a column type. This means you can have schema-less columns on a particular model, just like some NoSQL databases such as MongoDB would. This comes in handy when you may have undefined and evolving settings attributes. However, to finish this migration with all of the needed settings, two manual changes to the migration file are required. As the changes are not able to be specified through the Rails command.

```
class CreateAccounts < ActiveRecord::Migration[7.0]
  def change
    create_table :accounts, id: :uuid do |t|
      t.string :name
      t.string :addr1
      t.string :addr2
      t.string :city
      t.string :state
      t.string :zip
      t.string :country
      t.jsonb :settings, null: false, default: {} # Add n\
ot null and default

      t.timestamps
    end
    add_index :accounts, :settings, using: :gin # add GIN\
 index
  end
end
```

The highlighted lines are modified or added. Line 12 adds `null:false, default: '{}'` database-level constraints. This means there will always be an empty JSON object at the least

for the column. `add_index :accounts, :settings, using: :gin` allows Rails to use a particular JSON ready index to make lookups and queries against the `jsonb` column quick.

For a `belongs_to` relationship to work, the existing model will need an extra column that references the parent model's id. This is actually done relatively quickly with another Rails migration generator.

```
rails g migration AddAccountIdToUsers account:references
```

After, you will want to edit the migration file to remove the `null` setting and add an `index`

```
class AddAccountIdToUsers < ActiveRecord::Migration[7.0]
  def change
    add_reference :users, :account, index: true, foreign_\
key: true, type: :uuid
  end
end
```

Let's make sure all database changes are reflected in your actual development database by running `rails db:migrate`.

Now, for the instances of the objects to understand there is a relationship, there is one line you will need to add to each model.

Add `belongs_to :account, optional: true` in the `app/models/user.rb` model. The `optional: true` is added to this relationship as an account may not exist when a user is first created. If this option was left off, Rails would try and validate an Account exists when a user is created. Next, add `has_many :users` in the `app/models/account.rb` model.

There is one last piece to tie the new `Account` and `User` relationship to the signup process. After a new user signs up, an account will need to be created and the user related to that account.

While the description of what needs to be done is simple, it will require a few necessary files to be added. This will continue to improve your understanding of Rails' conventions of paradigms.

The first new file will be an `Accounts` controller. This file will begin with two action methods. The `new` action will render the form to capture the new account's name, and the `create` action, which will handle the saving of the new `Account` record. This file will reside in `app/controllers/` as a controller. We will be creating this file manually and will not use a generator or scaffold. Here is what the Accounts Controller will look like:

```
class AccountsController < ApplicationController
  def new
    redirect_to root_path unless current_user.account.nil?

    @account = Account.new
  end

  def create
    @account = Account.new(account_params)
    if @account.save
      current_user.account = @account
      current_user.save
      redirect_to root_path, success: "Your account has b\
een created!"
    else
      render :new
    end
  end

  private
  def account_params
    params.require(:account).permit(:name, :addr1, :addr2\
, :city, :state,
      :zip, :country)
```

```
  end
end
```

Let's go over each method individually. First, the method will redirect the user to the `root_path` if there is already an account. This redirect will protect a user from entering this URL manually after an account has been created. Next, the `new` method creates a new empty `Account` model object. This will allow the Rails view `form_with` helper to have access to an object to help build the form. This same pattern will also enable Rails to fill in existing values into forms on `edit` pages.

Next is the `create` action. An `Account` model object is once again created, but this time with the parameters sent from the form. The account object will be saved to the database and, when successful, will relate the new account to the currently logged-in user. Finally, the action will redirect the user to the main path previously set in the `routes.rb` file. If the record fails to save, the user will be redirected back to the form with the information regarding what errors occurred.

Of course, there is no reason for the record not to save. There are no validations. This is a good point to add a simple validation to guarantee a name field on every `Account` record. Here is the updated `app/model/account.rb` file:

```
class Account < ApplicationRecord
  has_many :users
  validates :name, presence: true
end
```

Now, back to the `AccountsController`. The latter method on the controller is part of the Rails Strong Parameters system. This system allows you to allowlist parameters accepted from a form. Meaning that someone can't construct another form on their machine to submit malicious parameters.

One last piece to add to make the controller method accessible in your application is to add a line to the `config/routes.rb` file to create the default resource Routes:

```
Rails.application.routes.draw do
  devise_for :users
  resource :accounts

  get 'activity/mine'

  get 'activity/feed'

  root to: 'activity#mine'
end
```

Rails use that line to automatically create the needed RESTful Routes for the account resource. This is different than the more commonly used `resources :thing` route line that will generate all of the RESTful Routes. You can run the command line Routes tool to see what routes were added.

```
rails routes
```

Once the command finishes running, it will show you all of the configured routes for your application. Here you can see the routes `resource :accounts` created:

```
new_accounts GET    /accounts/new(.:format)        accoun\
ts#new
edit_accounts GET    /accounts/edit(.:format)       accou\
nts#edit
    accounts GET    /accounts(.:format)            accoun\
ts#show
             PATCH  /accounts(.:format)            accoun\
ts#update
             PUT    /accounts(.:format)            accoun\
ts#update
             DELETE /accounts(.:format)            accoun\
ts#destroy
             POST   /accounts(.:format)            accoun\
ts#create
```

Here is a helpful place to add some tests that show the added functionality from the model and controller work! First, a test on the model. In this file, you will check the factory/model is valid and that an `Account` model object will not save without a `name`.

```
require 'rails_helper'

RSpec.describe Account, type: :model do
  context "valid Factory" do
    it "has a valid factory" do
      expect(FactoryBot.build(:account)).to be_valid
    end
  end

  context "valid validations" do
    it "will not save without a name" do
      expect(FactoryBot.build(:account, name: nil).save).\
to be_falsey
    end
  end
end
```

If you run `rspec`, the new model specs added pass, but now the controller specs are failing. This may seem like it is for no reason, but if you read the error message, you will see it is related to not having Devise-based spec support classes included in the test suite. You can add two lines to the `spec/rails_helper.rb` inside the `RSpec.configure do |config|` block, like this:

```
RSpec.configure do |config|
  config.include Devise::Test::IntegrationHelpers, type: \
:request
  config.include Devise::Test::ControllerHelpers, type: :\
view
```

If you then run `rspec` once more, you will see, the controller specs are still failing, although now, with a different error:

```
expected the response to have a success status code (2xx)\
 but it was 302
```

This may seem cryptic, but if you think about how the sign-in and authentication work in Devise, it makes quite a bit of sense. When you attempt to access a controller action that is behind a controller using Devise authentication (such as our case where the authentication `before_action :authenticate_user!` in the `ApplicationController` means all actions will get the authentication treatment) and the browser session is not authenticated it is redirected to the sign-in screen. Thus, the spec failure makes sense; the spec's session is not authenticated and is redirected instead of performing the expected action.

With that discovery out of the way, we can move toward fixing the spec. First, we will add code to log in to a session within the controller specs. We will start by adding a new support file (you may need to create the `spec/support/` folder as well) to log in a user:

```
module RequestMacros
  def login_user
    before(:each) do
      @user = FactoryBot.create(:user)
      sign_in @user
    end
  end
end
```

This file will set a spec mapping for Devise, create a User record and log in the user that was just created. Now, we just need to add a few lines to the RSpec `rails_helper` and the controller specs to continue.

First, let's add the following lines right after `require 'rspec/rails'` in the `spec/rails_helper.rb` file.

```
require 'devise'
require 'support/request_macros'
```

In the same file, after `config.include Devise::Test::ControllerHelpers, type: :view` add this line:

```
config.extend RequestMacros, type: :request
```

Lastly, to make sure a spec uses a logged-in user, you can include `login_user` within any spec. A shortcut to using it on any spec will be to include it at the beginning of the file:

```
require 'rails_helper'

RSpec.describe "Activities", type: :request do
  login_user

  describe "GET /mine" do
    it "returns http success" do
      get "/activity/mine"
      expect(response).to have_http_status(:success)
    end
  end

  describe "GET /feed" do
    it "returns http success" do
      get "/activity/feed"
      expect(response).to have_http_status(:success)
    end
  end

end
```

Now, if you finally rerun the specs, you will see they all pass.

```
rspec
.......

Finished in 25.94 seconds (files took 7.07 seconds to loa\
d)
7 examples, 0 failures
```

Now, let's test the new `Accounts` controller. This will start off by adding a new `accounts_spec.rb` file in `spec/requests` with the following content:

```
require 'rails_helper'

RSpec.describe "Accounts", type: :request do
  login_user

  describe "GET #new" do
    it "returns http success" do
      get '/accounts/new'
      expect(response).to have_http_status(:success)
    end

    it "renders new template" do
      get '/accounts/new'
      expect(response).to render_template :new
    end
  end

  describe "POST #create" do
    it "creates an account" do
      post '/accounts', params: { account: FactoryBot.att\
ributes_for(:account)}
      expect(response).to redirect_to root_path
    end

    it "renders :new on failure" do
      post '/accounts',  params: {account: FactoryBot.att\
ributes_for(:account,{
        name: nil
      })}
      expect(response).to render_template :new
    end
  end

end
```

The great thing about RSpec is that you can usually take someone

else's tests and understand what is being tested. So even if you aren't entirely familiar with the RSpec syntax in this example, both the new and create actions are tested for their template rendering and behavior.

Once again, run `rspec`. I bet you were expecting it this time. More failing specs. Still, RSpec and Rails make the failure and errors very clean by stating that there are no matching template files. Well, yes, that makes sense! We haven't created templates for the new `AccountsController` and its `new` action.

For the ' New Account ' form, let's add the `app/views/account/new.html.erb` template file. Also, you will need to add an `accounts` folder in `app/views` if it does not exist.

```
<div class="flex p-5" aria-label="Breadcrumb">
  <ol class="inline-flex items-center space-x-1 md:space-\
x-3">
    <li class="inline-flex items-center">
      <h2 href="#" class="inline-flex items-center font-m\
edium text-gray-700 hover:text-gray-900">
        Account
      </h2>
    </li>
    <li class="inline-flex items-center">
      <h2 href="#" class="inline-flex items-center font-m\
edium text-gray-700 hover:text-gray-900">
        <svg class="w-6 h-6 text-gray-400" fill="currentC\
olor" viewBox="0 0 20 20" xmlns="http://www.w3.org/2000/s\
vg"><path fill-rule="evenodd" d="M7.293 14.707a1 1 0 010-\
1.414L10.586 10 7.293 6.707a1 1 0 011.414-1.414l4 4a1 1 0\
 010 1.414l-4 4a1 1 0 01-1.414 0z" clip-rule="evenodd"></\
path></svg>
        New
      </h2>
    </li>
  </ol>
```

```
</div>
<div class="flex justify-center">
  <div class="p-4 w-4/5 bg-white rounded-lg border border\
-gray-200 shadow-md sm:p-6 lg:p-8">
    <%= form_for @account, html: {class: 'space-y-6'} do \
|f| %>
      <% if @account.errors.any? %>
        <div class="text-red-700">
          <h4>
            <%= "#{pluralize(@account.errors.count, "erro\
r")} prohibited this account from being saved:" %>
          </h4>
          <%== @account.errors.full_messages.map{|msg|con\
tent_tag(:li, msg)}.join %>
        </div>
      <% end %>
      <div>
        <%= f.label :name, class: 'block mb-2 text-sm fon\
t-medium text-gray-900' %>
        <%= f.text_field :name, placeholder: 'Enter your \
Account\'s Name', class: 'bg-gray-50 border border-gray-3\
00 text-gray-900 text-sm rounded-lg focus:ring-blue-500 f\
ocus:border-blue-500 block w-full p-2.5' %>
      </div>
      <%=f.submit "Save", class: "text-white bg-blue-700 \
hover:bg-blue-800 focus:ring-4 focus:outline-none focus:r\
ing-blue-300 font-medium rounded-lg text-sm w-full sm:w-a\
uto px-5 py-2.5 text-center" %>
    <% end %>
  </div>
</div>
```

The template provides a pretty standard web form with one text box for an `Account Name`. There is also an area to display errors, like the `Devise Sign Up` form.

However, this form is not exactly accessible after the `Devise Sign Up` is successfully submitted. This is where you will be making overrides to the Devise `RegistrationsController`. Devise will allow you to override their controller settings by creating a new controller file that inherits the Controller file from Devise. Then, you would only add the method you would like to override. If you change the redirect path after successful sign-up, you will simply need to add a method `after_sign_up_path_for`. You will need to create a `registrations_controller.rb` file in `app/controllers` with the following code:

```
class RegistrationsController < Devise::RegistrationsCont\
roller
  before_action :configure_permitted_parameters

  protected

  def configure_permitted_parameters
    devise_parameter_sanitizer.permit(:sign_up, keys: [:n\
ame])
  end

  def after_sign_up_path_for(resource)
    new_accounts_path
  end
end
```

Going over this file:

- Line 1: Allows the newly created `RegistrationsController` to inherit the methods from `Devise::RegistrationsController`
- Line 2: States a method `configure_permitted_parameters` be called before an action in this controller is performed.
- Line 4: This is a Ruby keyword that says only the current class or children classes can access the method. Without

going much more in-depth, this is simply an Object-Oriented programming concept to protect Objects from one another.
- Lines 6-8: This method allows Devise to accept a non-default parameter through Strong Parameters.
- Lines 10-12: This method changes the post successful signup redirect. Line 11 is explicitly the path helper who would basically equate to `accounts/new`. Which happens to be the route to the form you just created.

Finally, to make sure both your application and Devise uses your `RegistrationsController` controller, instead of the default Controller, you will modify the `devise` line of the `config/routes.rb` from:

```
devise_for :users
```

to

```
devise_for :users, controllers: { registrations: "registr\
ations" }
```

4.2.0.5 System Spec Time

Okay, let's create some System specs to ensure the Devise forms, Controllers, and Views set up in this chapter are working. There will be two System specs to set up. One for the `Sign In` interaction and another for the whole `Sign Up` process.

To create a system spec for the `Sign In` process, you will need to create a file to hold the scenarios to test. RSpec actually has a command-line tool to start this file for use:

```
rails g rspec:system sign_in_process
```

Now you can open that newly created system spec and add the following content:

```
require 'rails_helper'

RSpec.describe "SignInProcesses", type: :system do
  before do
    driven_by(:rack_test)
  end

  let(:password) { '123456789' }
  let(:user) do
    FactoryBot.create( :user, {
      password: password,
      password_confirmation: password
    })
  end
  it "should require the user to log in and successfully \
logs in" do
    visit root_path

    within "#new_user" do
      fill_in "user_email", with: user.email
      fill_in "user_password", with: password
    end

    click_button "Log in"

    expect(current_path).to eql(root_path)
  end

  it "should fail on an invalid user" do

    visit root_path

    within "#new_user" do
      fill_in "user_email", with: 'test'
      fill_in "user_password", with: 'password'
```

```
    end

    click_button "Log in"

    expect(current_path).to eql(new_user_session_path)
  end
end
```

The striking thing about Rails System tests that use Capybara is that even if you don't know the exact syntax used in this file, you can still follow along and understand what is being tested.

```
rspec spec/system/sign_in_processes_spec.rb

..

Finished in 3.56 seconds (files took 1.97 seconds to load)
2 examples, 0 failures
```

Finally, let's create the system spec for signing up. Create this file through the RSpec command tool as well:

```
rails g rspec:system sign_up_process
```

Again, open that newly created system spec and add the following content:

```
require 'rails_helper'

RSpec.describe "SignUpProcesses", type: :system do
  before do
    driven_by(:rack_test)
  end

  it "should require the user to sign up and successfully\
 sign up" do

    visit root_path

    click_on 'Sign up'

    within "#new_user" do
      fill_in "user_email", with: 'test@test.com'
      fill_in "user_password", with: 'password123'
      fill_in "user_password_confirmation", with: 'passwo\
rd123'
    end

    click_button "Sign Up"

    expect(current_path).to eql(new_accounts_path)

    within "#new_account" do
      fill_in "account_name", with: "Test Co"
    end

    click_button "Save"

    expect(current_path).to eql(root_path)
  end

  it "should fail on invalid user information" do
```

```
    visit root_path

    click_on 'Sign up'

    within "#new_user" do
      fill_in "user_email", with: 'test'
      fill_in "user_password", with: 'p'
    end

    click_button "Sign Up"

    expect(current_path).to eql('/users')
    expect(page).to have_content("error")
  end

  it "should fail on invalid account information" do

    visit root_path

    click_on 'Sign up'

    within "#new_user" do
      fill_in "user_email", with: 'test2@test.com'
      fill_in "user_password", with: 'password123'
      fill_in "user_password_confirmation", with: 'passwo\
rd123'
    end

    click_button "Sign Up"

    within "#new_account" do
      fill_in "account_name", with: ""
    end
```

```
    click_button "Save"
    expect(current_path).to eql('/accounts')
    expect(page).to have_content("error")
  end
end
```

ow, you can run rspec on that file:

```
rspec spec/system/sign_up_processes_spec.rb
...

Finished in 2.65 seconds (files took 7.43 seconds to load)
3 examples, 0 failures
```

Then on all spec files:

```
rspec
................

Finished in 3.43 seconds (files took 7.75 seconds to load)
16 examples, 0 failures
```

Everything should be passing! Again, it's suggested that you commit and push your code.

4.3 User Management

Now that users can sign up and access your application, let's give them an interface to view and manage other users. Another default Rails and Restful controller action will be introduced in this section, the `index` action. The `index` action is responsible for fetching and displaying a list of objects. In this case, it will be a list of `Users` that the current logged-in user has access to.

Once more, Rails helps us out with a command-line tool to help generate the scaffold of the files we need. Plus, it will create all of the auxiliary files, such as test files. Run the following command to produce a `UsersController` with the action methods stubbed out, default views, and stubbed out test files:

```
rails generate controller Users index show new create edi\
t update
```

Note that some of the generated files are not exactly how you will need them and will be edited and updated in the following sections. Before updating the actions in the `UsersController`, it is an excellent time to add the `User` invite system and `User` roles system.

4.3.1 User Invites

With the `devise_invitable` gem already installed, you can use the following commands to install the invite functionality to your application:

```
rails generate devise_invitable:install
rails generate devise_invitable User
rails generate devise_invitable:views
rails db:migrate
```

Those commands will update existing Devise settings and files. It will also create new attributes for the `users` table and generate view files to be edited into the style of your theme.

Rolify, like other third-party libraries and gems that interact with models/tables, are not ready "out of the box" to work with UUIDs. However, a quick edit to the migration that was just created will add the UUID settings to the new table and references.

```
class RolifyCreateRoles < ActiveRecord::Migration[7.0]
  def change
    create_table(:roles, id: :uuid) do |t|
      t.string :name
      t.references :resource, type: :uuid, :polymorphic =\
> true
      t.timestamps
    end

    create_table(:users_roles, :id => false) do |t|
      t.references :user, type: :uuid
      t.references :role, type: :uuid
    end

    add_index(:roles, [ :name, :resource_type, :resource_\
id ])
    add_index(:users_roles, [ :user_id, :role_id ])
  end
end
```

A quick `rails db:migrate` will ensure these changes are reflected in your database.

The install command handles nearly every part of Rolify that you need. The new Role model, model associations, and the role functionality into the `User` model. The only piece you will need to add manually is making the `Account` model available as a role resource. This means that you can scope roles for a `User` within an `Account`. Add the following to line 2 of the `app/models/account.rb` file:

```
resourcify
```

You can also add a default role to a user signing up through the primary Devise sign-up form. This can be done on one line within the `AccountsController` (the whole file is omitted for brevity):

```
...
if @account.save
  current_user.account = @account
  current_user.add_role :admin, @account # <-- new line h\
ere
  current_user.save
  redirect_to root_path, success: "Your account has been \
created!"
...
```

This completes the Rolify setup. You will actually be reading and writing user roles in the next section.

4.3.2 Managing the Users

OK. Back to managing the users within an account. You almost have every piece of auxiliary code in place to facilitate the `UsersController`. This may seem like a lot but will pay off as you get deeper into your application and keep reusing what is built in this chapter.

You will add a method to the `ApplicationController` to facilitate looking up the current account for the currently logged-in user. Meaning, that in your application, you will be able to call `current_account` to get the account object. This will be very helpful in scoping resources down, such as users or tasks.

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

  before_action :authenticate_user!
  layout :layout_by_resource

  ## New Code Starts Here ##
  helper_method :current_account

  def current_account
    @current_account ||= current_user.account
    @current_account
  end
  ## New Code Ends Here ##

  protected
  def layout_by_resource
    if devise_controller?
      "devise"
    else
      "application"
    end
  end
end
```

The code inside the method may seem a little weird. This method is actually using a technique called 'memoize'. Basically, it means that if the instance variable hasn't been set, set the variable to the current_user's account. On subsequent calls to the method in the same HTTP request, the variable will have been set and return the object without another [SQL] lookup.

Yes! Finally, the long-awaited `UsersController`. Here will be all of the methods needed for managing other users and the current user. Following the controller will be the template files for the added methods.

```
class UsersController < ApplicationController
  before_action :set_user, only: [:show, :edit, :update, \
:destroy]
  before_action :set_users, only: [:index]

  def index

  end

  def new
    @user = User.new
    set_choices
  end

  def edit
    set_choices
  end

  def me
    @user = current_user
  end

  def password
    @user = current_user
  end

  def update_password
    @user = current_user
    respond_to do |format|
      if @user.update(user_password_params)
        bypass_sign_in(@user)
        format.html {
          redirect_to my_password_path,
          notice: 'Password was successfully updated.'
        }
```

```
      else
        format.html { render :password }
      end
    end
  end

  def update_me
    @user = current_user
    respond_to do |format|
      if @user.update(user_params)
        format.html {
          redirect_to my_settings_path,
          notice: 'Your information was successfully upda\
ted.'
        }
      else
        format.html { render :me }
      end
    end
  end

  def create
    @user = User.unscoped.new(user_params.except("role"))
    @user.account = current_account
    @user.password = "password123"

    respond_to do |format|
      begin
        if @user.valid? &&  @user.invite!(current_user)
          @user.add_role user_params[:role].to_sym, curre\
nt_account
          format.html {
            redirect_to account_users_path,
            notice: 'User was successfully invited.'
          }
```

```
      else
        set_choices
        format.html { render :new }
      end
    rescue ActiveRecord::RecordNotUnique
      flash[:alert]= 'Email must be unique'
      format.html { render :new}
    end
  end
end

def update
  respond_to do |format|
    if @user.update(user_params.except('role'))
      update_roles
      format.html {
        redirect_to account_users_path,
        notice: 'User was successfully updated.'
      }
    else
      set_choices
      format.html { render :edit }
    end
  end
end

def destroy
  @user.destroy
  respond_to do |format|
    format.html {
      redirect_to account_users_path,
      notice: 'User was successfully destroyed.'
    }
    format.json { head :no_content }
  end
```

```
  end

  private

    # Use callbacks to share common setup or constraints \
between actions.
    def set_users
      @users = current_account.users
    end

    def set_user
      @user = User.find(params[:id])
    end

    def set_choices
      @choices = [["Admin", 'admin'],["User", 'user']]
    end

    def user_params
      params.require(:user).permit(:name, :email, :role, \
:time_zone)
    end

    def user_password_params
      params.require(:user).permit(:password, :password_c\
onfirmation)
    end

    def update_roles
      if @user.roles&.first&.name != user_params[:role]
        @user.remove_role @user.roles&.first&.name.to_sym
        @user.add_role user_params[:role].to_sym, current\
_account
      end
    end
```

```
end
```

A few things to note here. The current Rails convention for setting instance variables with shared behavior is using private methods called from a `before_action`. In this controller's case, it sets either the collection of users on the `index` action or the current user in others. Another method that may have caught your eye is `set_choices`. There will be a dropdown for the user's role in the templates. This method just collapses that logic into one method to be shared. Lastly, `me`, `password`, `update_password`, and `update_me` are new methods not mentioned in the command line generator but will be used to update aspects of a user.

Additionally, let's break down two of the more substantial, more complicated methods. First, the create method:

```
def create
  @user = User.unscoped.new(user_params.except("role"))
  @user.account = current_account
  @user.password = "password123"

  respond_to do |format|
    begin
      if @user.valid? &&  @user.invite!(current_user)
        @user.add_role user_params[:role].to_sym, current\
_account
        format.html {
          redirect_to account_users_path,
          notice: 'User was successfully invited.'
        }
      else
        format.html { render :new }
      end
    rescue ActiveRecord::RecordNotUnique
      flash[:alert]= 'Email must be unique'
      format.html { render :new}
```

```
    end
  end
end
```

This method sets up an object with the parameters from the form. Though, it excludes the role from that object's creation. The role itself is not an attribute of the method but a model association. It connects the new user to the `current_account` and sets a dummy password. Devise will not actually use this password but will have the invited user create a password upon signing up. The `respond_to` block will allow you to use and return different content types, but in this case, it will just be HTML. The rest of the block is wrapped in a `begin...rescue..end` block. Due to the uniqueness being constrained at the database level, you will need to catch the error and handle the response appropriately. Lastly, back on the main portion of that block of code. If the user is valid(passes model validations) and is invited, the previously ignored role param is used to set a role. With a " success " method, the current user is redirected back to the `index` page with a "success" method.

Lastly, let's go over the `update_password` method:

```
def update_password
  @user = current_user
  respond_to do |format|
    if @user.update(user_password_params)
      bypass_sign_in(@user)
      format.html {
        redirect_to my_password_path,
        notice: 'Password was successfully updated.'
      }
    else
      format.html { render :password }
    end
  end
end
```

The area to point out here is the `bypass_sign_in(@user)`. The session is typically invalidated when updating the password, and Devise would log you out. However, including this line in the success path, it allows the current user to stay logged in when updating their password.

There is one little bit of "generator cleanup" before working on the views. The route file was populated with some generic `get` routes from the generator that you can now fix to be specific to your application.

```
Rails.application.routes.draw do
  devise_for :users, controllers: { registrations: "regis\
trations" }
  resource :accounts

  ### Changes Start Here ##
  get 'user/me', to: 'users#me', as: 'my_settings'
  patch 'user/update_me', to: 'users#update_me', as: 'upd\
ate_my_settings'
  get 'user/password', to: 'users#password', as: 'my_pass\
word'
  patch 'user/update_password', to:'users#update_password\
', as:'my_update_password'

  scope 'account', as: 'account' do
    resources :users
  end
  ### Changes End Here ##

  get 'activity/mine'

  get 'activity/feed'

  root to: 'activity#mine'
end
```

Finally, it's here! You will write some markup and see the fruit of your labor on your locally running rails server. The following content will go into the previously created `app/views/users/index.html.erb`:

```
<div class="flex p-5" aria-label="Breadcrumb">
  <ol class="inline-flex items-center space-x-1 md:space-\
x-3">
    <li class="inline-flex items-center">
      <h2 href="#" class="inline-flex items-center font-m\
edium text-gray-700 hover:text-gray-900">
        Users
      </h2>
    </li>
  </ol>
</div>
<div class="flex justify-center">
  <div class="p-4 w-4/5 bg-white rounded-lg border border\
-gray-200 shadow-md sm:p-6 lg:p-8">
    <h5 class="text-xl font-bold leading-none text-gray-9\
00"><%= "Users in #{current_account.name}" %></h5>
    <table class="w-full text-sm text-left text-gray-500">
      <thead class="text-xs text-gray-700 uppercase bg-gr\
ay-50">
          <tr>
              <th scope="col" class="px-6 py-3">
                  Name
              </th>
              <th scope="col" class="px-6 py-3">
                  Email
              </th>
              <th scope="col" class="px-6 py-3">
                  Status
              </th>
              <th scope="col" class="px-6 py-3">
                  Role
```

```
              </th>
              <th scope="col" class="px-6 py-3">
                  Added at
              </th>
              <th scope="col" class="px-6 py-3">
              </th>
          </tr>
      </thead>
      <tbody>
        <% @users.each do |user|%>
          <tr class="bg-white border-b">
            <th scope="row" class="px-6 py-4 font-medium \
text-gray-900 whitespace-nowrap"><%= user.name %></th>
            <td class="px-6 py-4"><%= user.email %></td>
            <td class="px-6 py-4"><%= user.accepted_or_no\
t_invited? ? "Active" : "Invited" %></td>
            <td class="px-6 py-4"><%= user.roles&.first&.\
name&.capitalize %></td>
            <td class="px-6 py-4"><%= user.created_at.str\
ftime('%D') %></td>
            <td class="px-6 py-4">
              <%= link_to 'Edit', edit_account_user_path(\
user), class: 'text-blue-700 hover:text-white border bord\
er-blue-700 hover:bg-blue-800 focus:ring-4 focus:outline-\
none focus:ring-blue-300 font-medium rounded-lg text-sm p\
x-5 py-2.5 text-center mr-2 mb-2' %>
              <%= link_to 'Destroy', account_user_path(us\
er), {method: :delete, data: { confirm: 'Are you sure?' }\
, class: 'text-red-700 hover:text-white border border-red\
-700 hover:bg-red-800 focus:ring-4 focus:outline-none foc\
us:ring-red-300 font-medium rounded-lg text-sm px-5 py-2.\
5 text-center mr-2 mb-2'} unless user.id == current_user.\
id %>
            </td>
          </tr>
```

```
        <% end %>
      </tbody>
    </table>
    <%= link_to "Add User", new_account_user_path, class:\
 "inline-flex mt-4 text-white bg-blue-700 hover:bg-blue-8\
00 focus:ring-4 focus:outline-none focus:ring-blue-300 fo\
nt-medium rounded-lg text-sm w-full sm:w-auto px-5 py-2.5\
 text-center" %>
  </div>
</div>
```

Now that you are used to the `.erb` templating language, this is a pretty simple template. We are just iterating over the users fetched in the `UsersController` for the `current_account`. This is done within the `card` classes provided by the theme and a table where every row is a user. A quick look at http://localhost:3000/account/users[5] should show a table with one user, yourself.

Now, to jump right into the new user template at `app/views/users/new.html.erb`:

```
<div class="flex p-5" aria-label="Breadcrumb">
  <ol class="inline-flex items-center space-x-1 md:space-\
x-3">
    <li class="inline-flex items-center">
      <h2 href="#" class="inline-flex items-center font-m\
edium text-gray-700 hover:text-gray-900">
        Users
      </h2>
    </li>
    <li class="inline-flex items-center">
      <h2 href="#" class="inline-flex items-center font-m\
edium text-gray-700 hover:text-gray-900">
```

[5] http://localhost:3000/account/users

```
        <svg class="w-6 h-6 text-gray-400" fill="currentC\
olor" viewBox="0 0 20 20" xmlns="http://www.w3.org/2000/s\
vg"><path fill-rule="evenodd" d="M7.293 14.707a1 1 0 010-\
1.414L10.586 10 7.293 6.707a1 1 0 011.414-1.414l4 4a1 1 0\
 010 1.414l-4 4a1 1 0 01-1.414 0z" clip-rule="evenodd"></\
path></svg>
        New
      </h2>
    </li>
  </ol>
</div>
<div class="flex justify-center">
  <div class="p-4 w-4/5 bg-white rounded-lg border border\
-gray-200 shadow-md sm:p-6 lg:p-8">
    <h5 class="text-xl font-bold leading-none text-gray-9\
00">Add a new user:</h5>
    <%= form_with(model: @user, url: account_users_path, \
id: 'new_user', html: {class: 'space-y-6'})  do |f| %>
      <% if @user.errors.any? %>
        <div class="text-red-700">
          <h4>
            <%= "#{pluralize(@user.errors.count, "error")\
} prohibited this user from being saved:" %>
          </h4>
          <%== @user.errors.full_messages.map{|msg|conten\
t_tag(:li, msg)}.join %>
        </div>
      <% end %>
      <div>
        <%= f.label :name, class: 'block mb-2 text-sm fon\
t-medium text-gray-900' %>
        <%= f.text_field :name, placeholder: 'Name', clas\
s: 'bg-gray-50 border border-gray-300 text-gray-900 text-\
sm rounded-lg focus:ring-blue-500 focus:border-blue-500 b\
lock w-full p-2.5' %>
```

```
      </div>
      <div>
        <%= f.label :email, class: 'block mb-2 text-sm fo\
nt-medium text-gray-900' %>
        <%= f.email_field :email, placeholder: 'Enter the\
ir email address', class: 'bg-gray-50 border border-gray-\
300 text-gray-900 text-sm rounded-lg focus:ring-blue-500 \
focus:border-blue-500 block w-full p-2.5' %>
      </div>
      <div>
        <%= f.label :role, class: 'block mb-2 text-sm fon\
t-medium text-gray-900' %>
        <%= f.select :role, options_for_select(@choices, \
'user'), {}, class: 'bg-gray-50 border border-gray-300 te\
xt-gray-900 text-sm rounded-lg focus:ring-blue-500 focus:\
border-blue-500 block w-full p-2.5' %>
      </div>
      <%=f.submit "Add User", class: "text-white bg-blue-\
700 hover:bg-blue-800 focus:ring-4 focus:outline-none foc\
us:ring-blue-300 font-medium rounded-lg text-sm w-full sm\
:w-auto px-5 py-2.5 text-center" %>
    <% end %>
  </div>
</div>
```

Here is another standard form. This time it captures the invited user's name, email address, and user role. For example, head to http://localhost:3000/account/users/new[6], you should see the new user form.

If you are the curious developer I am, you have probably already tried to submit the new user. You will have found it worked, but no email has been sent anywhere. This is because the line to use the `letter_opener` gem has not been added to your development environment. You can resolve this by adding the following lines to

[6] http://localhost:3000/account/users/new

the second to last line in the `config/environments/development.rb` file.

```
config.action_mailer.delivery_method = :letter_opener
config.action_mailer.asset_host = "http://localhost:#{ENV\
['PORT'] || 3000}"
config.action_mailer.default_url_options = {
  :host => "http://localhost:#{ENV['PORT'] || 3000}"
}
```

Then, restart your rails server and fill out the form again.



Did you know you can restart your server from any other terminal window/tab that is is in the same folder by running `bin/rails restart`?

You should now see the email in another tab in your browser. If you were to follow that link in a different browser or incognito tab (since your current session is already logged in), you would see the form to set your password is not styled. OK, you have been here before; let's turn that template, `devise/invitations/edit`, into a `.erb` file with the theme's markup:

```
<h2 class="mt-6 text-center text-3xl font-extrabold text-\
gray-900"> Set Your Password </h2>
<%= form_for(resource, as: resource_name, url: invitation\
_path(resource_name), html: { method: :put }) do |f| %>
  <%= render "devise/shared/error_messages", resource: re\
source %>
  <%= f.hidden_field :invitation_token, readonly: true %>
  <% if f.object.class.require_password_on_accepting %>
    <div class="rounded-md shadow-sm -space-y-px">
      <div>
        <%= f.label :password, "Password", class: "sr-onl\
```

```
y" %>
        <%= f.password_field :password, autocomplete: "ne\
w-password", class: "appearance-none rounded-none relativ\
e block w-full px-3 py-2 border border-gray-300 placehold\
er-gray-500 text-gray-900 rounded-t-md focus:outline-none\
 focus:ring-indigo-500 focus:border-indigo-500 focus:z-10\
 sm:text-sm", placeholder: "Password" %>
      </div>
      <div>
        <%= f.label :password_confirmation, "Confirm Pass\
word", class: "sr-only" %>
        <%= f.password_field :password_confirmation, auto\
complete: "new-password", class: "appearance-none rounded\
-none relative block w-full px-3 py-2 border border-gray-\
300 placeholder-gray-500 text-gray-900 rounded-b-md focus\
:outline-none focus:ring-indigo-500 focus:border-indigo-5\
00 focus:z-10 sm:text-sm", placeholder: 'Confirm Password\
' %>
      </div>
    </div>
  <% end %>
  <div>
    <%= f.submit t("devise.invitations.edit.submit_button\
"), class: "mt-4 group relative w-full flex justify-cente\
r py-2 px-4 border border-transparent text-sm font-medium\
 rounded-md text-white bg-indigo-600 hover:bg-indigo-700 \
focus:outline-none focus:ring-2 focus:ring-offset-2 focus\
:ring-indigo-500" %>
  </div>
<% end %>
```

Perfect, now it matches other Devise forms and allows the new user to log in after submission. We're in the home stretch here. There are only a few more templates needed to manage users.

Next up is the `edit` form, which will be very near but not exactly

like the new form:

```
<div class="flex p-5" aria-label="Breadcrumb">
  <ol class="inline-flex items-center space-x-1 md:space-\
x-3">
    <li class="inline-flex items-center">
      <h2 href="#" class="inline-flex items-center font-m\
edium text-gray-700 hover:text-gray-900">
        Users
      </h2>
    </li>
    <li class="inline-flex items-center">
      <h2 href="#" class="inline-flex items-center font-m\
edium text-gray-700 hover:text-gray-900">
        <svg class="w-6 h-6 text-gray-400" fill="currentC\
olor" viewBox="0 0 20 20" xmlns="http://www.w3.org/2000/s\
vg"><path fill-rule="evenodd" d="M7.293 14.707a1 1 0 010-\
1.414L10.586 10 7.293 6.707a1 1 0 011.414-1.414l4 4a1 1 0\
 010 1.414l-4 4a1 1 0 01-1.414 0z" clip-rule="evenodd"></\
path></svg>
        Edit <%= @user.email %>
      </h2>
    </li>
  </ol>
</div>
<div class="flex justify-center">
  <div class="p-4 w-4/5 bg-white rounded-lg border border\
-gray-200 shadow-md sm:p-6 lg:p-8">
    <h5 class="text-xl font-bold leading-none text-gray-9\
00">Edit User Information:</h5>
    <%= form_with(model: @user, url: account_user_path(@u\
ser), id: 'edit_user', html: {class: 'space-y-6'}) do |f|\
 %>
      <% if @user.errors.any? %>
        <div class="text-red-700">
          <h4>
```

```
            <%= "#{pluralize(@user.errors.count, "error")\
} prohibited this user from being saved:" %>
          </h4>
          <%== @user.errors.full_messages.map{|msg|conten\
t_tag(:li, msg)}.join %>
        </div>
      <% end %>
      <div>
        <%= f.label :name, class: 'block mb-2 text-sm fon\
t-medium text-gray-900' %>
        <%= f.text_field :name, placeholder: 'Name', clas\
s: 'bg-gray-50 border border-gray-300 text-gray-900 text-\
sm rounded-lg focus:ring-blue-500 focus:border-blue-500 b\
lock w-full p-2.5' %>
      </div>
      <div>
        <%= f.label :email, class: 'block mb-2 text-sm fo\
nt-medium text-gray-900' %>
        <%= f.email_field :email, placeholder: 'Enter the\
ir email address', class: 'bg-gray-50 border border-gray-\
300 text-gray-900 text-sm rounded-lg focus:ring-blue-500 \
focus:border-blue-500 block w-full p-2.5' %>
      </div>
      <div>
        <%= f.label :role, class: 'block mb-2 text-sm fon\
t-medium text-gray-900' %>
        <%= f.select :role, options_for_select(@choices, \
@user&.roles&.first&.name), {}, class: 'bg-gray-50 border\
 border-gray-300 text-gray-900 text-sm rounded-lg focus:r\
ing-blue-500 focus:border-blue-500 block w-full p-2.5' %>
      </div>
      <%=f.submit "Edit User", class: "text-white bg-blue\
-700 hover:bg-blue-800 focus:ring-4 focus:outline-none fo\
cus:ring-blue-300 font-medium rounded-lg text-sm w-full s\
m:w-auto px-5 py-2.5 text-center" %>
```

```
    <% end %>
  </div>
</div>
```

The following template to take care of is the page where the current user can update their information. Currently, they would be able to update their name, email address, or time zone. Therefore, you will need to create this new file at `app/views/users/me.html.erb`:

```
<div class="flex p-5" aria-label="Breadcrumb">
  <ol class="inline-flex items-center space-x-1 md:space-\
x-3">
    <li class="inline-flex items-center">
      <h2 href="#" class="inline-flex items-center font-m\
edium text-gray-700 hover:text-gray-900">
        Users
      </h2>
    </li>
    <li class="inline-flex items-center">
      <h2 href="#" class="inline-flex items-center font-m\
edium text-gray-700 hover:text-gray-900">
        <svg class="w-6 h-6 text-gray-400" fill="currentC\
olor" viewBox="0 0 20 20" xmlns="http://www.w3.org/2000/s\
vg"><path fill-rule="evenodd" d="M7.293 14.707a1 1 0 010-\
1.414L10.586 10 7.293 6.707a1 1 0 011.414-1.414l4 4a1 1 0\
 010 1.414l-4 4a1 1 0 01-1.414 0z" clip-rule="evenodd"></\
path></svg>
        My Settings
      </h2>
    </li>
  </ol>
</div>
<div class="flex justify-center">
  <div class="p-4 w-4/5 bg-white rounded-lg border border\
-gray-200 shadow-md sm:p-6 lg:p-8">
```

```
    <h5 class="text-xl font-bold leading-none text-gray-9\
00">Edit My Information:</h5>
    <%= form_with(model: @user, url: update_my_settings_p\
ath, id: 'edit_my_user', html: {class: 'space-y-6'}) do |\
f| %>
      <% if @user.errors.any? %>
        <div class="text-red-700">
          <h4>
            <%= "#{pluralize(@user.errors.count, "error")\
} prohibited this user from being saved:" %>
          </h4>
          <%== @user.errors.full_messages.map{|msg|conten\
t_tag(:li, msg)}.join %>
        </div>
      <% end %>
      <div>
        <%= f.label :name, class: 'block mb-2 text-sm fon\
t-medium text-gray-900' %>
        <%= f.text_field :name, placeholder: 'Name', clas\
s: 'bg-gray-50 border border-gray-300 text-gray-900 text-\
sm rounded-lg focus:ring-blue-500 focus:border-blue-500 b\
lock w-full p-2.5' %>
      </div>
      <div>
        <%= f.label :email, class: 'block mb-2 text-sm fo\
nt-medium text-gray-900' %>
        <%= f.email_field :email, placeholder: 'Enter the\
ir email address', class: 'bg-gray-50 border border-gray-\
300 text-gray-900 text-sm rounded-lg focus:ring-blue-500 \
focus:border-blue-500 block w-full p-2.5' %>
      </div>
      <div>
        <%= f.label :time_zone, class: 'block mb-2 text-s\
m font-medium text-gray-900' %>
        <%= f.time_zone_select :time_zone, ActiveSupport:\
```

```
:TimeZone.us_zones, {}, class: 'bg-gray-50 border border-\
gray-300 text-gray-900 text-sm rounded-lg focus:ring-blue\
-500 focus:border-blue-500 block w-full p-2.5' %>
      </div>
      <%=f.submit "Update My Settings", class: "text-whit\
e bg-blue-700 hover:bg-blue-800 focus:ring-4 focus:outlin\
e-none focus:ring-blue-300 font-medium rounded-lg text-sm\
 w-full sm:w-auto px-5 py-2.5 text-center" %>
    <% end %>
  </div>
</div>
```

Once added, if you navigate to http://localhost:3000/user/me[7], you should see a page where you can edit your email and the timezone of your user.

The final template to create is the page where the current user can update their password. You will need to make this new file at `app/views/users/password.html.erb`:

```
<div class="flex p-5" aria-label="Breadcrumb">
  <ol class="inline-flex items-center space-x-1 md:space-\
x-3">
    <li class="inline-flex items-center">
      <h2 href="#" class="inline-flex items-center font-m\
edium text-gray-700 hover:text-gray-900">
        Users
      </h2>
    </li>
    <li class="inline-flex items-center">
      <h2 href="#" class="inline-flex items-center font-m\
edium text-gray-700 hover:text-gray-900">
        <svg class="w-6 h-6 text-gray-400" fill="currentC\
olor" viewBox="0 0 20 20" xmlns="http://www.w3.org/2000/s\
```

[7] http://localhost:3000/user/me

```
vg"><path fill-rule="evenodd" d="M7.293 14.707a1 1 0 010-\
1.414L10.586 10 7.293 6.707a1 1 0 011.414-1.414l4 4a1 1 0\
 010 1.414l-4 4a1 1 0 01-1.414 0z" clip-rule="evenodd"></\
path></svg>
        My Password
      </h2>
    </li>
  </ol>
</div>
<div class="flex justify-center">
  <div class="p-4 w-4/5 bg-white rounded-lg border border\
-gray-200 shadow-md sm:p-6 lg:p-8">
    <h5 class="text-xl font-bold leading-none text-gray-9\
00">Update My Information:</h5>
    <%= form_with(model: @user, url: my_update_password_p\
ath, id: 'edit_my_password', html: {class: 'space-y-6'}) \
do |f| %>
      <% if @user.errors.any? %>
        <div class="text-red-700">
          <h4>
            <%= "#{pluralize(@user.errors.count, "error")\
} prohibited this user from being saved:" %>
          </h4>
          <%== @user.errors.full_messages.map{|msg|conten\
t_tag(:li, msg)}.join %>
        </div>
      <% end %>
      <div>
        <%= f.label :password, class: 'block mb-2 text-sm\
 font-medium text-gray-900' %>
        <%= f.password_field :password, placeholder: 'New\
 Password', class: 'bg-gray-50 border border-gray-300 tex\
t-gray-900 text-sm rounded-lg focus:ring-blue-500 focus:b\
order-blue-500 block w-full p-2.5' %>
      </div>
```

```
      <div>
        <%= f.label :password_confirmation, class: 'block\
 mb-2 text-sm font-medium text-gray-900' %>
        <%= f.password_field :password_confirmation, plac\
eholder: 'Confirm Password', class: 'bg-gray-50 border bo\
rder-gray-300 text-gray-900 text-sm rounded-lg focus:ring\
-blue-500 focus:border-blue-500 block w-full p-2.5' %>
      </div>
      <%=f.submit "Update Password", class: "text-white b\
g-blue-700 hover:bg-blue-800 focus:ring-4 focus:outline-n\
one focus:ring-blue-300 font-medium rounded-lg text-sm w-\
full sm:w-auto px-5 py-2.5 text-center" %>
    <% end %>
  </div>
</div>
```

Now, if you navigate to http://localhost:3000/user/password[8], you should see a page similar to where you updated your information, except now you can update your password.

These changes complete the files needed to manage users in your application. First, the primary user will be able to sign in or sign up. Then invite users to the application. Finally, the invited users will be able to log in and manage their accounts.

A few more things, you can delete the following template files that Rails created when auto-generating the `User` controller and its files. Inside the `app/views/users` folder, you can delete `create.html.erb`, `update.html.erb` and `show.html.erb`. Additionally, you can delete the whole `spec/views/users` folder as we will not use any view specs in favor of using System-level specs.

Now would be an excellent time to commit your changes and grab a coffee or a nap because it's time to write some tests for all of that added functionality!

[8] http://localhost:3000/user/password

4.3.3 Testing User Management

The primary focus of testing the new functionality you have just added will be controller specs and system specs. There is one small addition to the `users` factory that will connect the `user` belonging to an `account`:

```
FactoryBot.define do
  factory :user do
    account
    name {"MyString"}
    email { Faker::Internet.email }
    password {"123ewq"}
    password_confirmation {"123ewq"}
  end
end
```

In the FactoryBot DSL, simple associations are created by adding the association name as another line. You can reference the documentation here[9] if you want to look further into associations in FactoryBot.

To be able to use the `Faker` library, we will need to add the gem. Add `gem 'faker'` in the `group :development, :test do` block and `bundle install`.

Also, as the `UsersController` is responsible for the email that goes out due to specific controller actions, there is a simple line to add in before the last `end` in the `config/environments/test.rb` file. This setting will ensure the Test environments have the information to render the mail templates.

[9] https://github.com/thoughtbot/factory_bot/blob/master/GETTING_STARTED.md#associations

```
...
  Rails.application.routes.default_url_options[:host] = '\
localhost:3000'
```

With that update out of the way, you can now implement a spec file for the UsersController. The file was created with the Rails Controller generator but will be mostly overwritten with more accurate tests and tests for each action. You can simply paste the following contents into `spec/controllers/users_controller_spec.rb`:

```
require 'rails_helper'

RSpec.describe "Users", type: :request do
  login_user

  describe "GET /index" do
    it "returns http success" do
      get "/account/users"
      expect(response).to have_http_status(:success)
    end

    it "renders :index" do
      get "/account/users"
      expect(response).to render_template :index
    end

    it "lists users" do
      get "/account/users"
      expect(assigns(:users).length).to equal(1)
    end
  end

  describe "GET /me" do
    it "returns http success" do
      get "/user/me"
```

```
      expect(response).to have_http_status(:success)
    end
  end

  describe "PATCH #update_me" do
    user = FactoryBot.create(:user)
    it "returns http success" do
      patch  "/user/update_me", params: {id: user.id, use\
r: {name: 'John'}}
      expect(response).to redirect_to my_settings_path
    end

    it "renders :me of failure" do
      patch  "/user/update_me", params: {
        id: user.id,
        user: {name: 'John', email: nil}
      }
      expect(response).to render_template :me
    end
  end

  describe "GET #password" do
    it "returns http success" do
      get '/user/password'
      expect(response).to have_http_status(:success)
    end
  end

  describe "PATCH #update_password" do
    user = FactoryBot.create(:user)
    it "returns http success" do
      patch '/user/update_password', params: {
        id: user.id,
        user: {password: 'd2g4j6mlp', password_confirmati\
on: 'd2g4j6mlp'}
```

```
      }
      expect(response).to redirect_to my_password_path
    end

    it "renders :password of failure" do
      patch '/user/update_password', params: {id: user.id\
, user: {
        password: '123',
        password_confirmation: '123'
      }}
      expect(response).to render_template :password
    end
  end

  describe "GET /new" do
    it "returns http success" do
      get "/account/users/new"
      expect(response).to have_http_status(:success)
    end

    it "renders new template" do
      get "/account/users/new"
      expect(response).to render_template :new
    end
  end

  describe "POST #create" do
    it "creates an user" do
      post "/account/users", params: {
        user: FactoryBot.attributes_for(:user, {role: 'us\
er'})
      }
      expect(response).to redirect_to account_users_path
    end
```

```
    it "renders :new on failure" do
      post "/account/users",  params: {
        user: FactoryBot.attributes_for(:user,{email: nil\
, role: 'user'})
      }
      expect(response).to render_template :new
    end

    it "fails on non-unique email" do
      user = FactoryBot.create(:user)
      post "/account/users",  params: {
        user: FactoryBot.attributes_for(:user,{
          email: user.email,
          role: 'user'
          }
        )}
      expect(response).to render_template :new
    end
  end

  describe "GET /edit" do
    user = FactoryBot.create(:user)
    it "returns http success" do
      get "/account/users/#{user.id}/edit"
      expect(response).to have_http_status(:success)
    end

    it "renders edit template" do
       get "/account/users/#{user.id}/edit"
      expect(response).to render_template :edit
    end
  end

  describe "PUT #update" do
    user = FactoryBot.create(:user)
```

```
    it "returns http success" do
      put "/account/users/#{user.id}",  params: {id: user\
.id, user: {name: "Yay"}}
      expect(response).to redirect_to account_users_path
    end

    it "renders :edit on failure" do
      put "/account/users/#{user.id}",  params: {
        id: user.id,
        user: FactoryBot.attributes_for(:user,{email: nil\
, role: 'user'})
      }
      expect(response).to render_template :edit
    end
  end

  describe "DELETE #destroy" do
    user = FactoryBot.create(:user)
    it "returns http success" do
      delete "/account/users/#{user.id}",  params: {id: u\
ser.id}
      expect(response).to redirect_to account_users_path
    end
  end
end
```

This controller spec file is relatively easy to follow. It tests the response, template rendered, or redirect followed on all of the actions in the `UsersController`. The syntax here is pretty much the same as the `accounts_controller_spec` file earlier. However, there are quite a few more actions and conditional branches to check with this file.

If you were to run your specs right now, you would see that there are some failing tests from the `requests/accounts_spec` while the new tests pass.

```
...FF.............................

Failures:

  1) Accounts GET #new returns http success
     Failure/Error: expect(response).to have_http_status(\
:success)
       expected the response to have a success status cod\
e (2xx) but it was 302
     # ./spec/requests/accounts_spec.rb:9:in `block (3 le\
vels) in <top (required)>'

  2) Accounts GET #new renders new template
     Failure/Error: expect(response).to render_template :\
new
       expecting <"new"> but was a redirect to <http://ww\
w.example.com/>
     # ./spec/requests/accounts_spec.rb:14:in `block (3 l\
evels) in <top (required)>'

Finished in 1.77 seconds (files took 3.31 seconds to load)
35 examples, 2 failures

Failed examples:

rspec ./spec/requests/accounts_spec.rb:7 # Accounts GET #\
new returns http success
rspec ./spec/requests/accounts_spec.rb:12 # Accounts GET \
#new renders new template
```

When the `account` line is added to the user factory, it creates the account and associates it with the logged-in user. However, the two failing tests rely on the `account` not being set up. The fix is actually pretty simple. You will replace the `login_user` line with a new `before(:each)` block. This block will basically set up a new user

before each test in `accounts_spec` manually, without an associated `account`:

```
require 'rails_helper'

RSpec.describe "Accounts", type: :request do
  before(:each) do
    user = User.create!(
      name: "John",
      email: "specuser@test.com",
      password: 'spec123'
      )
    sign_in user
  end

  describe "GET #new" do
    it "returns http success" do
      get '/accounts/new'
      expect(response).to have_http_status(:success)
    end

    it "renders new template" do
      get '/accounts/new'
      expect(response).to render_template :new
    end
  end

  describe "POST #create" do
    it "creates an account" do
      post '/accounts', params: { account: FactoryBot.att\
ributes_for(:account)}
      expect(response).to redirect_to root_path
    end

    it "renders :new on failure" do
      post '/accounts',  params: {account: FactoryBot.att\
```

```
ributes_for(:account,{
        name: nil
      })}
      expect(response).to render_template :new
    end
  end
end
```

At this time, all of your tests should now be passing:

```
rspec
.....................................

Finished in 1.68 seconds (files took 3.13 seconds to load)
35 examples, 0 failures
```

It is also a valuable idea to write some system specs for the new functionality. This will ensure the main success and failure paths will behave in the browser as expected.

When dealing with a general system spec concept but many views or actions, the best approach will usually be to create a folder and then put system specs for each view/action in separate folders. In your case, it would be advisable to create a folder `spec/system/user_management`, which will house individual spec files such as `new_spec.rb`.

Once again, system specs will want to work with Devise like controller specs. Thus, creating a macro helper for system specs will help keep the system specs clean. Back in `spec/support`, You can add a file called `system_macros.rb` with the following content:

```
module SystemMacros
  def login_user
    before(:each) do
      @user = FactoryBot.create(:user)
      @user.add_role :user, @user.account
      login_as(@user, scope: :user)
    end
  end

  def login_admin
    before(:each) do
      @admin = FactoryBot.create(:user)
      @admin.add_role :admin, @admin.account
      login_as(@admin, :scope => :user)
    end
  end
end
```

This is very close to the code in the request_macros but mainly uses a different method specific to logging a user in from a system spec. Though now we can use either method to login a user with the correct role. The file will need to be included in the `spec/rails_helper.rb` file. This will make the method available in all system specs by adding the line `login_user`. Here are the new lines:

```
...
config.extend ControllerMacros, :type => :controller
config.extend SystemMacros, type: :system
...
```

Instead of adding another `require` line at the top to have Rails (and specifically this file) load the module, you can uncomment a line already included in the `rails_helper` that is designed to load any file in a support folder. To do this, remove the `#` from this line:

```
Dir[Rails.root.join('spec', 'support', '**', '*.rb')].sor\
t.each { |f| require f }
```

Let's jump into the first system spec for `user_management`!

You may notice that now System specs fail due to a missing web driver. We can add a gem to solve this issue as the error hints. First, to add the gem, we will add `gem 'webdrivers', '~> 5.0', require: false` to the `group :development, :test do` and run `bundle install`. This gem makes it easy to have the latest version of the chromedriver available for your specs. Next, to make sure the spec suite uses the webdriver gem, update the `specs/spec_helper.rb` file. The striking thing about Rails System tests that use Capybara is that even if you don't know the exact syntax used in this file. You can still follow along with this file and understand what is being tested. Running RSpec on the new file should now yield two passing tests:

```
require 'simplecov'
SimpleCov.start 'rails'

require 'webdrivers' ## <-- New line
require 'capybara/rspec'  ## <-- New line
```

Then, at the bottom add the following:

```
config.before(:each, type: :system) do
  driven_by :rack_test
end

config.before(:each, type: :system, js: true) do
  driven_by :selenium_chrome_headless
end

...
```

```
Capybara.javascript_driver = :selenium_chrome_headless
Capybara.server = :puma, { Silent: true }
```

This will set up our non-javascript-based specs to use RackTest. This driver is nearly a text-based browser that will keep static system specs running quickly. Next, it sets `selenium_chrome_headless` as the JavaScript-based system spec driver. When JavaScript is needed, it will run in a Headless Chrome Browser. Finally, the last line keeps `Puma` logs from littering the RSpec output.

Back to the System spec:

```
require 'rails_helper'

RSpec.describe 'New User', type: :system do
  login_admin

  it 'should be able to visit the user index and see at l\
east one user' do
    visit account_users_path

    expect(page).to have_current_path(account_users_path)
    expect(page).to have_content(@admin.email.to_s)
  end

  context 'a regular user' do
    login_user

    it 'should redirect with only user access' do
      visit account_users_path

      expect(page).to have_current_path(root_path)
    end
  end

  context 'click and new the user' do
```

```
    it 'should click new and go to new page' do
      visit account_users_path
      click_on('Add User')
      expect(page).to have_current_path(new_account_user_\
path)
    end

    context 'and on the new page' do
      it 'and new the user successfully' do
        visit account_users_path
        click_on('Add User')

        within '#new_user' do
          fill_in 'user_email', with: 'test@test.com'
          select('User', from: 'user_role')
        end

        click_button('Add User')
        expect(page).to have_current_path(account_users_p\
ath)
      end

      it 'and fail to new the user successfully' do
        visit account_users_path
        click_on('Add User')

        within '#new_user' do
          fill_in 'user_email', with: ''
          select('User', from: 'user_role')
        end

        click_button('Add User')
        expect(page).to have_current_path(account_users_p\
ath)
      end
```

```
    end
  end
end
```

Nothing too crazy going on in this spec file. It's using the new `system_macros` method. Then it accesses the `new user` form from the `/accounts/users` page. After that, it fills in the form and tests for both a successful and failed form submission.

Next is the `edit` spec, which will be nearly identical:

```
require 'rails_helper'

RSpec.describe "Edit User", type: :system do
  login_user

  it "should be able to visit the user index and see at l\
east one user" do

    visit account_users_path

    expect(current_path).to eql(account_users_path)
    expect(page).to have_content("#{@user.email}")
  end

  context "click and edit the user" do

    it "should click edit and go to edit page" do
      visit account_users_path
      first('tbody tr').click_on("Edit")
      expect(current_path).to eql(edit_account_user_path(\
@user))
    end

    context "and on the edit page" do
      it "and edit the user successfully" do
```

```
        visit account_users_path
        first('tbody tr').click_on("Edit")

        within "#edit_user" do
          fill_in "user_email", with: 'test@test.com'
        end

        click_button("Edit User")
        expect(current_path).to eql(account_users_path)
      end

      it "and fail to edit the user successfully" do
        visit account_users_path
        first('tbody tr').click_on("Edit")

        within "#edit_user" do
          fill_in "user_email", with: ''
          select('User', :from => 'user_role')
        end

        click_button("Edit User")
        expect(current_path).to eql(account_user_path(@us\
er))
      end
    end
  end
end
```

This file has the same idea but uses CSS matchers in the `first()` method to find the first row and click the button in that row. When using Capybara, it can be necessary to use methods to specify HTML elements as some methods such as `click("some button text")` may be too ambiguous. Anyway, the rest of the file is nearly identical, aside from a small change. When selecting the form, you may notice that the selector uses a different syntax, `#edit_user`.

Here we're continuing on with more system specs with the addition of `me_spec`:

```
require 'rails_helper'

RSpec.describe 'My Settings', type: :system do
  login_user

  it 'should be able to visit my_settings' do
    visit my_settings_path

    expect(page).to have_current_path(my_settings_path)
  end

  it 'should edit my settings successfully' do
    visit my_settings_path

    within '#edit_my_user' do
      fill_in 'user_email', with: 'test@test.com'
    end

    click_button('Update My Settings')
    expect(page).to have_current_path(my_settings_path)
  end

  it 'and fail to edit my settings successfully' do
    visit my_settings_path

    within '#edit_my_user' do
      fill_in 'user_email', with: ''
    end

    click_button('Update My Settings')
    expect(page).to have_current_path(update_my_settings_\
path)
    expect(page).to have_content('error')
```

```
  end
end
```

This file also follows closely with others in the `user_management` folder. However, this time, it tests the form on the page that allows the logged-in user to edit their own settings.

Lastly, we will be working on the `password_spec` file:

```
require 'rails_helper'

RSpec.describe 'Password Change', type: :system do
  login_user

  it 'should be able to visit my_password' do
    visit my_password_path

    expect(page).to have_current_path(my_password_path)
  end

  it 'should edit my password successfully' do
    visit my_password_path

    within '#edit_my_password' do
      fill_in 'user_password', with: 'pass123'
      fill_in 'user_password_confirmation', with: 'pass12\
3'
    end

    click_button('Update Password')
    expect(page).to have_current_path(my_password_path)
  end

  it 'and fail to edit my password successfully' do
    visit my_password_path
```

```
    within '#edit_my_password' do
      fill_in 'user_password', with: 'Test'
      fill_in 'user_password_confirmation', with: ''
    end

    click_button('Update Password')
    expect(page).to have_current_path(my_update_password_\
path)
    expect(page).to have_content('error')
  end
end
```

The `password_spec` is nearly identical to the `me_spec` before it, with minor differences for button text and input field selectors.

Before running all those specs, two more spec files were autogenerated that we will not need. You can delete `spec/models/role_spec.rb` and `spec/helpers/users_helper_spec.rb`. If you have been running the whole test suite while adding files, you may have seen those two files were adding two auto-generated pending spec methods.

With these previously added spec files in place, running RSpec should yield passing test results with a high percentage of code coverage:

```
rspec
.........................................F........

Failures:

  1) New User a regular user should redirect with only us\
er access
     Failure/Error: expect(page).to have_current_path(roo\
t_path)
       expected "/account/users" to equal "/"
```

```
    # ./spec/system/user_management/new_spec.rb:19:in `b\
lock (3 levels) in <top (required)>'

Finished in 2.96 seconds (files took 3.52 seconds to load)
50 examples, 1 failure
```

However, it actually has one failing test. This test relates to the actual handling of role-based access, which you are about to set up.

4.3.4 CanCanCan

CanCanCan is a ruby gem authorization library that solves authorization from a single Ruby file to specify permissions. With that file implemented, your controllers and view can use built-in CanCanCan helper methods.

With CanCanCan already installed through bundled from a previous chapter, only a few commands will be run to get CanCanCan working.

The syntax CanCanCan uses follows `[can or cannot]`. Then, an array of action names the permission applies to, such as `:index` or `:new` followed by the permitted resource object, such as a `User`.

A user is passed into the `Ability` `initialize` method. With that, we can then base permission levels on their role within the application.

You may notice the lack of all actions being mentioned. This is due to the `:manage` symbol representing all standard Rails RESTful actions. Also, this means nonconventional actions such as `:feed` or `:me` need to be explicitly specified.

Next, you can add a CanCanCan command to each controller that will be secured by CanCanCan. You will add the line to `UsersController`, `AccountsController` and `ActivityController`:

```
class UsersController < ApplicationController
  before_action :set_user, only: [:show, :edit, :update, \
:destroy]
  before_action :set_users, only: [:index]
  load_and_authorize_resource
...
```

and

```
class AccountsController < ApplicationController
  load_and_authorize_resource
```

and finally...

```
class ActivityController < ApplicationController
  authorize_resource :class => "ActivityController"
```

Here, there is another slight change. As no particular resource object is tied to this controller, we instruct CanCanCan to pass the Controller class name instead of inferring a model class name.

There are two small additions to add to get your controllers using your new `ability.rb` file. First, we will add `attr_accessor :role` to the `User` model. This will ensure that when the permission file loads a User object, it has access to a role attribute, even though roles are managed through an association and a gem.

```
class User < ApplicationRecord
  rolify
  attr_accessor :role
  belongs_to :account, optional: true
  # Include default devise modules. Others available are:
  # :confirmable, :lockable, :timeoutable and :omniauthab\
le
  devise :invitable, :database_authenticatable, :register\
able,
         :recoverable, :rememberable, :validatable
end
```

Lastly, by default, CanCanCan throws an error when someone tries to access a page or action they do not have access to. To handle this within the application, you can add the following block of code to the `ApplicationController` to redirect and inform the user they do not have permissions.

```
rescue_from CanCan::AccessDenied do |exception|
  redirect_to root_url, :error => exception.message
end
add_flash_types :error
```

This block of code mostly catches the `CanCan::AccessDenied` error and redirects the user to the root_path with an error "flash" message. By default, Rails does not actually work with `error` flash messages. However, the `add_flash_types :error` line allows flash messages with an `error` to now be called.

If you were to visit the app as an Admin user right now, it would work as it did previously. However, if you were to log in as an invited user, you would now not be able to access the `User` list page, User new form, or the User edit form. Again, though, access to your own settings and password remains open.

If you have been running your tests while adding the permission functionality, you would now notice there are actually plenty of

failing tests. This is good, actually! Without these tests, you may have gone on with regression bugs. This is the power of 'testing' while developing. With that being said, the failing tests are actually due to them not recognizing the differences between the `:user` and `:admin` user roles.

We will first modify the macro spec helper files to have both an admin user and a regular-user method. This will allow the tests to perform under an admin or regular user to match the expectations based on CanCanCan permissions.

```
module RequestMacros
  def login_user
    before(:each) do
      @user = FactoryBot.create(:user)
      @user.add_role :user, @user.account
      sign_in @user
    end
  end

  def login_admin
    before(:each) do
      @admin = FactoryBot.create(:user)
      @admin.add_role :admin, @admin.account
      sign_in @admin
    end
  end
end
```

The new controller macros are pretty straightforward and match the changes to the System Macros changed earlier for the system specs you wrote.

We're in the home stretch here. The `requests/users_spec` file will now have a new default user role in the `before(:each)` test block of `:admin`. Then set individual tests user's roles based on the

permission level needed to test the success and failure patterns. The easiest way to show the updates is to show you the whole file:

```
require 'rails_helper'

RSpec.describe "Users", type: :request do
  login_user

  describe "GET /index" do
    login_admin

    it "returns http success" do
      get "/account/users"
      expect(response).to have_http_status(:success)
    end

    it "renders :index" do
      get "/account/users"
      expect(response).to render_template :index
    end

    it "lists users" do
      get "/account/users"
      expect(assigns(:users).length).to equal(1)
    end
  end

  describe "GET /me" do
    it "returns http success" do
      get "/user/me"
      expect(response).to have_http_status(:success)
    end
  end

  describe "PATCH #update_me" do
    user = FactoryBot.create(:user)
```

```
    it "returns http success" do
      patch  "/user/update_me", params: {id: user.id, use\
r: {name: 'John'}}
      expect(response).to redirect_to my_settings_path
    end

    it "renders :me of failure" do
      patch  "/user/update_me", params: {
        id: user.id,
        user: {name: 'John', email: nil}
      }
      expect(response).to render_template :me
    end
  end

  describe "GET #password" do
    it "returns http success" do
      get '/user/password'
      expect(response).to have_http_status(:success)
    end
  end

  describe "PATCH #update_password" do
    user = FactoryBot.create(:user)
    it "returns http success" do
      patch '/user/update_password', params: {
        id: user.id,
        user: {password: 'd2g4j6mlp', password_confirmati\
on: 'd2g4j6mlp'}
      }
      expect(response).to redirect_to my_password_path
    end

    it "renders :password of failure" do
      patch '/user/update_password', params: {id: user.id\
```

```
, user: {
        password: '123',
        password_confirmation: '123'
      }}
      expect(response).to render_template :password
    end
  end

  describe "GET /new" do
    login_admin

    it "returns http success" do
      get "/account/users/new"
      expect(response).to have_http_status(:success)
    end

    it "renders new template" do
      get "/account/users/new"
      expect(response).to render_template :new
    end

    it "fails on unauthenticated user" do
      user = User.create!(
        name: "John",
        email: "specuseruser@test.com",
        password: 'spec123')
      user.add_role :user
      sign_in user
      get "/account/users/new"
      expect(response).to have_http_status(:redirect)
    end
  end

  describe "POST #create" do
    login_admin
```

```
    it "creates an user" do
      post "/account/users", params: {
        user: FactoryBot.attributes_for(:user, {role: 'us\
er'})
      }
      expect(response).to redirect_to account_users_path
    end

    it "renders :new on failure" do
      post "/account/users",  params: {
        user: FactoryBot.attributes_for(:user,{email: nil\
, role: 'user'})
      }
      expect(response).to render_template :new
    end

    it "fails on non-unique email" do
      user = FactoryBot.create(:user)
      post "/account/users",  params: {
        user: FactoryBot.attributes_for(:user,{
          email: user.email,
          role: 'user'
          }
        )}
      expect(response).to render_template :new
    end
  end

  describe "GET /edit" do
    login_admin

    user = FactoryBot.create(:user)
    it "returns http success" do
      get "/account/users/#{user.id}/edit"
```

```
      expect(response).to have_http_status(:success)
    end

    it "renders edit template" do
       get "/account/users/#{user.id}/edit"
      expect(response).to render_template :edit
    end
  end

  describe "PUT #update" do
    login_admin

    user = FactoryBot.create(:user)
    it "returns http success" do
      put "/account/users/#{user.id}",  params: {id: user\
.id, user: {name: "Yay"}}
      expect(response).to redirect_to account_users_path
    end

    it "renders :edit on failure" do
      put "/account/users/#{user.id}",  params: {
        id: user.id,
        user: FactoryBot.attributes_for(:user,{email: nil\
, role: 'user'})
      }
      expect(response).to render_template :edit
    end
  end

  describe "DELETE #destroy" do
    login_admin

    user = FactoryBot.create(:user)
    it "returns http success" do
      delete "/account/users/#{user.id}",  params: {id: u\
```

```
ser.id}
      expect(response).to redirect_to account_users_path
    end
  end
end
```

Finally, the last system spec to change is `user_management/edit_spec` with the same changes from `user` to `admin` and a new test to ensure the application behaves correctly for failed permissions.

```
require 'rails_helper'

RSpec.describe "Edit User", type: :system do
  login_admin

  it "should be able to visit the user index and see at l\
east one user" do

    visit account_users_path

    expect(current_path).to eql(account_users_path)
    expect(page).to have_content("#{@admin.email}")
  end

  context "click and edit the user" do

    it "should click edit and go to edit page" do
      visit account_users_path
      first('tbody tr').click_on("Edit")
      expect(current_path).to eql(edit_account_user_path(\
@admin))
    end

    context "and on the edit page" do
      it "and edit the user successfully" do
```

```
        visit account_users_path
        first('tbody tr').click_on("Edit")

        within "#edit_user" do
          fill_in "user_email", with: 'test@test.com'
        end

        click_button("Edit User")
        expect(current_path).to eql(account_users_path)
      end

      it "and fail to edit the user successfully" do
        visit account_users_path
        first('tbody tr').click_on("Edit")

        within "#edit_user" do
          fill_in "user_email", with: ''
          select('User', :from => 'user_role')
        end

        click_button("Edit User")
        expect(current_path).to eql(account_user_path(@ad\
min))
      end
    end
  end
end
```

Running the `rspec` command one more time should yield all tests passing, with nearly 100% code coverage.

```
rspec
.................................................

Finished in 3.79 seconds (files took 3.49 seconds to load)
51 examples, 0 failures

Coverage report generated for RSpec to standup_app/covera\
ge. 145 / 164 LOC (88.41%) covered.
```

While wrapping up this chapter, there is one slight little variation to your usual git-flow. Firstly, you will want to commit your updates. Afterward, you will switch to the main branch and merge in all of the `users` branch you have worked on.

```
git add .
git commit -am "User/CanCanCan tests updates"
git checkout main
git merge users
git push
```

You will now have all your user code in the main branch and pushed up to your remote main branch.

4.4 What did you learn in this chapter?

- You learned how to create a Rails model, specifically a User model.
- You learned how to integrate Devise into your application for authentication.
- You learned to create views with tables and forms that interact with database objects.

- You learned how to create a user invitation system with devise_invitable.
- You learned how to associate roles with users to later be used in authorization techniques.
- You learned how to apply authorization through CanCanCan.
- You improved your test writing skills.

4.5 Exercises

Exercise answers can be found with any package or bundle purchases of Build A SaaS App in Ruby on Rails 7[10]!

1. Sometimes, you just need more data about your user. Use `rails generate migration` to add a `title` attribute to the User model.
2. Display the user's title on the Users list page.
3. Allow both the current user to change their own title and admin users to edit all titles.

[10] https://buildasaasappinrails.com

5. Business Models and Logic

Applications are complex structures that have many layers of business objects. That business logic controls how data behaves. The business objects might represent a top-level object such as a `standup`. A `standup` can belong to a user and have many related `Tasks` or `Blockers`.

Other models function as a collection or grouping mechanism, such as Teams. Yes, teams hold settings to help control notifications or the time when a standup occurs. It can also contain a list of users that serves as a grouping or filtering mechanism. Imagine if there were 20 users in an account and no other way to filter down daily standups. That would be a firehose of updates!

Additionally, other objects span the request and are used as a `current` context. Earlier, you saw that Devise gave you a `current_user` object representing the currently logged-in user. Also, you added a `current_account` method as a more straightforward way to get the current account object that the `current_user` `belongs_to`. Finally, you will shortly add a new approach that memoizes a string representation of the `current_date`. The current date is an essential concept in a standup_app. This happens to be yet another way daily standups are grouped.

Teams handle configuration of two things first, a way to group users into smaller logical groups within the account and particular settings for that group, such as how notifications, reminders, and recaps are sent. Additionally, it keeps the standup meeting time and other time-related settings.

Standups themselves are a simple object. They hold only a few attributes. Such as the user it belongs to, the date it was submitted

for (a standup could be created on a different date than the date it represents), and when it was sent via the included Rails' timestamp attributes. The standup model also holds a few child associations that connect dids, todos, and blockers.

Both the Todos and Blockers hold very few attributes. It includes the body of text, a title, the user it `belongs_to`, and the standup it `belongs_to`.

The great thing about Rails is that relationships are defined by a small one-line setting on each model in the relationship and a referential column on the child tables. Once relationships are set up and established, it will be simple to get referential objects:

```
Standup.find(1).todos

Todo.find(2).standup

Blocker.find(3).standup.todos
```

You can also see the benefit when writing your view templates too:

```
<ul>
  <% @standup.todos.each do |todo| %>
    <li><%= todo.name %></li>
  <% end %>
</ul>
```

You have used various rails command-line interfaces (CLIs) throughout the chapters. You used different command line generators, such as creating a controller (`rails g controller Name action1`). However, one of the most robust Rails CLI generators is the Scaffold. The Scaffold means it creates the resource specified in the command throughout the whole stack. Generating a scaffold Rails creates:

- A complete controller and route that contains all the endpoints.
- Controller actions needed to have an entire RESTful interface.
- A model and a migration to create the database table
- View files for each of the controller's actions that need a template
- A resource named stylesheet
- A resource named javascript file
- Test files to cover all the files created.

As powerful as this command is, there may be some settings you want to change on a per-use basis. Using `rails g scaffold --help` gives you a complete list of options and Rails' explanation of the scaffold generator. A few notable options are:

- `--no-javascripts` - There are some cases where you would like to skip the javascript file generations. `--no-stylesheets` do the same for skipping the CSS or SCSS file generation. Finally, `--no-assets` skips both stylesheet and javascript file generation.
- `--api` makes the controller an API controller instead of the standard Rails view template defaults.
- `--model-name=MODEL_NAME` - Will allow you to create the resource stack with a different name for the model. With that model not following the automatic singular/plural inferred defaults.

Again, those are a few of the commands and are not an exhaustive list.

5.0.1 `rails g scaffold`

What good would all of that scaffolding explanation be if we didn't put it into practice? So, our first scaffold use will be the `Standup`

resource. Now, to have the scaffold command generate the files, enter the following on the command line:

```
rails g scaffold Standup user:references standup_date:dat\
e --no-view-specs --no-helper-specs --no-helper --no-jbui\
lder --no-routing-specs
      invoke  active_record
      create    db/migrate/20220422211702_create_standups\
.rb
      create    app/models/standup.rb
      invoke    rspec
      create      spec/models/standup_spec.rb
      invoke      factory_bot
      create        spec/factories/standups.rb
      invoke  resource_route
       route    resources :standups
      invoke  scaffold_controller
      create    app/controllers/standups_controller.rb
      invoke    tailwindcss
      create      app/views/standups
      create      app/views/standups/index.html.erb
      create      app/views/standups/edit.html.erb
      create      app/views/standups/show.html.erb
      create      app/views/standups/new.html.erb
      create      app/views/standups/_form.html.erb
      create      app/views/standups/_standup.html.erb
      invoke    resource_route
      invoke    rspec
      create      spec/requests/standups_spec.rb
```

A few extra options were included to skip test files that would not be used.

Now, migrate the database:

```
rails db:migrate
== 20220422211702 CreateStandups: migrating =============\
======================
-- create_table(:standups, {:id=>:uuid})
   -> 0.0269s
== 20220422211702 CreateStandups: migrated (0.0270s) ====\
======================
```

Now, let's dig into the main files created and why they were generated the way they were from the scaffold.

5.0.2 Model

You will find a model file for Standups at `app/models/standup.rb`. As mentioned previously, your model file is your interface to the database tables it represents. In this case, the `Standup` model will interface with rows from the `standups` table. Make sure your `Standup` model matches the following code:

```
class Standup < ApplicationRecord
  belongs_to :user
end
```

5.0.3 Controller

As stated in an earlier chapter, a controller is the brains of the most requests in a Rails application. After a request is routed to a controller's action, that action gathers data, changes data, and returns a response to the user. A response may be a server-generated page, JSON, or a simple status code.

In this case, the controller was created for us from the scaffold generator command earlier. The best part about generated controllers is

that they will handle all the necessary RESTful routes most essential resources need.

Let's take a look at the generated controller:

```
class StandupsController < ApplicationController
  before_action :set_standup, only: %i[ show edit update \
destroy ]

  # GET /standups or /standups.json
  def index
    @standups = Standup.all
  end

  # GET /standups/1 or /standups/1.json
  def show
  end

  # GET /standups/new
  def new
    @standup = Standup.new
  end

  # GET /standups/1/edit
  def edit
  end

  # POST /standups or /standups.json
  def create
    @standup = Standup.new(standup_params)

    respond_to do |format|
      if @standup.save
        format.html { redirect_to standup_url(@standup), \
notice: "Standup was successfully created." }
        format.json { render :show, status: :created, loc\
```

```
ation: @standup }
      else
        format.html { render :new, status: :unprocessable\
_entity }
        format.json { render json: @standup.errors, statu\
s: :unprocessable_entity }
      end
    end
  end

  # PATCH/PUT /standups/1 or /standups/1.json
  def update
    respond_to do |format|
      if @standup.update(standup_params)
        format.html { redirect_to standup_url(@standup), \
notice: "Standup was successfully updated." }
        format.json { render :show, status: :ok, location\
: @standup }
      else
        format.html { render :edit, status: :unprocessabl\
e_entity }
        format.json { render json: @standup.errors, statu\
s: :unprocessable_entity }
      end
    end
  end

  # DELETE /standups/1 or /standups/1.json
  def destroy
    @standup.destroy

    respond_to do |format|
      format.html { redirect_to standups_url, notice: "St\
andup was successfully destroyed." }
      format.json { head :no_content }
```

```
    end
  end

  private
    # Use callbacks to share common setup or constraints \
between actions.
    def set_standup
      @standup = Standup.find(params[:id])
    end

    # Only allow a list of trusted parameters through.
    def standup_params
      params.require(:standup).permit(:user_id, :standup_\
date)
    end
end
```

As you can see in this file, there is an `index` page, which would return a listing of standups. A `show`, which would represent one standup. `new` and `create` to handle a form and its submission route. `edit` and `update` to do the same for changes to a standup. Finally, there is a `destroy` endpoint to delete a standup.

However, normal isn't how most SaaS apps end up. So we will begin modifying this controller to match the interface we will be using for standups in a bit.

5.0.4 Views

The same can be said about the automatically generated views. These views will match the actions and are meant to serve as markup assets. There are no views for the `create`, `update`, and `destroy` actions. Those actually will attempt to perform an action and redirect back to the appropriate route based on its success.

Let's look at a list of the scaffold's command line output:

```
create      app/views/standups
create      app/views/standups/index.html.erb
create      app/views/standups/edit.html.erb
create      app/views/standups/show.html.erb
create      app/views/standups/new.html.erb
create      app/views/standups/_form.html.erb
create      app/views/standups/_standup.html.erb
```

As you can see, four `.erb` view templates (and a `_form` partial) were created. Since they are relatively simple and even lack the markup that will make them match the TailwindCSS markup, let's take a quick look:

The First template will be the `edit` template. This template will nearly exactly match the `new` form. There is a slight change in text, but both will mainly defer the logic to a `_form` partial.

```
<div class="mx-auto md:w-2/3 w-full">
  <h1 class="font-bold text-4xl">Editing standup</h1>

  <%= render "form", standup: @standup %>

  <%= link_to "Show this standup", @standup, class: "ml-2\
 rounded-lg py-3 px-5 bg-gray-100 inline-block font-mediu\
m" %>
  <%= link_to "Back to standups", standups_path, class: "\
ml-2 rounded-lg py-3 px-5 bg-gray-100 inline-block font-m\
edium" %>
</div>
```

Put simply the view displays text, render's the form(which you will see in a moment) and shows some navigational links. Let's look at the form real quick:

```
<%= form_with(model: standup, class: "contents") do |form\
| %>
  <% if standup.errors.any? %>
    <div id="error_explanation" class="bg-red-50 text-red\
-500 px-3 py-2 font-medium rounded-lg mt-3">
      <h2><%= pluralize(standup.errors.count, "error") %>\
 prohibited this standup from being saved:</h2>

      <ul>
        <% standup.errors.each do |error| %>
          <li><%= error.full_message %></li>
        <% end %>
      </ul>
    </div>
  <% end %>

  <div class="my-5">
    <%= form.label :user_id %>
    <%= form.text_field :user_id, class: "block shadow ro\
unded-md border border-gray-200 outline-none px-3 py-2 mt\
-2 w-full" %>
  </div>

  <div class="my-5">
    <%= form.label :standup_date %>
    <%= form.date_field :standup_date, class: "block shad\
ow rounded-md border border-gray-200 outline-none px-3 py\
-2 mt-2 w-full" %>
  </div>

  <div class="inline">
    <%= form.submit class: "rounded-lg py-3 px-5 bg-blue-\
600 text-white inline-block font-medium cursor-pointer" %\
>
  </div>
```

```
<% end %>
```

There is a bit more in this file to mention. First, you will notice the `form_with` form helper. In most cases, you can just provide an object, and Rails handles the form specifics for you. For example, a new unsaved `@standup` object will trigger Rails to generate a form with the `create` path as the destination. Whereas if a `@standup` object exists, a form with the existing values in the form fields and submits to the `update` action will be triggered.

Next, you see a conditional statement based on errors existing. When attempting to save a new or updated object, if Rails encounters any errors on model validation, the form will be re-rendered and displays the errors that are preventing the object from being saved.

Following that, form helpers display form inputs for the existing standup attributes. As we go deeper into the application, you will be able to see all the different form input helpers Rails will provide you for all the standard HTML input types.

We'll wrap up the views section by going to the `index` and `show` templates, which are super straightforward.

```
<div class="w-full">
  <% if notice.present? %>
    <p class="py-2 px-3 bg-green-50 mb-5 text-green-500 f\
ont-medium rounded-lg inline-block" id="notice"><%= notic\
e %></p>
  <% end %>

  <div class="flex justify-between items-center">
    <h1 class="font-bold text-4xl">Standups</h1>
    <%= link_to 'New standup', new_standup_path, class: "\
rounded-lg py-3 px-5 bg-blue-600 text-white block font-me\
dium" %>
  </div>
```

```
  <div id="standups" class="min-w-full">
    <%= render @standups %>
  </div>
</div>
```

Here, the standups collected from the `index` action in the controller are used. Then they are iterated over, displaying themselves as elements. In Rails 7, the generator goes further than in past versions. It now creates and uses a partial for each individual `standup`. This will also help when HotWire is introduced to the application later.

```
<div id="<%= dom_id standup %>">
  <p class="my-5">
    <strong class="block font-medium mb-1">User:</strong>
    <%= standup.user_id %>
  </p>

  <p class="my-5">
    <strong class="block font-medium mb-1">Standup date:<\
/strong>
    <%= standup.standup_date %>
  </p>

  <% if action_name != "show" %>
    <%= link_to "Show this standup", standup, class: "rou\
nded-lg py-3 px-5 bg-gray-100 inline-block font-medium" %\
>
    <%= link_to 'Edit this standup', edit_standup_path(st\
andup), class: "rounded-lg py-3 ml-2 px-5 bg-gray-100 inl\
ine-block font-medium" %>
    <hr class="mt-6">
  <% end %>
</div>
```

Lastly, a link on the index page to create a new standup.

Finally, here is the show template:

```
<div class="mx-auto md:w-2/3 w-full flex">
  <div class="mx-auto">
    <% if notice.present? %>
      <p class="py-2 px-3 bg-green-50 mb-5 text-green-500\
 font-medium rounded-lg inline-block" id="notice"><%= not\
ice %></p>
    <% end %>

    <%= render @standup %>

    <%= link_to 'Edit this standup', edit_standup_path(@s\
tandup), class: "mt-2 rounded-lg py-3 px-5 bg-gray-100 in\
line-block font-medium" %>
    <div class="inline-block ml-2">
      <%= button_to 'Destroy this standup', standup_path(\
@standup), method: :delete, class: "mt-2 rounded-lg py-3 \
px-5 bg-gray-100 font-medium" %>
    </div>
    <%= link_to 'Back to standups', standups_path, class:\
 "ml-2 rounded-lg py-3 px-5 bg-gray-100 inline-block font\
-medium" %>
  </div>
</div>
```

This view template only shows the attributes of the singular standup by using the same partial used in the index and a few links to navigate through other resource routes.

You may be wondering why specs have been ignored. It may not make sense to try and formulate your tests upfront when you are iterating quickly, such as when you are building a SaaS application. It is tricky as you have not yet figured out user interfaces or even more accurate data interfaces and relationships. The views created

by the scaffold will usually suffer the same fate. The controller can also experience some change before they serve their entire purpose.

5.1 Creating The Remaining Business Logic

5.1.1 Current Date

One of the core concepts of the application is `current_date`. Not only can this hold the very current date and time, but the selected date, the end-user may want to view standups, tasks, or roadblocks.

The implementation includes a few new concepts that you have yet to see in detail in this book. First, it introduces the direct use of `sessions`. Rails offers `session` storage that uses an encrypted browser cookie to store data persisted through a user's browser session. This approach benefits the ability to track a data value through a user's whole session in the navigation.

Another almost new concept is using helper methods inside the `ApplicationController`. While this concept is used in the `current_account` method, it will be explained further.

Let's update the `ApplicationController`:

```
class ApplicationController < ActionController::Base
  before_action :authenticate_user!
  layout :layout_by_resource

  helper_method :current_account
  helper_method :current_date ## new code

  rescue_from CanCan::AccessDenied do |exception|
    redirect_to root_url, :error => exception.message
  end
```

```
  add_flash_types :error

  def current_account
    @current_account ||= current_user.account
    @current_account
  end

  def current_date ## new method start
    session[:current_date] =
      session[:current_date] || Date.today.iso8601
    @current_date ||= session[:current_date]
  end ## new method end

  protected
  def layout_by_resource
    if devise_controller?
      "devise"
    else
      "application"
    end
  end
end
```

First, we added the line `helper_method :current_date`, which allows the method to be called from inside any view template. This method will become helpful when displaying the markup around the area of `current_date` and changing it.

Next, you see the actual `current_date` method. While the method may seem different, the new concepts are simple when broken down. First, if there is no session-based value, you find that a session variable is being set by either the current session's `current_date` value *or* the actual current computed date.

The following line uses a technique called `memoization`, which is a fancy way of saying store this value in some cache. In the case of Ruby, Rails, and controller-based values, after the `current_date`

method is run once the value is stored in an instance variable for the duration of the request/response cycle.

You may be thinking, great, we set up a `getter` pattern for `current_date`, but how do we set this? I am glad you asked! It is relatively simple; we will create a new Controller and Action to set the session value.

When creating the most simple controllers, I will often opt to skip generators and do some old fashion `new file` work. Create a new file at `app/controllers/dates_controller.rb` with the following content:

```
class DatesController < ApplicationController
  def update
    session[:current_date] = params[:date]
    if params[:reload_path]
      redirect_to(params[:reload_path])
    else
      redirect_back(fallback_location: root_path, turboli\
nks: true)
    end
  end
end
```

This action is pretty simple. Set the `session[:current_date]` with the parameter from the HTTP request and, in this case, the URL. When that is done, redirect back to where the request came. That is it! Told you it would be simple. Well, except for one little thing. There are times when we will want to control where the redirect back comes from and what it will redirect to, which becomes essential when using the date within the URL itself. However, it is still simple!

To round this out, we need to add a line into the `config/routes.rb` file to let the application know about this controller and action.

Insert the following line right before the `root to:` line near the end of the file:

```
get 'dates/:date', to: 'dates#update', as: 'update_date'
```



Now, you may notice that this route isn't perfectly RESTful while a restful action name of `update` was used. If the date may be set from a link, javascript(which may or may not be a third party), or some other form of an HTTP request, it is easier to use a GET request to grab the date right from the URL. Building quickly and using the best practices can sometimes be at odds. It is helpful to strike a balance now and come back later to refactor.

Lastly, we can write a quick test to confirm the new controller and its action is performing as expected. Create a new file at `spec/requests/dates_spec.rb` with the following content:

```
require 'rails_helper'

RSpec.describe "Dates", type: :request do
  login_user

  describe 'GET #update' do
    it 'returns http success' do
      get update_date_url(date: '01-01-2022')
      expect(response).to have_http_status(:redirect)
      expect(session[:current_date]).to eq('01-01-2022')
    end
  end
end
```

The pasted code will test the application routes the get request as expected. Then updates the session value and redirects back

to where the request came from. The behavior will be further examined later. As we add some markup, javascript, and more broadly test the user interface for changing the current date.

5.1.1.1 Current Date User Interface

Now that the groundwork has been laid for the `current_date` value to be set and retrieved, we need to add a User Interface to let the end-user change this. In this case, the User Interface should, at minimum, exhibit the following:

- The current date that has been set
- A link to go back one day
- A link to go ahead one day
- A date picker to change to an arbitrarily selected date

The first thing you need to do is create some markup, and for right now, we will just add it to the `app/views/activity/mine.html.erb` template. Since this template file was only a placeholder file, you can replace the contents with the following `erb` markup:

```
<div class="flex p-5" aria-label="Breadcrumb">
  <ol class="inline-flex items-center space-x-1 md:space-\
x-3">
    <li class="inline-flex items-center">
      <h2 href="#" class="inline-flex items-center font-m\
edium text-gray-700 hover:text-gray-900">
        My Activity
      </h2>
    </li>
  </ol>
</div>

<div class="flex flex-row m-auto p-5">
  <div class="basis-1/4">
    <div class="w-6 ml-auto">
```

```
      <%= link_to update_date_path(date: Date.parse(curren\
t_date) - 1.day), id: 'date_backwards', class: '' do %>
        <%= heroicon "arrow-left", class: 'inline' %>
      <% end %>
    </div>
  </div>
  <div class="basis-1/2">
    <div class="text-center" data-controller="flatpickr" \
data-flatpickr-wrap="true">
      <input id="datePickerInput" type="text" name="curre\
nt_date" style="visibility:hidden;height:.5px;width:.5px;\
" data-input/>
      <div data-toggle class="inline cursor-pointer">
        <%= heroicon "calendar", class: 'inline' %>
        <h3 class="ml-3 inline">
          <%= Date.parse(current_date).strftime("%a, %b  \
#{Date.parse(current_date).day.ordinalize}") %>
        </h3>
      </div>
    </div>
  </div>
  <div class="basis-1/4 text-left">
    <div class="w-6 mr-auto">
      <%= link_to update_date_path(date: Date.parse(curre\
nt_date) + 1.day), id: 'date_forward' do %>
        <div class="card">
          <div class="text-left">
            <%= heroicon "arrow-right", class: 'inline' %>
          </div>
        </div>
      <% end %>
    </div>
  </div>
</div>
```

So what's going on here? The first nine lines are the header area

markup to display the page's name. Inside the second main block of HTML, we start using the TailwindCSS Grid system. Using the `basis` classes to set up the sections of date controls. First and last, two smaller ones house the `last` and `next` day link. The middle grid box will display the current date and open a date picker(which will be added later in this section).

You may notice a new ERB output calling `heroicons`. This is the free icon pack we will use for this application and can be installed through a gem. Add `gem "rails_heroicon"` to your Gemfile, then run `bundle install`, and finally restarting the Rails server.

```
<div class="basis-1/4">
  <div class="w-6 ml-auto">
   <%= link_to update_date_path(date: Date.parse(current_\
date) - 1.day), id: 'date_backwards', class: '' do %>
      <%= heroicon "arrow-left", class: 'inline' %>
    <% end %>
  </div>
</div>
```

Here we are creating the first link. `link_to` is a Rails view helper that will help you quickly create dynamic links based on existing application routes. Usually, you would see something like `link_to("Visible Text", some_path)` to generate a link that would route to `some_paths` destination with the text "Visible Text". The striking thing we are doing here is that Rails also lets you pass a `block` to the `link_to` helper and inserts any markup into the body of the link. Thus, you have a whole content block as part of the link in this case. The final dynamic part of this link is its path parameter. When using Rails' path helpers, you can pass in a particular dynamic named parameter and, in our case, `date`. Here, we take the string representation of the `current_date`, parsing that into a Date object to do some Date math. This math says, "give me the date one day ago."

The last column and link are very similar except using Date math to get the *next* date:

```
<div class="basis-1/4 text-left">
  <div class="w-6 mr-auto">
    <%= link_to update_date_path(date: Date.parse(current\
_date) + 1.day), id: 'date_forward' do %>
      <%= heroicon "arrow-right", class: 'inline' %>
    <% end %>
  </div>
</div>
```

Now, for the middle column's content block explanation:

```
<div class="basis-1/2">
  <div class="text-center" data-controller="flatpickr" da\
ta-flatpickr-wrap="true">
    <input id="datePickerInput" type="text" name="current\
_date" value="<%= current_date %>" style="visibility:hidd\
en;height:.5px;width:.5px;" data-input/>
    <div data-toggle class="inline cursor-pointer">
      <%= heroicon "calendar", class: 'inline' %>
      <h3 class="ml-3 inline">
        <%= Date.parse(current_date).strftime("%a, %b  #{\
Date.parse(current_date).day.ordinalize}") %>
      </h3>
    </div>
  </div>
</div>
```

There are only a few notable mentions in this markup. First, a `data-controller="flatpickr"` attribute is added to the main box and used by a javascript date picker library, and other data attributes that will be used as a calendar picker trigger. Inside the input markup, the `current_date` is supplied into the `value` attribute

simply by string interpolation. Lastly, the `current_date` string is cast into a `Date` object to be reformatted into a new, more human-readable date representation.

Let's round this out by adding the javascript library and a handful of lines that will handle the date picker's initialization.

First, we are going to install the `flatpickr` and `stimulus-flatpickr` libraries through the importmap tool by running `bin/importmap pin stimulus-flatpickr@beta flatpickr --download`.

Once installed, we will add a new Stimulus controller, which extends the package you just installed to modify the behavior when the date selection event happens.

Add the following code:

```
import Flatpickr from 'stimulus-flatpickr'

export default class extends Flatpickr {

  connect() {
    super.connect()
  }

  change(selectedDates, _dateStr, _instance) {
    window.Turbo.visit(`/dates/${selectedDates[0].toISOSt\
ring().substring(0,10)}`)
  }
}
```

Inside the `change` function, initially defined in the Flatpickr package, the selected date is formatted to the ISO8601 format, then passed to Turbo to redirect to the date change controller to update the stored date in the session.

Now you should be able to click on the middle grid box, see a date picker show up, select a date and have the page update to the chosen date.

The last thing we will do here is write a system spec to test whether the browser behavior works as intended. Create the following file at `spec/system/date_change_spec.rb`

```
require 'rails_helper'

RSpec.describe "DateChanges", type: :system do
  login_admin

  it "changes the date when the back arrow is pressed" do
    visit activity_mine_path

    click_on(id: 'date_backwards')

    expect(page).to have_content(
      (Date.today - 1.day)
      .strftime("%a, %b #{(Date.today - 1.day).day.ordina\
lize}")
    )
  end

  it "changes the date when the forwards arrow is pressed\
" do
    visit activity_mine_path

    click_on(id: 'date_forward')

    expect(page).to have_content(
      (Date.today + 1.day)
      .strftime("%a, %b #{(Date.today + 1.day).day.ordina\
lize}")
    )
  end

  it 'should be able to visit standups and move date from\
 picker', js: true do
```

```
    visit activity_mine_path

    date = (Time.now + 1.day).strftime("%B %-d, %Y")

    find('#datePicker').click
    find(".flatpickr-day[aria-label='#{date}']", visible:\
 false).click

    expect(page).to have_content(
      (Time.now + 1.day)
      .strftime(
        "%a, %b #{(Time.now + 1.day).day.ordinalize}"
      )
    )
  end
end
```

This may seem like a big crazy test, but there are only three tests. Let's go over the first two as they don't diverge from the system specs you wrote thus far.

The first spec visits the `activity#mine` page we have created the `data change markup` on. Then, clicks on the link that goes to the next date in the future and verifies the page has changed to represent being a day further in the future.

The second spec visits the `activity#mine` page on which we created the data change markup. Then, it clicks on the link that goes back a day in the past and verifies that the page has changed to represent the previous day.

Here you will notice it is the first time we use `js: true`. This is because now that we used the system spec and web driver setup in the last chapter, we will not need any other setup!

```
it 'should be able to visit standups and move date from p\
icker', js: true do
  visit activity_mine_path

  date = (Time.now + 1.day).strftime("%B %-d, %Y")

  find('#datePicker').click
  find(".flatpickr-day[aria-label='#{date}']", visible: f\
alse).click

  expect(page).to have_content(
    (Time.now + 1.day)
    .strftime(
      "%a, %b #{(Time.now + 1.day).day.ordinalize}"
    )
  )
end
```

Let's go over this line by line. Again, this test uses the javascript driver due to the `js: true`. Then, like the tests before, it heads to the `activity#mine` page. Since the `flatpickr` library uses the date within an attribute, we generate a string that will match the attribute and be used in the test expectation later. That is then used to find a date to `click` in the opened date picker. Now, we test the page redirects back to the page it was on, as in previous tests.

One more thing we can do here to improve the modularity of the date-changing markup is to extract the markup into a partial and then include it in any template or layout we may need. This is easily accomplished by creating a new file `app/views/layouts/_date_change.html.erb`:

```
<div class="flex flex-row m-auto p-5">
  <div class="basis-1/4">
    <div class="w-6 ml-auto">
     <%= link_to update_date_path(date: Date.parse(curren\
t_date) - 1.day), id: 'date_backwards', class: '' do %>
        <%= heroicon "arrow-left", class: 'inline' %>
      <% end %>
    </div>
  </div>
  <div class="basis-1/2">
    <div class="text-center" data-controller="flatpickr" \
data-flatpickr-wrap="true" id="datePicker">
      <input id="datePickerInput" type="text" name="curre\
nt_date" value="<%= current_date %>" style="visibility:hi\
dden;height:.5px;width:.5px;" data-input/>
      <div data-toggle class="inline cursor-pointer">
        <%= heroicon "calendar", class: 'inline' %>
        <h3 class="ml-3 inline">
          <%= Date.parse(current_date).strftime("%a, %b  \
#{Date.parse(current_date).day.ordinalize}") %>
        </h3>
      </div>
    </div>
  </div>
  <div class="basis-1/4 text-left">
    <div class="w-6 mr-auto">
      <%= link_to update_date_path(date: Date.parse(curre\
nt_date) + 1.day), id: 'date_forward' do %>
        <%= heroicon "arrow-right", class: 'inline' %>
      <% end %>
    </div>
  </div>
</div>
```

If you attempt to run all of the specs right now, you will run across a few failing specs around the generated `standups` files. However,

there is actually one specific test that may stand out as failing. You will most likely see `spec/views/activity/mine.html.erb_spec.rb` failing now. This isn't due to the change made, but more on how RSpec, its mock system, and its view specs handle helpers. Thus, the view spec is unable to find `current_date`.

This also shows that while view specs can be beneficial by being faster than system specs, there may be tradeoffs with its lightweight view loading mechanisms.

With that out of the way, here are the changes you will need to make. Update the spec with the following:

```
require 'rails_helper'

RSpec.describe "activity/mine.html.erb", type: :view do
  before do
    allow(view).to receive(:current_date)
      .and_return(Date.today.strftime('%a %-d %b %Y'))
  end
  it "renders the word feed" do
    render template: 'activity/mine'
    expect(rendered).to match(/My Activity/)
  end
end
```

Due to a recent change in RSpec, you will also need to change a setting regarding view helpers and their mocks. Change `mocks.verify_partial_doubles = true` to `mocks.verify_partial_doubles = false`.

Now would be an excellent time to commit your file changes if you have not been so far, and depending on your git management style, you may have been doing this work on a feature branch. If so, you would be able to `commit` and `merge` like so:

```
git add .
git commit -am "Date Change controller and User Interface\
s"
git checkout main
git merge date-change
git push
```

5.1.2 Teams using Has Many Through

With a critical aspect of the User Interface worked out, we can now start modeling the rest of our data. There can be many different approaches to building an application, but I have always found that modeling data is almost always a great start. You will have your base models and work out how they may connect and interact with each other. Here we are on to the next significant aspect of the Daily Standup Application, `teams`.

To recap the functionality of the team model and its two functions, I will quote the beginning of the chapter:

> Teams handle configuration of two things first. First, as stated before, it is a way to group users into smaller logical groups within the account. It holds particular settings for that group, such as how notifications, reminders, and recaps are sent. Additionally, it keeps the standup meeting time and other time-related settings.

You will have a Team model with attributes that will keep team-specific data, like what time of the day is standup or what is the team's name. However, maintaining a list of users that belong to a team isn't as simple as a **list of users**. In this case, the most efficient and maintainable way to keep track of the team members would be to have another table called `team_memberships` which would hold columns for the `team_id` and `user_id`. This way, either relationship can ask `what belongs here`. For example, a user can ask what teams

am I on? with `user.teams`. Likewise, if you have a `Team` object, you can ask what users belong to this team? with `team.users`.

Throughout this section, we will create three models to hold these relationships: controllers to access and modify the data, new user interfaces to do the same, and wrap it up with some specs.

5.1.2.1 Team Model

While some files are easy to start by manually creating them, models are usually best left to using a generator. You can generally get a model with a few relationships, a migration, and a few files virtually free. Now, just before running the generator, let's go over the model attributes you may need:

Team:

- Name
- Account(it belongs_to)
- Timezone
- Days of the Week, the Standup actually happens on
- Sends Reminders? (Boolean for sending reminder notifications)
  - Reminder Time, if true
- Sends Recaps? (Boolean for sending recap notifications)
  - Recap Time, if true

This is all pretty simple stuff, just a few attribute types, and we're done. However, it looks like we discovered yet another relationship on the Teams table. Days of the Week. While it may have been easy to store days of the week as a Serialized Array or JSON column, it would leave much to be desired when looking up Teams by the day of the week. Thus, the best approach would be another `_membership` relationship.



As an exercise for your own experience, you will be prompted for a few other membership tables later in the chapter.

Let's start with the Team Model's `rails generator`:

```
rails g model Team name account:references timezone has_r\
eminder:boolean \
has_recap:boolean reminder_time:time recap_time:time
```

..and finally run the migration:

```
rails db:migrate
```

Now, we can add the `TeamMembership` to create the `User <-> Team` association:

```
rails g model TeamMembership team:references user:referen\
ces
```

Then run the migration.

```
rails db:migrate
```

While most of the migration attributes and their types have helped automatically generate model associations, the migrations leave out the ability to go right from the `Team` or `User` model to load their own relationships. Through another intermediary model, this type of association is called `Has Many Through`. It's actually relatively simple to set up. First, let's look at the `Team` model as it stands after being generated:

```
class Team < ApplicationRecord
  belongs_to :account

end
```

As you can see, based on the `account:references` in the model generator, the model has a `belongs_to :account`. You can ask the team object `.account` to get the account object. Now let's add the lines to bring in the `users` association:

```
class Team < ApplicationRecord
  belongs_to :account
  validates :account, presence: true
  validates :timezone, presence: true
  validates :name, presence: true

  has_many :team_memberships
  has_many :users, through: :team_memberships
end
```

It looks simple, right? However, using a `has_many_through` is really a two-step relationship. First, you reference the `has_many` aspect of `team_memberships` in the model. This now allows `Team` to ask, "what are my `team_memberships`?". Now, to tie users all the way back to teams, you need one more relationship. So now, we create a `has_many` with the last resource we want and tell Rails to use the existing `:team_memberships`. Next, let's add the same relationship on the opposite side:

```
class User < ApplicationRecord
  rolify
  attr_accessor :role
  belongs_to :account
  # Include default devise modules. Others available are:
  # :confirmable, :lockable, :timeoutable and :omniauthab\
le
  devise :invitable, :database_authenticatable, :register\
able,
         :recoverable, :rememberable, :trackable, :valida\
table

  has_many :team_memberships
  has_many :teams, through: :team_memberships
end
```

Now you will be able to call `team.users` and `user.teams`.

Lastly, for the `Account` model to acknowledge it's half of the `Account <-> Team` relationship, you will simply need to add `has_many :teams` to the model:

```
class Account < ApplicationRecord
  resourcify
  has_many :users
  has_many :teams
  validates :name, presence: true
end
```

5.1.2.2 DaysOfTheWeek Model

I held off the `DaysOfTheWeekMembership` model, as it has yet another twist. `DaysOfTheWeekMembership` is not entirely a two-sided `has_many_through`. `DaysOfTheWeekMembership` has one side of the membership is a static `enum`. Enum translates Days of the Week into integers to be used as IDs. The other side is the `team_id` to tie the

team's day selection into the join table. Outside of a hiccup, you will see later when saving `Team` records with updates. This seems the best way to handle non-database object static data relationships.

```
rails g model DaysOfTheWeekMembership team:references day\
:integer
```

Again, one more time, let's open up the migration file to add in UUID (I hope the Rails code development team fixes the references not generating UUID at some point.). Then, finally, you can migrate the data, `rails db:migrate`.

Then you can adjust the `DaysOfTheWeekMembership` model to include the `enum` fields:

```
class DaysOfTheWeekMembership < ApplicationRecord
  belongs_to :team
  enum day:
    [:sunday, :monday, :tuesday, :wednesday, :thursday, :\
friday, :saturday]
end
```

Then a quick update to the `Team` model to reference the new membership table:

```
class Team < ApplicationRecord
  belongs_to :account
  validates :account, presence: true
  validates :timezone, presence: true
  validates :name, presence: true

  has_many :team_memberships
  has_many :users, through: :team_memberships

  has_many :days, class_name: 'DaysOfTheWeekMembership', \
```

```
dependent: :delete_all
  accepts_nested_attributes_for :days ## <- Added Lines
end
```

We have enough models to facilitate saving a `Team` object with that in place and keep necessary attributes and associations to other models within the application. This would be an excellent place to throw in some quick model specs to ensure all model creation is and continues to work as intended.

5.1.2.3 Model Specs

One gem that makes it easy to test the existence of validators and relationships defined in the model is `shoulda-matchers`. To add this gem, insert in `gem 'shoulda-matchers'`, right under `gem 'rails-controller-testing'`. Then, run `bundle install`, and you will be all set with installing this gem.

Finally, add the following to the very bottom of `spec/rails_helper.rb`:

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

    with.library :rails
  end
end
```

Let's start with an update to the `model/team_spec.rb` as it does not have any actual tests after being generated by the Rails command:

N> Change `account nil` to `account`, `time_zone "MyString"` to `timezone "Arizona"` in `spec/factories/team.rb`, `team nil` to `team` in `spec/factories/days_of_the_week_membership.rb` and `team nil`, `user nil` to `team`, `user` in `spec/factories/team_membership.rb` to ensure the Factories are using the associations.

```
require 'rails_helper'

RSpec.describe Team, type: :model do
 it "has a valid factory" do
   expect(FactoryBot.build(:team)).to be_valid
 end

 it { is_expected.to belong_to(:account) }
 it { is_expected.to validate_presence_of(:account) }

 it { is_expected.to validate_presence_of(:name) }
 it { is_expected.to validate_presence_of(:timezone) }

 it { is_expected.to have_many(:team_memberships) }
 it { is_expected.to have_many(:users).through(:team_memb\
erships) }

 it { is_expected.to have_many(:days) }
end
```

Starting at the first spec, it is business as usual, making sure the factory and model work together as intended. Next, you are introduced to the `shoulda-matchers` syntax. This simplifies testing relationships and validators down to one line. This also removes the need to manually build and test for failed saves to test validators. With that being said, as you go down the file, the account relationship asserts it exists as a `belongs_to` relationship and is present when saving the record. Running tests for the `team_memberships` relationship and the `days` relationship with a quick `rspec spec/models/teams_spec.rb` should have 8 tests passing.

Next, we can go ahead and update the `User` model's spec with relationship specs:

```
require 'rails_helper'

RSpec.describe User, type: :model do
  context "valid Factory" do
    it "has a valid factory" do
      expect(FactoryBot.build(:user)).to be_valid
    end
  end

  it { is_expected.to have_many(:team_memberships) }
  it { is_expected.to have_many(:teams).through(:team_mem\
berships) }
end
```

Now, another quick spec is the `TeamMembership` model spec:

```
require 'rails_helper'

RSpec.describe TeamMembership, type: :model do
  it { is_expected.to belong_to(:team) }
  it { is_expected.to belong_to(:user) }
end
```

Let's keep going. Now we can update the model spec for `DaysOfTheWeekMembership`:

```
require 'rails_helper'

RSpec.describe DaysOfTheWeekMembership, type: :model do
  it { is_expected.to belong_to(:team) }
end
```

Lastly, it would make sense to update any validators to the new syntax so let's update the last test in the `Account` model spec:

```
require 'rails_helper'

RSpec.describe Account, type: :model do
  context "valid Factory" do
    it "has a valid factory" do
      expect(FactoryBot.build(:account)).to be_valid
    end
  end

  context "valid validations" do
    it "will not save without a name" do
      expect(FactoryBot.build(:account, name: nil).save).\
to be_falsey
    end
  end

  it { is_expected.to validate_presence_of(:name) }
end
```

That should do it with the model specs. You can test all of the model specs by running `rspec spec/models/`, and those specs should have all passing tests and a pending mark for the `Standup` model.

5.1.2.4 Team User Interface

We will start building the `Team` User Interface for our next bit of the application. In this case, we want to accomplish the following:

- Create a new `Team`
- View a list of `Teams`
- Show a specific `Team`
- Edit a `Team`
- Delete a `Team`
- Have the Team changes reflected elsewhere, such as the sidebar navigation

The first thing we can do here is where most UI's in Rails start by making changes in the `routes.rb` file. Since it looks like we need all of the different CRUD actions and endpoints, we can use the `resources` route helper(plus, we can always exclude action/paths we don't need later). Add `resources :teams, path: 't'`, right under `resources :account`. Whoa! What is that `path` option? When using the `resources` helper, you can specify the URL's path. To showcase that option(and make the URLs shorter), we chose a path of `t`. Just for a quick example, here is what the `edit_team_path` would look like, `/t/:id/edit`.

The next area of business is to create some controller actions to handle some specific use cases. After a few controllers, you get in the habit of just creating them yourself, which we are going to do here. Create a new file at `app/controllers/teams_controller.rb`, with the following content:

```
class TeamsController < ApplicationController
  load_and_authorize_resource except: [:create]

  def index
    @teams = visible_teams
  end

  def show
    @team = Team.includes(:users).find(params[:id])
  end

  def new
    @team = Team.new
    set_users
  end

  def create

  end
```

```
  def destroy
    @team = Team.find(params[:id])
    @team.destroy
    redirect_to teams_url, notice: 'Team was successfully\
 destroyed.'
  end

...

private

def set_users
  @account_users ||=
    (
      current_account.users.where.not(invitation_accepted\
_at: nil) +
      User.with_role(:admin, current_account)
    ).uniq
end

end
```

As you're getting familiar with Rails, I don't think we need to go in-depth on the usual controller actions, what they do, and the typical code you would find in the methods. What is out of the ordinary, though, is `visible_teams`. In a SaaS app, you find most resources need to be scoped to the current account with whom the user belongs. In this case, we are adding this method to the `app/controllers/application_controller.rb`:

```
def visible_teams
  @visible_teams ||=
    if current_user.has_role? :admin, current_account
      current_account.teams.includes(:users)
    else
      current_user.teams.includes(:users)
    end
  @visible_teams
end
```

In this method, we're memoizing the output of the conditional statement's selection of teams. If the user is an admin, they will see all `teams` that belong to an `account`. Otherwise, regular users will only see teams they are a member of. The `.includes(:users)`, means Rails makes a more efficient request to get all of the team's user's in one SQL query instead of loading them one by one as the `teams` are iterated upon inside the `index` template's loop.

Now, let's add some `view` templates. First, you will need to create a new folder `app/views/teams`, and then add in `app/views/teams/index.html.erb`:

```
<div class="flex p-5" aria-label="Breadcrumb">
  <ol class="inline-flex items-center space-x-1 md:space-\
x-3">
    <li class="inline-flex items-center">
      <h2 href="#" class="inline-flex items-center font-m\
edium text-gray-700 hover:text-gray-900">
        Teams
      </h2>
    </li>
  </ol>
</div>

<div class="flex justify-center">
  <div class="p-4">
```

```
    <div class="grid grid-cols-4 gap-4">
      <%= render @teams.to_a.push(Team.new) %>
    </div>
  </div>
</div>
```

This is a bit more advanced than a regular iteration, so I'll go through the interesting bits. Also, we're pushing a dummy `team` on the line. This is done so that the count of teams can still be correctly placed into the CSS/Tailwind grid. The grid is established by using the grid classes and specifying a row of `4` columns. Later in the template, an add button is placed by using the `dummy` object.

Ok, let's look at the `_team` partial now:

```
<% if team.id.blank? %>
    <%= link_to new_team_path, { id: 'new_team'} do %>
        <%= heroicon "plus", class: 'h-20 w-20' %>
    <% end %>
<% else %>
  <div class="block p-6 max-w-sm bg-white rounded-lg bord\
er border-gray-200 shadow-md hover:bg-gray-100">
    <div>
      <%= link_to team_path(team) do %>
        <h5 class="inline mb-2 text-2xl font-bold trackin\
g-tight text-gray-900"><%= team.name %></h5>
      <% end %>
      <div class="float-right">
        <%= link_to edit_team_path(team) do %>
          <%= heroicon "pencil-alt", class: 'inline' %>
        <% end %>
        <%= button_to team_path(team), {form_class: 'inli\
ne', method: :delete, form: {data: { turbo_confirm: "Are \
you sure? Deleting this team is permanent"}}} do %>
          <%= heroicon "trash", class: 'inline' %>
        <% end %>
```

```
      </div>
    </div>
    <%= link_to team_path(team) do %>
      <h6>Description: </h6>
      <p>
        <%=team.description %>
      </p>
      <h6> Members </h6>
      <div class="team-members">
        <% if team.users.blank? %>
          <p>
            No Users have been added to the team
          </p>
        <% else %>
          <%=team.users.pluck(:name).join(', ') %>
        <% end %>
      </div>
      <h6>Standup Details</h6>
      <div class="row m-t-sm">
        <div class="col-sm-6">
          <%= "On days #{team.days.pluck(:day).map(&:huma\
nize).join(', ')}" %>
        </div>
        <div class="col-sm-6">
          <%="Reminder is #{team.reminder_time.strftime("\
at %I:%M%p")}" %>
          <br />
          <%="Recap is #{team.recap_time.strftime("at %I:\
%M%p")}" %>
        </div>
      </div>
    <% end %>
  </div>
<% end %>
```

This template is a lot more straightforward. A few things to call

out. You may see `.pluck` in a few places, which allows you to pull just one attribute from a collection of objects. Also, you see the `.join`-syntax, which takes an array and joins the elements into one string, with the argument being the separator. There is one specific curveball in this template, though. You might have even seen the error if you tried loading the page. The view template is looking for a `team.description`, which does not exist. Also, the description field will be where the end-user can write a long-form piece of text describing the team.

This seems like an excellent place to go over how simple it is to add a column to an existing table in Rails. Another area where convention over configuration can make your life easier. Head over to your terminal and run:

```
rails g migration AddDescriptionToTeams description:text
rails db:migrate
```

That's it! (Re)load the page, and you see an add button and maybe even a team if you have been playing in the (yet to be covered), Rails Console.

Now is a better time than ever to go over the Rails Console. The Ruby language comes with Interactive Ruby(IRB for short). Rails takes that further and loads your entire Rails application into an IRB session. You can drop into the console via `rails c` and literally modify your application. Do you want to add a record on the fly, essentially on the command line? Do it in `rails c`. In fact, here is how you might create a `team` via Rails Console:

```
3.0.4 :001 > Team.create(name: "Name", account_id: Accoun\
t.first.id, timezone: 'Arizona', has_reminder: true, has_\
recap: true,
reminder_time: "2000-01-01 05:00:00", recap_time: "2000-0\
1-01 06:00:00")
  Account Load (6.7ms)  SELECT "accounts".* FROM "account\
s" ORDER BY "accounts"."created_at" ASC, "accounts"."id" \
ASC LIMIT $1  [["LIMIT", 1]]
  TRANSACTION (4.7ms)  BEGIN
  Account Load (0.2ms)  SELECT "accounts".* FROM "account\
s" WHERE "accounts"."id" = $1 LIMIT $2  [["id", "a15d4141\
-2cbd-4a36-9cf6-2e0e4ead7c71"], ["LIMIT", 1]]
  Team Create (4.3ms)  INSERT INTO "teams" ("name", "acco\
unt_id", "timezone", "has_reminder", "has_recap", "remind\
er_time", "recap_time", "created_at", "updated_at", "desc\
ription") VALUES ($1, $2, $3, $4, $5, $6, $7, $8, $9, $10\
) RETURNING "id"  [["name", "Name"], ["account_id", "a15d\
4141-2cbd-4a36-9cf6-2e0e4ead7c71"], ["timezone", "Arizona\
"], ["has_reminder", true], ["has_recap", true], ["remind\
er_time", "05:00:00"], ["recap_time", "06:00:00"], ["crea\
ted_at", "2022-04-25 21:53:11.963477"], ["updated_at", "2\
022-04-25 21:53:11.963477"], ["description", nil]]
  TRANSACTION (0.4ms)  COMMIT
 =>
#<Team:0x00007fb29a246b58
 id: "a2fd5014-3e91-4101-8aa6-10c924a64989",
 name: "Name",
 account_id: "a15d4141-2cbd-4a36-9cf6-2e0e4ead7c71",
 timezone: "Arizona",
 has_reminder: true,
 has_recap: true,
 reminder_time: Sat, 01 Jan 2000 05:00:00.000000000 UTC +\
00:00,
 recap_time: Sat, 01 Jan 2000 06:00:00.000000000 UTC +00:\
00,
```

```
 created_at: Mon, 25 Apr 2022 21:53:11.963477000 UTC +00:\
00,
 updated_at: Mon, 25 Apr 2022 21:53:11.963477000 UTC +00:\
00,
 description: nil>
```

You can run nearly any code you would see in your Rails Application in Rails Console. For example, find an object, run `.where` statements, etc. However, you typically do not find artifacts of the request/response cycle. That said, `rails c` is also a perfect place to test out code and more!

Ok, back to templates. Now that you may or may not have a list of `Teams`, we will create the interface for creating new teams. To recap from earlier, in Rails it is typical to have one `_form` partial that passes in either a blank new model object `@team = Team.new` or an existing model object `@team = Team.find(params[:id])`. The form_with helper deciphers the object type and sets up all form methods and paths for you.

```
<div class="flex p-5" aria-label="Breadcrumb">
  <ol class="inline-flex items-center space-x-1 md:space-\
x-3">
    <li class="inline-flex items-center">
      <h2 href="#" class="inline-flex items-center font-m\
edium text-gray-700 hover:text-gray-900">
        Teams
      </h2>
    </li>
    <li class="inline-flex items-center">
      <h2 href="#" class="inline-flex items-center font-m\
edium text-gray-700 hover:text-gray-900">
        <svg class="w-6 h-6 text-gray-400" fill="currentC\
olor" viewBox="0 0 20 20" xmlns="http://www.w3.org/2000/s\
vg"><path fill-rule="evenodd" d="M7.293 14.707a1 1 0 010-\
1.414L10.586 10 7.293 6.707a1 1 0 011.414-1.414l4 4a1 1 0\
```

```
 010 1.414l-4 4a1 1 0 01-1.414 0z" clip-rule="evenodd"></\
path></svg>
        New
      </h2>
    </li>
  </ol>
</div>
<div class="flex justify-center">
  <div class="p-4 w-4/5 bg-white rounded-lg border border\
-gray-200 shadow-md sm:p-6 lg:p-8">
    <h5 class="text-xl font-bold leading-none text-gray-9\
00">Add a new team:</h5>
     <%= render 'form' %>
  </div>
</div>


<%= form_with model: @team, id: 'team-form', html: {class\
: 'space-y-6'}  do |f| %>
  <% if @team.errors.any? %>
    <div class="text-red-700">
      <h4>
        <%= "#{pluralize(@team.errors.count, "error")} pr\
ohibited this team from being saved:" %>
      </h4>
      <%= @team.errors.full_messages.map{|msg|content_tag\
(:li, msg)}.join %>
    </div>
  <% end %>
  <div>
    <%= f.label :name, class: 'block mb-2 text-sm font-me\
dium text-gray-900' %>
    <%= f.text_field :name, placeholder: 'Enter a Team na\
me', class: 'bg-gray-50 border border-gray-300 text-gray-\
900 text-sm rounded-lg focus:ring-blue-500 focus:border-b\
lue-500 block w-full p-2.5' %>
```

```
  </div>
  <div>
    <%= f.label :description, class: 'block mb-2 text-sm \
font-medium text-gray-900' %>
    <%= f.text_area :description, placeholder: 'Enter a t\
eam description', class: 'bg-gray-50 border border-gray-3\
00 text-gray-900 text-sm rounded-lg focus:ring-blue-500 f\
ocus:border-blue-500 block w-full p-2.5' %>
  </div>
  <div>
    <%= f.label :timezone, class: 'block mb-2 text-sm fon\
t-medium text-gray-900' %>
    <%= f.time_zone_select :timezone, ActiveSupport::Time\
Zone.us_zones, {}, class: 'bg-gray-50 border border-gray-\
300 text-gray-900 text-sm rounded-lg focus:ring-blue-500 \
focus:border-blue-500 block w-full p-2.5' %>
  </div>
  <div>
    <%= f.label "","Days of the week", class: 'block mb-2\
 text-sm font-medium text-gray-900' %>
      <% DaysOfTheWeekMembership.days.each do |day_name, \
i| %>
        <div class="inline">
          <%= check_box_tag "team[days][]", day_name,\
            f.object.days.pluck(:day).include?(day_name),\
            {id: "team_days_#{day_name.downcase}", class:\
 'w-4 h-4 text-blue-600 bg-gray-100 border-gray-300 round\
ed focus:ring-blue-500'} %>
          <%= f.label "days_#{day_name.downcase}",\
            day_name.capitalize,\
            {style: "padding-left: 0px"} %>
        </div>
      <% end %>
      <p id="helper-text-explanation" class="mt-2 text-sm\
 text-gray-500">The days of the week to hold status repor\
```

```
ts.</p>
  </div>
  <div class="py-4">
    <div class="w-full border-t border-gray-300"></div>
  </div>
  <div>
  <%= f.label "","Users", class: 'block mb-2 text-sm font\
-medium text-gray-900' %>
    <% @account_users.each do |user| %>
      <div class="inline">
        <%= check_box_tag "team[user_ids][]", user.id,\
          f.object.users.pluck(:id).include?(user.id),\
          {id: "team_user_#{user.id}", class: 'w-4 h-4 te\
xt-blue-600 bg-gray-100 border-gray-300 rounded focus:rin\
g-blue-500'} %>
          <%= f.label "user_#{user.id}", user.email, {sty\
le: "padding-left: 0px"} %>
        </div>
    <% end %>
    <p id="helper-text-explanation" class="mt-2 text-sm t\
ext-gray-500">Users to be included in this team.</p>
  </div>
  <div class="pt-4">
    <div class="w-full border-t border-gray-300"></div>
  </div>
  <h6 class="text-center"> Notification Settings </h6>
  <div data-controller="visibility">
    <div>
      <%= label_tag '', "Send Reminder Notification?",\
        class: 'block mb-2 text-sm font-medium text-gray-\
900' %>
        <div class="">
          <%= f.check_box :has_reminder, data: {action: "\
visibility#toggleTargets"}, class: 'w-4 h-4 text-blue-600\
 bg-gray-100 border-gray-300 rounded focus:ring-blue-500'\
```

```
 %>
          <%= label_tag 'team_has_reminder', " ".html\
_safe, class: 'inline' %>
        </div>
      <p id="helper-text-explanation" class="mt-2 text-sm\
 text-gray-500">Send a reminder notification on days the \
team has reports.</p>
    </div>
    <div class="pt-4" data-visibility-target="hideable" i\
d="reminder-time-box" hidden="<%= f.object.has_reminder ?\
 '' : 'hidden' %>">
      <%= f.label :reminder_time, class: 'block mb-2 text\
-sm font-medium text-gray-900' %>
      <%= f.time_select :reminder_time, {ampm: true, minu\
te_step: 15},\
        {style:'    display: inline;width: 10%;',class: '\
bg-gray-50 border border-gray-300 text-gray-900 text-sm r\
ounded-lg focus:ring-blue-500 focus:border-blue-500 block\
 w-full p-2.5'} %>
      <p id="helper-text-explanation" class="mt-2 text-sm\
 text-gray-500">Time of the day to send report reminder i\
n the selected time zone.</p>
    </div>
  </div>
  <div data-controller="visibility">
    <div>
      <%= label_tag '', "Send Recap Notification?",\
        class: 'block mb-2 text-sm font-medium text-gray-\
900' %>
        <div class="">
          <%= f.check_box :has_recap, data: {action: "vis\
ibility#toggleTargets"}, class: 'w-4 h-4 text-blue-600 bg\
-gray-100 border-gray-300 rounded focus:ring-blue-500' %>
          <%= label_tag 'team_has_recap', " ".html_sa\
fe, class: 'inline' %>
```

```
        </div>
      <p id="helper-text-explanation" class="mt-2 text-sm\
 text-gray-500">Send a recap notification on days the tea\
m has reports.</p>
    </div>
    <div class="pt-4" data-visibility-target="hideable" i\
d="recap-time-box" hidden="<%= f.object.has_recap ? '' : \
'hidden' %>">
      <%= f.label :recap_time, class: 'block mb-2 text-sm\
 font-medium text-gray-900' %>
      <%= f.time_select :recap_time, {ampm: true, minute_\
step: 15},\
        {style:'    display: inline;width: 10%;',class: '\
bg-gray-50 border border-gray-300 text-gray-900 text-sm r\
ounded-lg focus:ring-blue-500 focus:border-blue-500 block\
 w-full p-2.5'} %>
      <p id="helper-text-explanation" class="mt-2 text-sm\
 text-gray-500">Time of the day to send report recap in t\
he selected time zone.</p>
    </div>
  </div>
  <%= f.submit "Save", class: "text-white bg-blue-700 hov\
er:bg-blue-800 focus:ring-4 focus:outline-none focus:ring\
-blue-300 font-medium rounded-lg text-sm w-full sm:w-auto\
 px-5 py-2.5 text-center" %>
<% end %>
```

Let's go through this view in sections:

- Here, you can see the `form_with` helper method utilized with an optional `HTML` parameter being included to add a class name to the form to make the HTML form work with the TailwindCSS styling.
- The lines are checking for any model validation errors to be displayed back on the form.

- Next are the lines featuring essential form elements and markup that is valid in TailwindCSS

    - After, we are enumerating over the Days of the Week values from the `DaysOfTheWeekMembership` days enums. A checkbox is created every day (and label) to select 0-7 days of the week. Also, some specific checkbox classes are being added to use an enhanced checkbox display.
    - We're doing nearly the same thing here with users on the account. The values here will be automatically saved into the `TeamMembership` table by using a named convention `user_ids`.

- Here are the boolean checkboxes for reminders and recaps, along with time selection boxes to be displayed if the checkbox is checked. While this code is correct for existing `teams` with the options set to true, there will need to be a few lines of Javascript added to make the time selection fields show up dynamically. The code uses StimulusJS and the data attribute conventions to toggle the visibilty on checkbox changes:

```
import { Controller } from "@hotwired/stimulus"

export default class extends Controller {
  static targets = [ "hideable" ]

  toggleTargets() {
    this.hideableTargets.forEach((el) => {
      el.hidden = !el.hidden
    });
  }
}
```

Last, there is a `Save` button to trigger the form submission on the last few lines, which uses the `create` method in the `TeamsController`.

With the new `team` markup in place, we can begin on the `create` action in the `TeamsController` to create the new `team` database record. Here is a copy of that method:

```
class TeamsController < ApplicationController
  load_and_authorize_resource except: [:create]

...

  def create
    @team = Team.new(team_params.except("days"))
    @team.account_id = current_account.id
    @team.days = days
    convert_zone_times_to_utc
    authorize!(:create, @team)

    if @team.save
      redirect_to teams_path, notice: 'Team was successfu\
lly created.'
    else
      set_users
      render :new, status: :unprocessable_entity
    end
  end

  private

  def team_params
    params.require(:team).permit(:name, :description, :ti\
mezone, :has_reminder,
    :has_recap, :reminder_time, :recap_time, :days=> [], \
:user_ids=> [])
  end

  def set_users
    @account_users ||=
```

```
      (
        current_account.users.where.not(invitation_accept\
ed_at: nil) +
        User.with_role(:admin, current_account)
      ).uniq
  end

  def days
    params[:team][:days]
    &.map do |day|
      DaysOfTheWeekMembership.new(
        team_id: @team.id,
        day: day
      )
    end || []
  end

  def convert_zone_times_to_utc
      convert_reminder
      convert_recap
    end

  def convert_reminder
    return nil unless @team.reminder_time && @team.has_re\
minder
    @team.reminder_time =
      ActiveSupport::TimeZone[@team.timezone]
      .parse(@team.reminder_time.to_s[11..18])
      .utc
  end

  def convert_recap
    return nil unless @team.recap_time && @team.has_recap
    @team.recap_time =
      ActiveSupport::TimeZone[@team.timezone]
```

```
      .parse(@team.recap_time.to_s[11..18])
      .utc
  end

end
```

Now that we have the form in place, we can add the create action. Plus, you will need to add auxiliary private methods(the `...` in the snippet signifies those methods are at the end of the class). The methods will help construct safe parameters and build `DaysOfTheWeekMembership` objects. But, you may be asking, why make `DaysOfTheWeekMembership` objects when other memberships aren't needed? This is one of the few pitfalls of not having an actual ActiveRecord model on the other end of the `has_many through` relationship. Thus, you have to build the objects up front and include them when saving the team object.

Also, what is any web application without having to mess with timezones? In this case, the app will set and receive the times as UTC. Yet, you will need to add the methods above to convert into UTC. Including putting the default time zone settings in the `config/application.rb` file(replacing the main block):

```
module StandupApp
  class Application < Rails::Application

    config.time_zone = 'UTC'
    config.active_record.default_timezone = :utc
    config.active_record.time_zone_aware_types = [:dateti\
me, :time]

    # Initialize configuration defaults for originally ge\
nerated Rails version.
    config.load_defaults 7.0

    # Configuration for the application, engines, and rai\
```

```
lties goes here.
    #
    # These settings can be overridden in specific enviro\
nments using the files
    # in config/environments, which are processed later.
    #
    # config.time_zone = "Central Time (US & Canada)"
    # config.eager_load_paths << Rails.root.join("extras")

    # Don't generate system test files.
    config.generators.system_tests = nil
  end
end
```

Additionally, on one of the few occasions, due to how the `DaysOfTheWeekMembership` works, we will need to wait until the `@team` object is managed to authorize the resource. Situations like this are few and far between.

Outside of that, we're making sure the `current_account` gets added as the `account_id` to ensure that the relationship gets all synced up.

Editing and updating the `team` is much of the same; first edit action...:

```
class TeamsController < ApplicationController
  load_and_authorize_resource except: [:create]

  ...

  def edit
    @team = Team.find(params[:id])
    set_users
  end

  ...
```

```
end
```

and then the `edit.html.erb` file.

```
<div class="flex p-5" aria-label="Breadcrumb">
  <ol class="inline-flex items-center space-x-1 md:space-\
x-3">
    <li class="inline-flex items-center">
      <h2 href="#" class="inline-flex items-center font-m\
edium text-gray-700 hover:text-gray-900">
        Teams
      </h2>
    </li>
    <li class="inline-flex items-center">
      <h2 href="#" class="inline-flex items-center font-m\
edium text-gray-700 hover:text-gray-900">
        <svg class="w-6 h-6 text-gray-400" fill="currentC\
olor" viewBox="0 0 20 20" xmlns="http://www.w3.org/2000/s\
vg"><path fill-rule="evenodd" d="M7.293 14.707a1 1 0 010-\
1.414L10.586 10 7.293 6.707a1 1 0 011.414-1.414l4 4a1 1 0\
 010 1.414l-4 4a1 1 0 01-1.414 0z" clip-rule="evenodd"></\
path></svg>
        Edit
      </h2>
    </li>
  </ol>
</div>
<div class="flex justify-center">
  <div class="p-4 w-4/5 bg-white rounded-lg border border\
-gray-200 shadow-md sm:p-6 lg:p-8">
    <h5 class="text-xl font-bold leading-none text-gray-9\
00">Edit <%= @team.name %>:</h5>
     <%= render 'form' %>
  </div>
</div>
```

As mentioned before, the `_form` partial takes both new and edit scenarios. So, we can jump right to the update action on the controller!

```
class TeamsController < ApplicationController
  load_and_authorize_resource except: [:create]
  around_action :use_time_zone, only: [:edit]

...

  def update
    @team = Team.find(params[:id])
    @team.attributes = team_params.except('days')
    @team.days = days
    convert_zone_times_to_utc

    if @team.save
      redirect_to teams_url, notice: 'Team was successful\
ly updated.'
    else
      set_users
      render :edit, status: :unprocessable_entity
    end
  end

  ...

  private

  ...


  def use_time_zone(&block)
    Time.use_zone(@team.timezone, &block)
  end
end
```

First, due to the addition of the timezone, we will need to use a new action filter. This filter will set the timezone for the time-select boxes based on the team's time zone. The use of the `around_action` is interesting here, as this filter lets you yield the action through a method you define. In this case, through a Time method that allows code within a block to be applied to a timezone. Additionally, the update method also calls the private methods to adjust the timezone.

This update method ends up being much the same as the create action. The real difference is the `Team` object lookup and the `.update` method call.

You may have noticed a few calls to `team_path(@team)` of the `show` template. So, without any more waiting, let's get the `show` template in there:

```
<div class="flex p-5" aria-label="Breadcrumb">
  <ol class="inline-flex items-center space-x-1 md:space-\
x-3">
    <li class="inline-flex items-center">
      <h2 href="#" class="inline-flex items-center font-m\
edium text-gray-700 hover:text-gray-900">
        Teams
      </h2>
    </li>
    <li class="inline-flex items-center">
      <h2 href="#" class="inline-flex items-center font-m\
edium text-gray-700 hover:text-gray-900">
        <svg class="w-6 h-6 text-gray-400" fill="currentC\
olor" viewBox="0 0 20 20" xmlns="http://www.w3.org/2000/s\
vg"><path fill-rule="evenodd" d="M7.293 14.707a1 1 0 010-\
1.414L10.586 10 7.293 6.707a1 1 0 011.414-1.414l4 4a1 1 0\
 010 1.414l-4 4a1 1 0 01-1.414 0z" clip-rule="evenodd"></\
path></svg>
        <% @team.name %>
      </h2>
```

```
    </li>
  </ol>
</div>

<%= render partial: 'layouts/components/date_change' %>

<div class="flex justify-center">
  <div class="p-4 w-4/5 bg-white rounded-lg border border\
-gray-200 shadow-md sm:p-6 lg:p-8">
    <div class="float-right">
      <% if current_user.has_role? :admin, current_accoun\
t %>
        <%= link_to edit_team_path(@team), class: 'inline\
' do %>
          <%= heroicon "pencil-alt", class: 'inline' %>
        <% end %>
        <%= button_to team_path(@team), {form_class: 'inl\
ine', method: :delete, form: {data: { turbo_confirm: "Are\
 you sure? Deleting this team is permanent"}}} do %>
          <%= heroicon "trash", class: 'inline' %>
        <% end %>
      <% end %>
    </div>
    <h3 id="team-settings"><%=@team.name %> Team Settings\
</h3>
    <p>
      <span>Users:</span>
      <%= @team.users.map(&:name).join(", ") %>
      <% if @team.has_reminder %>
        <span>| Reminder at: </span>
        <%= "#{@team.reminder_time.in_time_zone(current_u\
ser.time_zone)\
          .strftime("%I:%M %p %Z")}\
          (#{@team.reminder_time.in_time_zone(current_use\
r.time_zone)\
```

```
          .strftime("%H:%M")})" %>
      <% end %>
      <% if @team.has_recap %>
        <span>| Recap at: </span>
        <%= "#{@team.recap_time.in_time_zone(current_user\
.time_zone)\
        .strftime("%I:%M %p %Z")}\
        (#{@team.recap_time.in_time_zone(current_user.tim\
e_zone)\
        .strftime("%H:%M")}) " %>
      <% end %>
    </p>
  </div>
</div>
```

- `= render partial: 'layouts/components/date_change'` is being used to supply a date change widget on the page. This is the same one created on the `Dashboard` page.
- `- if current_user.has_role? :admin, current_account` is being used to gate access to edit and delete pages. This can also be controlled with CanCanCan permissions.
- The times use the `in_time_zone` method in the chain to display the times in the end user's selected time_zone.

Phew! We made it through the functionality, the models, the controller, and all the markup. Let's write some tests and get on to some more of the Business Logic in this application. Earlier in the chapter, we wrote some model specs. Next, let's write a suite of controller specs for `TeamsController` and a suite of system specs to make sure the pages behave as they should(and continue to act that way).

5.1.2.5 Team Specs

Here is the request spec file:

```
require 'rails_helper'

RSpec.describe "Teams", type: :request do
  login_admin

  let(:account) {@admin.account}

  let(:valid_attributes) do
    {
      name: Faker::Team.name,
      description: Faker::Company.catch_phrase,
      account_id: account.id,
      timezone: 'Arizona',
      days: ['monday']
    }
  end

  let(:invalid_attributes) do
    {
      name: nil,
      description: Faker::Company.catch_phrase,
      account_id: account.id,
      timezone: 'Arizona'
    }
  end

  describe "GET #index" do
    it "assigns all teams as @teams" do
      team = FactoryBot.create(:team, account_id: account\
.id)
      get teams_path, params: {}
      expect(assigns(:teams)).to eq([team])
    end
  end
```

```
  describe "GET #show" do
    it "assigns the requested team as @team" do
      team = FactoryBot.create(:team, account_id: account\
.id)
      get team_path(team), params: {id: team.to_param}
      expect(assigns(:team)).to eq(team)
    end
  end

  describe "GET #new" do
    it "assigns a new team as @team" do
      get new_team_path, params: {}
      expect(assigns(:team)).to be_a_new(Team)
    end
  end

  describe "GET #edit" do
    it "assigns the requested team as @team" do
      team = FactoryBot.create(:team, account_id: account\
.id)
      get edit_team_path(team), params: {id: team.to_para\
m}
      expect(assigns(:team)).to eq(team)
    end
  end

  describe "POST #create" do
    context "with valid params" do
      it "creates a new Team" do
        expect {
          post teams_path, params: {team: valid_attribute\
s}
        }.to change(Team, :count).by(1)
      end
```

```
      it "assigns a newly created team as @team" do
        post teams_path, params: {team: valid_attributes}
        expect(assigns(:team)).to be_a(Team)
        expect(assigns(:team)).to be_persisted
      end

      it "redirects to the teams#index" do
        post teams_path, params: {team: valid_attributes}
        expect(response).to redirect_to(teams_path)
      end
    end

    context "with invalid params" do
      it "assigns a newly created but unsaved team as @te\
am" do
        post teams_path, params: {team: invalid_attribute\
s}
        expect(assigns(:team)).to be_a_new(Team)
      end

      it "re-renders the 'new' template" do
        post teams_path, params: {team: invalid_attribute\
s}
        expect(response).to render_template("new")
      end
    end
  end

  describe "PUT #update" do
    context "with valid params" do
      let(:new_attributes) do
        {
          name: Faker::Team.name,
          description: Faker::Company.catch_phrase,
          timezone: 'Arizona',
```

```
          days: ['monday']
        }
      end

      it "updates the requested team" do
        team = FactoryBot.create(:team, account_id: accou\
nt.id)
        put team_path(team), params: {id: team.to_param, \
team: new_attributes}
        team.reload
        expect(team.updated_at).to be > team.created_at
      end

      it "assigns the requested team as @team" do
        team = FactoryBot.create(:team, account_id: accou\
nt.id)
        put team_path(team), params: {id: team.to_param, \
team: valid_attributes}
        expect(assigns(:team)).to eq(team)
      end

      it "redirects to the teams" do
        team = FactoryBot.create(:team, account_id: accou\
nt.id)
        put team_path(team), params: {id: team.to_param, \
team: valid_attributes}
        expect(response).to redirect_to(teams_path)
      end
    end

    context "with invalid params" do
      it "assigns the team as @team" do
        team = FactoryBot.create(:team, account_id: accou\
nt.id)
        put team_path(team), params: {id: team.to_param, \
```

```
team: invalid_attributes}
        expect(assigns(:team)).to eq(team)
      end

      it "re-renders the 'edit' template" do
        team = FactoryBot.create(:team, account_id: accou\
nt.id)
        put team_path(team), params: {id: team.to_param, \
team: invalid_attributes}
        expect(response).to render_template("edit")
      end
    end
  end

  describe "DELETE #destroy" do
    it "destroys the requested team" do
      team = FactoryBot.create(:team, account_id: account\
.id)
      expect {
        delete team_path(team), params: {id: team.to_para\
m}
      }.to change(Team, :count).by(-1)
    end

    it "redirects to the teams list" do
      team = FactoryBot.create(:team, account_id: account\
.id)
      delete team_path(team), params: {id: team.to_param}
      expect(response).to redirect_to(teams_url)
    end
  end

end
```

While relatively large at about 140 lines, this spec file is simple. It merely goes through the standard controller actions and tests

against the typical behaviors. In addition, a few stubs are being created with `let` to set up the data needed for tests near the top. These tests are meant to cover most or all of the critical code paths in the controller.

A few system specs will need to be made to test the browser's behavior. In this case, it's best to split them out into 3 spec files: Test New Team workflow, update team workflow, and the index/show workflow. Let's start with the index/show spec:

```
require 'rails_helper'

RSpec.describe "teams/new", type: :system do
  login_admin

  let!(:team) { FactoryBot.create(:team, account_id: @adm\
in.account.id) }

  it "should be able to visit the team index and see at l\
east one team" do
    visit teams_path

    expect(current_path).to eql(teams_path)
    expect(page).to have_content("Teams")
    expect(page).to have_content("#{team.name}")
  end

  it "should click team and see show page" do
    visit teams_path
    click_link(href: team_path(team), match: :first)

    expect(current_path).to eql(team_path(team))
    expect(page).to have_content("#{team.name}")
  end
end
```

As you can see, we're testing the happy path that will display the one `team` on the index page, as well as the `team` itself on the `show` template. Now onto the `new_spec`:

```
require 'rails_helper'

RSpec.describe "teams/new", type: :system do
  login_admin

  context 'click and new the team' do
    it 'should click new and go to new page' do
      visit teams_path
      click_link(id: 'new_team')

      expect(page).to have_current_path(new_team_path)
    end

    context 'and on the new page' do
      it 'and new the team successfully' do
        visit teams_path
        click_link(id: 'new_team')

        within '#team-form' do
          fill_in 'team_name', with: 'Test'
        end

        click_button('Save')
        expect(page).to have_current_path(teams_path)
        expect(page).to have_content(Team.last.name.to_s)
      end

      it 'and fail to new the team successfully' do
        visit teams_path
        click_link(id: 'new_team')

        within '#team-form' do
```

```
          fill_in 'team_name', with: ''
        end

        click_button('Save')
        expect(page).to have_current_path(teams_path)
        expect(page).to have_content("Name can't be blank\
")
      end
    end
  end
end
```

And here, we have a spec testing that the `new` form renders, fills in, and creates a `team` and fails to create a `team` when the validations are not met. This is much like the `user` forms tested in Chapter 4. Lastly, a very similar spec, `edit_spec`, to test out the `edit` workflow:

```
require 'rails_helper'

RSpec.describe "teams/edit", type: :system do
  login_admin

  let!(:team) { FactoryBot.create(:team, account_id: @adm\
in.account.id) }

  context 'click and edit the team' do
    it 'should click team, edit and go to edit page' do
      visit teams_path
      click_link(href:team_path(team), match: :first)
      click_link(href: edit_team_path(team), match: :firs\
t)

      expect(page).to have_current_path(edit_team_path(te\
am))
      expect(page).to have_content(team.name.to_s)
```

```
    end

    context 'click on the edit page' do
      it 'and edit the team successfully' do
        visit teams_path
        click_link(href:team_path(team), match: :first)
      click_link(href: edit_team_path(team), match: :firs\
t)

        within '#team-form' do
          fill_in 'team_name', with: 'Test'
        end

        click_button('Save')
        expect(page).to have_current_path(teams_path)
        expect(page).to have_content('Test')
      end

      it 'and fail to edit the team successfully' do
        visit teams_path
        click_link(href:team_path(team), match: :first)
        click_link(href: edit_team_path(team), match: :fi\
rst)

        within '#team-form' do
          fill_in 'team_name', with: ''
        end

        click_button('Save')
        expect(page).to have_current_path(team_path(team))
        expect(page).to have_content("Name can't be blank\
")
      end
    end
  end
```

```
end
```

Finally, we test the' edit' workflow using typical system spec patterns in this spec file.

This concludes the Team portion of your application(for now). Now is a perfect time to commit your updates. Afterward, you will switch to the main branch and merge into all of the `teams` branches you have worked on.

```
git add .
git commit -am "Add teams and all auxiliary models and ma\
rkup"
git checkout main
git merge teams
git push
```

5.1.3 Navigating better navigation

5.1.3.1 Navigating better navigation

Now that we have a few internal separate sections of our application, such as `teams`, `account users`, `user profile`, it is time to give the navigation some attention.

Here is a general overview of the changes:

- Make the sidebar only display when there is a `team` context. Otherwise, there is no sidebar.
- Have the sidebar display `team` information and links.
- Update the top navigation to show a link to the `team` and `standup` index.
- Upgrade the navigation to only keep the `notifications` placeholder and `user` dropdown
- Update the user dropdown links and style

While this seems like a lot, we're reducing the template's markup footprint and keeping some basic information and links.

Next up is the changes to the sidebar:

```
<% if @team.present? && !@team.id.nil? %>
  <div class="bg-gray-800 shadow-xl h-16 fixed bottom-0 m\
t-12 md:relative md:h-screen z-10 w-full md:w-48">

    <div class="md:mt-12 md:w-48 md:fixed md:left-0 md:to\
p-0 content-center md:content-start text-left justify-bet\
ween">
      <ul class="list-reset flex flex-row md:flex-col py-\
0 md:py-3 px-1 md:px-2 text-center md:text-left">
        <li class="mr-3 flex-1">
          <h3 class=" pl-3 pb-1 md:pb-0 text-lg text-gray\
-600 md:text-gray-400 block md:inline-block">Team: <%= @t\
eam.name %>
          </h3>
        </li>
        <li class="mr-3 flex-1">
          <%= link_to team_path(@team), {class: "block py\
-1 md:py-3 pl-1 align-middle text-gray-300 no-underline h\
over:text-white border-b-2 border-gray-800 hover:border-w\
hite", id: "edit_#{@team.id}_sidebar"} do %>
            <%= heroicon 'identification', class: 'ml-6  \
inline'%>
            <span class=""> Details</span>
          <% end %>
        </li>
        <li class="mr-3 flex-1">
          <%= link_to edit_team_path(@team), {class: "blo\
ck py-1 md:py-3 pl-1 align-middle text-gray-300 no-underl\
ine hover:text-white border-b-2 border-gray-800 hover:bor\
der-white", id: "edit_#{@team.id}_sidebar"} do %>
            <%= heroicon 'cog', class: 'ml-6  inline'%>
```

```
            <span class=""> Settings</span>
          <% end %>
        </li>
        <li class="mr-3 flex-1">
          <a href=<%="/t/#{@team.id}/s"%> class="block py\
-1 md:py-3 pl-1 align-middle text-gray-300 no-underline h\
over:text-white border-b-2 border-gray-800 hover:border-w\
hite">
            <%= heroicon 'collection', class: 'ml-6  inli\
ne'%>
            <span class=""> Standups</span>
          </a>
        </li>
      </ul>
    </div>
  </div>
<% end %>
```

We're switching the sidebar `on` in the presence of a `@team` instance variable, which checks that there is an actual team(not a new blank object). Otherwise, a few links are being added to the sidebar and adding the Team name.

Next, a quick update to the `application` layout to make sure the layout always uses the full window height as a minimum using `h-full` from TailwindCSS:

```
<!DOCTYPE html>
  <html class="h-full">
    <head>
    <title>StandupApp</title>
    <meta name="viewport" content="width=device-width,ini\
tial-scale=1">
    <%= csrf_meta_tags %>
    <%= csp_meta_tag %>
    <%= stylesheet_link_tag "tailwind", "inter-font", "da\
ta-turbo-track": "reload" %>

    <%= stylesheet_link_tag "application", "data-turbo-tr\
ack": "reload" %>

    <link rel="stylesheet" href="https://ga.jspm.io/npm:f\
latpickr@4.6.9/dist/flatpickr.min.css">
    <%= javascript_importmap_tags %>
  </head>

  <body class="bg-gray-800 font-sans leading-normal track\
ing-normal mt-12 h-full">

    <%= render partial: 'layouts/components/header' %>
    <div class="flex flex-col md:flex-row min-h-full">
      <%= render partial: 'layouts/components/sidebar' %>
      <div class="main-content flex-1 bg-gray-100 mt-12 m\
d:mt-2 pb-24 md:pb-5">
        <div class="flex items-center justify-center">
          <%= flash_messages %>
        </div>
          <%= yield %>
        <%= render partial: 'layouts/components/footer' %>
      </div>
    </div>
  </body>
```

```
</html>
```

Next, we will clean up the header to include relevant links and dropdowns and ensure existing links go to their intended destinations.

```
<header>
  <!--Nav-->
  <nav aria-label="menu nav" class="bg-gray-800 pt-2 md:p\
t-1 pb-1 px-1 mt-0 h-auto fixed w-full z-20 top-0">

    <div class="flex flex-wrap items-center">
      <div class="flex flex-shrink md:w-1/2 justify-cente\
r md:justify-start text-white">
        <a href="/" aria-label="Home">
          <span class="text-xl pl-2">Standup App</span>
        </a>
      </div>

      <div class="flex w-full pt-2 content-center justify\
-between md:w-1/2 md:justify-end">
        <ul class="list-reset flex justify-between flex-1\
 md:flex-none items-center">
          <li class="flex-1 md:flex-none md:mr-3">
            <a href="/t" class="inline-block py-2 px-4 te\
xt-white no-underline">Teams</a>
            <a href="/s" class="inline-block py-2 px-4 te\
xt-white no-underline">Standups</a>
          </li>
          <li class="flex-1 md:flex-none md:mr-3">
            <div data-controller="dropdown" class="relati\
ve inline-block">
              <button data-action="dropdown#toggle click@\
window->dropdown#hide" class="drop-button text-white py-2\
 px-2">
                <%= heroicon 'bell' %></button>
```

```
              <div data-dropdown-target="menu" class="w-6\
4 hidden absolute bg-gray-800 text-white right-0 mt-3 p-3\
 overflow-auto z-30" data-transition-enter-from="opacity-\
0 scale-95" data-transition-enter-to="opacity-100 scale-1\
00" data-transition-leave-from="opacity-100 scale-100" da\
ta-transition-leave-to="opacity-0 scale-95" id="nav-dropd\
own">
                <span class="text-xs text-gray-400">15 No\
tifications</span>
                <div class="border border-gray-800"></div>
                <a href="#" data-action="dropdown#toggle"\
 class="p-2 hover:bg-gray-800 text-white text-sm no-under\
line hover:no-underline block">Some Notification</a>
                <a href="#" data-action="dropdown#toggle"\
 class="p-2 hover:bg-gray-800 text-white text-sm no-under\
line hover:no-underline block">AnotherSome Notification</\
a>
              </div>
            </div>
          </li>
          <li class="flex-1 md:flex-none md:mr-3">
            <div data-controller="dropdown" class="relati\
ve inline-block">
              <button data-action="dropdown#toggle click@\
window->dropdown#hide" class="drop-button text-white py-2\
 px-2">
                <%= heroicon 'cog' %>
              </button>
              <div data-dropdown-target="menu" class="hid\
den absolute bg-gray-800 text-white right-0 mt-3 p-3 over\
flow-auto z-30" data-transition-enter-from="opacity-0 sca\
le-95" data-transition-enter-to="opacity-100 scale-100" d\
ata-transition-leave-from="opacity-100 scale-100" data-tr\
ansition-leave-to="opacity-0 scale-95" id="nav-dropdown">
                <img
```

```
                  class="mx-auto rounded-full"
                  alt="User Image"
                  src='<%="#{user_avatar_url(current_user\
.email, 45 )} "%>'
                 />
                <span class="text-xs">
                  <%= current_user.email %>
                </span>

                <div class="py-4">
                  <div class="w-full border-t border-gray\
-300"></div>
                </div>

                <a href="/user/me" data-action="dropdown#\
toggle" class="p-2 hover:bg-gray-800 text-white text-md n\
o-underline hover:no-underline block"><i class="fa fa-use\
r fa-fw"></i> Profile</a>
                <a href="/account/users" data-action="dro\
pdown#toggle" class="p-2 hover:bg-gray-800 text-white tex\
t-md no-underline hover:no-underline block"><i class="fa \
fa-cog fa-fw"></i> Settings</a>

                <div class="py-2">
                  <div class="w-full border-t border-gray\
-300"></div>
                </div>

                <%= link_to("Log out", destroy_user_sessi\
on_path, :method => :delete, class: 'p-2 hover:bg-gray-80\
0 text-white text-md no-underline hover:no-underline bloc\
k')%>

              </div>
            </div>
```

```
          </li>
        </ul>
      </div>
    </div>

  </nav>
</header>
```

Awesome! Your application is starting to take shape and has a more cohesive navigation strategy. Feel free to continue tweaking this yourself, maybe using different links, styles, or icons. With that said, let's go ahead and commit your work:

```
git add .
git commit -am "Update the markup for navigation"
git push
```

5.1.4 Tasks with Single Table Inheritance

Now that we have a robust grouping data model and a base model for which most things revolve around (`standups`), we can start interacting with auxiliary objects/models. As mentioned throughout this book, a Standup consists of:
What you have done (yesterday).
What you are doing (today).
What is preventing you from moving forward(blockers/roadblocks).

If you think about it, both work that was done, work that is still left to do, work that is blocked, and work that was done previously are all `tasks`. The only thing changing between all those is the label or `type`. This is where it gets interesting in Rails. `type` is an extraordinary thing in an `ActiveRecord` model. This column, coupled with `inheriting` or `extending` models from one another, will result in using Rails' built-in 'Single table inheritance.

Here is what the Task Model should look like:

- Title
- `is_completed`
- Type - Used in single table inheritance
- User(to who it belongs to)

Some optional fields can always be added later and not addressed in this book:

- Description - If you wanted a more extensive interface to explain the particular task
- Comments - If you want others to comment on a task(this would most likely be an association)
- Labels - Being able to label the task(Q1, UX, client work, etc.)
- Due Date - set a due date

Plus, anything more you can think of to enhance your application!

At this point, after talking about `Tasks` so much, you may be wondering how they relate to `Standups`. You're in for another Memberships model, as both sides of the association can have multiple relations. Think of it this way, it is rather apparent a `Standup` can have many `Tasks`. Yet, thinking about the inverse, `Tasks` can also belong to many `Standups`. Imagine you marked a `Todo` that you anticipated would be the day you listed the `Standup`, but it actually did not get completed! Now you would want a new `Membership` that relates that `Task` to a new `Standup`.

Let's create the `Task` model and migration:

```
rails g model Task type title is_completed:boolean
```

The `TaskMembership` model and migration:

```
rails g model TaskMembership task:references standup:refe\
rences
rails db:migrate
```

This will get you set up with the models needed, database migrated, factories, and stubbed-out spec files. First, let's dig into the model files to ensure the associations are set up the way we need them. Also, create two more model files manually to get going with the Single Table Inheritance.

```
class Task < ApplicationRecord

  validates_presence_of :title
  validates_presence_of :type

  has_many :task_memberships
  has_many :standups, through: :task_memberships
end
```

We're manually adding the validations and association to the `TaskMembership` to get a `has_many` on `Standup`.

Now let's look at the `TaskMembership` model file:

```
class TaskMembership < ApplicationRecord
  belongs_to :task
  belongs_to :standup
end
```

The `Did` model file:

```
class Did < Task

end
```

The `Todo` model file:

```
class Todo < Task

end
```

And the `Blocker` model file:

```
class Blocker < Task

end
```

Lastly, let's update the `Standup` model, which will now deal with being the inverse of some `Task` associations:

```
class Standup < ApplicationRecord
  belongs_to :user

  has_many :task_memberships, dependent: :delete_all
  has_many :tasks, through: :task_memberships
  has_many :dids,
           -> { where(type: 'Did') },
           through: :task_memberships,
           source: :task
  has_many :todos,
           -> { where(type: 'Todo') },
           through: :task_memberships,
           source: :task
  has_many :blockers,
           -> { where(type: 'Blocker') },
```

```
            through: :task_memberships,
            source: :task
end
```

The model is straightforward when connecting to the `Task` and sub-models. However, there is a new piece of syntax introduced `->{}`. This is best explained straight from the API docs:

> You can pass a second argument scope as a callable (i.e., proc or lambda) to retrieve a specific set of records or customize the generated query when accessing the associated collection.

You can now call something like `Standup.first.todos` and get a list of `Todo` objects, joined through the `TaskMembership` table.

With all those models set up and ready to go, let's write some model specs. The specs will make sure we are asserting the data is functioning as expected with all associations and validations:

```
require 'rails_helper'

RSpec.describe Task, type: :model do
  it "has a valid factory" do
    expect(FactoryBot.build(:task)).to be_valid
  end

  it { is_expected.to validate_presence_of(:title) }
  it { is_expected.to validate_presence_of(:type) }

  it { is_expected.to have_many(:task_memberships) }
  it { is_expected.to have_many(:standups).through(:task_\
memberships) }
end
```

You may notice one spec fails relating to the type not being present, this will require a small change to the factory that was generated by Rails:

From:

```
type { "" }
```

to:

```
type { "Todo" }
```

Continuing with the other models, we'll be just validating they build, all associations work, and all validations work. So let's go through the remaining four:

title="spec/models/task_membership_spec.rb"}

```
require 'rails_helper'

RSpec.describe TaskMembership, type: :model do
  it { is_expected.to belong_to(:task) }
  it { is_expected.to belong_to(:standup) }
end


require 'rails_helper'

RSpec.describe Did, type: :model do
  it "has a valid factory" do
    expect(FactoryBot.build(:task, type: "Did")).to be_va\
lid
  end
end
```

```
require 'rails_helper'

RSpec.describe Todo, type: :model do
  it "has a valid factory" do
    expect(FactoryBot.build(:task, type: "Todo")).to be_v\
alid
  end
end


require 'rails_helper'

RSpec.describe Blocker, type: :model do
  it "has a valid factory" do
    expect(FactoryBot.build(:task, type: "Blocker")).to b\
e_valid
  end
end
```

Lastly, again as we updated the existing `Standup` model, we will update the `Standup` model spec:

```
require 'rails_helper'

RSpec.describe Standup, type: :model do
  it "has a valid factory" do
    expect(FactoryBot.build(:standup)).to be_valid
  end

  it { is_expected.to belong_to(:user) }

  it { is_expected.to have_many(:task_memberships) }
  it { is_expected.to have_many(:tasks).through(:task_mem\
berships) }
```

```
  it { is_expected.to have_many(:dids).through(:task_memb\
erships) }
  it { is_expected.to have_many(:todos).through(:task_mem\
berships) }
  it { is_expected.to have_many(:blockers).through(:task_\
memberships) }
end
```

Once more, we're met with a failing spec that requires an update to the factory. We want the factory to use an association instead of the default `nil`. Make the change in `spec/factories/standups.rb` from:

```
user { nil }
```

to:

```
user
```

This will allow FactoryBot to use the user factory as an association.

5.1.4.1 Standup and Task User Interfaces

Now that we have much of the application's data layer laid out in the database and similar models(and join table-based models), we can lay out more interfaces. Here are some interfaces we will be adding or implementing:

- Create a Standup
- Add/Remove/Complete Todo/Blocker
- View Standups based on context(date/team)

The first markup change we will make is to a `Submit Standup` button in the header that will serve as an entry point into creating a new Standup. Hold on tight because this change will open Pandora's box of changes that will need to be made to continue to support the interfaces needed:

```
<header>
  <!--Nav-->
  <nav aria-label="menu nav" class="bg-gray-800 pt-2 md:p\
t-1 pb-1 px-1 mt-0 h-auto fixed w-full z-20 top-0">

    <div class="flex flex-wrap items-center">
      <div class="flex flex-shrink md:w-1/3 justify-cente\
r md:justify-start text-white">
        <a href="/" aria-label="Home">
          <span class="text-xl pl-2">Standup App</span>
        </a>
      </div>

      <div class="flex w-full content-center justify-betw\
een md:w-1/3 md:justify-center">
        <ul class="list-reset flex justify-between flex-1\
 md:flex-none items-center">
          <li class="flex-1 md:flex-none md:mr-3">
            <a href='/s/new/<%="#{Date.today.iso8601}"%>'\
 class="font-medium rounded-lg text-sm px-5 py-2.5 text-c\
enter mr-2 mb-2 border border-gray-400 text-gray-400 hove\
r:text-white hover:bg-gray-600 focus:ring-gray-800">
              Submit a Standup
            </a>
          </li>
        </ul>
      </div>

      <div class="flex w-full pt-2 content-center justify\
-between md:w-1/3 md:justify-end">
        <ul class="list-reset flex justify-between flex-1\
 md:flex-none items-center">
          <li class="flex-1 md:flex-none md:mr-3">
            <a href="/t" class="inline-block py-2 px-4 te\
xt-white no-underline">Teams</a>
```

```
            <a href="/s" class="inline-block py-2 px-4 te\
xt-white no-underline">Standups</a>
          </li>
          <li class="flex-1 md:flex-none md:mr-3">
            <div data-controller="dropdown" class="relati\
ve inline-block">
              <button data-action="dropdown#toggle click@\
window->dropdown#hide" class="drop-button text-white py-2\
 px-2">
                <%= heroicon 'bell' %></button>
              <div data-dropdown-target="menu" class="w-6\
4 hidden absolute bg-gray-800 text-white right-0 mt-3 p-3\
 overflow-auto z-30" data-transition-enter-from="opacity-\
0 scale-95" data-transition-enter-to="opacity-100 scale-1\
00" data-transition-leave-from="opacity-100 scale-100" da\
ta-transition-leave-to="opacity-0 scale-95" id="nav-dropd\
own">
                <span class="text-xs text-gray-400">15 No\
tifications</span>
                <div class="border border-gray-800"></div>
                <a href="#" data-action="dropdown#toggle"\
 class="p-2 hover:bg-gray-800 text-white text-sm no-under\
line hover:no-underline block">Some Notification</a>
                <a href="#" data-action="dropdown#toggle"\
 class="p-2 hover:bg-gray-800 text-white text-sm no-under\
line hover:no-underline block">AnotherSome Notification</\
a>
              </div>
            </div>
          </li>
          <li class="flex-1 md:flex-none md:mr-3">
            <div data-controller="dropdown" class="relati\
ve inline-block">
              <button data-action="dropdown#toggle click@\
window->dropdown#hide" class="drop-button text-white py-2\
```

```
 px-2">
                <%= heroicon 'cog' %>
              </button>
              <div data-dropdown-target="menu" class="hid\
den absolute bg-gray-800 text-white right-0 mt-3 p-3 over\
flow-auto z-30" data-transition-enter-from="opacity-0 sca\
le-95" data-transition-enter-to="opacity-100 scale-100" d\
ata-transition-leave-from="opacity-100 scale-100" data-tr\
ansition-leave-to="opacity-0 scale-95" id="nav-dropdown">
                <img
                  class="mx-auto rounded-full"
                  alt="User Image"
                  src='<%="#{user_avatar_url(current_user\
.email, 45 )} "%>'
                 />
                <span class="text-xs">
                  <%= current_user.email %>
                </span>

                <div class="py-4">
                  <div class="w-full border-t border-gray\
-300"></div>
                </div>

                <a href="/user/me" data-action="dropdown#\
toggle" class="p-2 hover:bg-gray-800 text-white text-md n\
o-underline hover:no-underline block"><i class="fa fa-use\
r fa-fw"></i> Profile</a>
                <a href="/account/users" data-action="dro\
pdown#toggle" class="p-2 hover:bg-gray-800 text-white tex\
t-md no-underline hover:no-underline block"><i class="fa \
fa-cog fa-fw"></i> Settings</a>

                <div class="py-2">
                  <div class="w-full border-t border-gray\
```

```
-300"></div>
                </div>

                <%= link_to("Log out", destroy_user_sessi\
on_path, :method => :delete, class: 'p-2 hover:bg-gray-80\
0 text-white text-md no-underline hover:no-underline bloc\
k')%>

              </div>
            </div>
          </li>
        </ul>
      </div>
    </div>

  </nav>
</header>
```

The important addition here is (but do use the whole file as a few other classes were modified):

```
<div class="flex w-full content-center justify-between md\
:w-1/3 md:justify-center">
  <ul class="list-reset flex justify-between flex-1 md:fl\
ex-none items-center">
    <li class="flex-1 md:flex-none md:mr-3">
      <a href='/s/new/<%="#{Date.today.iso8601}"%>' class\
="font-medium rounded-lg text-sm px-5 py-2.5 text-center \
mr-2 mb-2 border border-gray-400 text-gray-400 hover:text\
-white hover:bg-gray-600 focus:ring-gray-800">
        Submit a Standup
      </a>
    </li>
  </ul>
</div>
```

This is just a simple entry point into a new `Standup`, using today's date as the default date to apply for the new `Standup`.

To make sure the link goes to a valid location, we will need to make a few more edits and changes. First, we will update the `config/routes.rb` file to allow the date to be part of an optional portion of the path and, when present, gets used as a parameter in the controller. Next, we will update the controller to work as needed and finally adjust some of the default markup created by the Rails generator.

First up, the `routes.rb` file. Change:

```
resources :standups
```

to

```
get 's/new/(:date)', to: 'standups#new', as: 'new_standup'
get 's/edit/(:date)', to: 'standups#edit', as: 'edit_stan\
dup'
resources :standups, path: 's', except: [:new, :edit]
```

```
class StandupsController < ApplicationController
  def index
    redirect_to(root_path)
  end

  def new
    return if check_for_blank_date
    return if check_for_existance
    @standup = Standup.new
  end

  def edit
    return if check_for_blank_date
```

```
    return if check_for_existance
    @standup = Standup.find_by(
      user_id: current_user.id,
      standup_date: current_date
    )
  end

  # POST /standups
  # POST /standups.json
  def create
    @standup = Standup.new(standup_params)
    @standup.user = current_user

    if @standup.save
      redirect_back(
        fallback_location: root_path,
        notice: 'Standup was successfully created.'
      )
    else
      render :new
    end
  end

  def update
    if @standup.update(standup_params)
      redirect_back(
        fallback_location: root_path,
        notice: 'Standup was successfully updated.'
      )
    end
  end

  private

  # Use callbacks to share common setup or constraints be\
```

```
tween actions.
  def set_standup
    @standup = Standup.find(params[:id])
  end

  # Never trust parameters from the scary internet, only \
allow the white list
  # through.
  def standup_params
    params.require(:standup).permit(:standup_date, dids_a\
ttributes:\
      [:id, :title, :_destroy], todos_attributes: [:id, :\
title, :_destroy],\
      blockers_attributes: [:id, :title, :_destroy])
  end

  def check_for_blank_date
    if params[:date].blank?
      redirect_to(
        update_date_path(
          date: Date.today.iso8601,
          reload_path: "/s/#{action_name}/#{Date.today.is\
o8601}"
        )
      ) and return true
    end
  end

  def check_for_existance
    standup = Standup.find_by(
      user_id: current_user.id, standup_date: current_date
    )
    if standup.present? && action_name == 'new'
      redirect_to(edit_standup_path(date: current_date)) \
and return true
```

```
    elsif standup.nil? && action_name == 'edit'
      redirect_to(new_standup_path(date: current_date)) a\
nd return true
    end
  end
end
```

Let's go over what we got in here so far. First, you will notice the `new` method, which will get called from the `Submit a Standup` button. That method calls out two methods first, `check_for_blank_date`, which will check if the `params[:date]` field is blank. If the parameter is empty, the method will call `redirect_to` to the Date Change controller, then redirect back.

The following method does something similar. This time, the method checks for some conditions on the existence and redirects if the current `action_name` does not match a particular `Standup` state.

You may notice the `redirect_to` lines also return true. This is because controller actions will try their best to run the entire action but get mad if you redirect or render it multiple times. Thus, we return `true` out of the private method to force a return in the action. Which, in turn, stops the action and exits.

`standup_params` and `set_standup` are default private methods added to the generated controller.

The create action may be the most difficult to reason about in this chapter. Create a new, unsaved `Standup` object, and add the current user. Once saved, the application will redirect back to the last page.

Let's start with a simple `new` page, an `edit` page and `_form` to get some markup going.

```
<div class="flex p-5" aria-label="Breadcrumb">
  <ol class="inline-flex items-center space-x-1 md:space-\
x-3">
  </ol>
</div>
<div class="flex justify-center">
  <div class="p-4 w-3/5 bg-white rounded-lg border border\
-gray-200 shadow-md sm:p-6 lg:p-8">
    <h5 class="text-xl font-bold leading-none text-gray-9\
00">Add a standup:</h5>
     <%= render 'form' %>
  </div>
</div>
```

```
<div class="flex p-5" aria-label="Breadcrumb">
  <ol class="inline-flex items-center space-x-1 md:space-\
x-3">
  </ol>
</div>
<div class="flex justify-center">
  <div class="p-4 w-3/5 bg-white rounded-lg border border\
-gray-200 shadow-md sm:p-6 lg:p-8">
    <h5 class="text-xl font-bold leading-none text-gray-9\
00">Edit the standup:</h5>
     <%= render 'form' %>
  </div>
</div>
```

Pretty simple stuff. Set a header tag and call out to the partial to be rendered. Now, here is the markup:

```
<%= form_with model: @standup, id: 'team-form', html: {cl\
ass: 'space-y-6'}  do |f| %>
  <%= f.hidden_field :standup_date, value: current_date %>
  <%= render partial: 'layouts/components/date_change', l\
ocals: {reload_path: "/s/#{action_name}"} %>
  <% if @standup.errors.any? %>
    <div class="text-red-700">
      <h4>
        <%= "#{pluralize(@standup.errors.count, "error")}\
 prohibited this team from being saved:" %>
      </h4>
      <%== @standup.errors.full_messages.map{|msg|content\
_tag(:li, msg)}.join %>
    </div>
  <% end %>
  <h3>
    What did you do since your last standup?
  </h3>
  <div data-controller='nested-form' data-nested-form-wra\
pper-selector-value='.nested-did'>
    <template data-nested-form-target="template">
      <%= f.fields_for :dids, Did.new, child_index: 'NEW_\
RECORD' do |did| %>
        <%= render "task_field", f: did, placeholder: 'A \
thing I did' %>
      <% end %>
    </template>
    <%= f.fields_for :dids do |did| %>
      <%= render 'task_field', f: did, placeholder: 'A th\
ing I did' %>
    <% end %>
    <div data-nested-form-target="target"></div>

    <button type="button" class="text-green-700 hover:tex\
t-white border border-green-700 hover:bg-green-800 focus:\
```

```
ring-4 focus:outline-none focus:ring-green-300 font-mediu\
m rounded-lg text-sm px-5 py-2.5 text-center my-3" data-a\
ction="nested-form#add">
      <%= heroicon 'plus' %>
    </button>
  </div>

  <h3>
    What will I do today?
  </h3>
  <div data-controller='nested-form' data-nested-form-wra\
pper-selector-value='.nested-todo'>
    <template data-nested-form-target="template">
      <%= f.fields_for :todos, Todo.new, child_index: 'NE\
W_RECORD' do |todo| %>
        <%= render "task_field", f: todo, placeholder: 'A\
 thing I have to do' %>
      <% end %>
    </template>
    <%= f.fields_for :todos do |todo| %>
      <%= render 'task_field', f: todo, placeholder: 'A t\
hing I have to do' %>
    <% end %>
    <div data-nested-form-target="target"></div>

    <button type="button" class="text-green-700 hover:tex\
t-white border border-green-700 hover:bg-green-800 focus:\
ring-4 focus:outline-none focus:ring-green-300 font-mediu\
m rounded-lg text-sm px-5 py-2.5 text-center my-3" data-a\
ction="nested-form#add">
      <%= heroicon 'plus' %>
    </button>
  </div>

  <h3>
```

```
    What blocked you since your last standup?
  </h3>
  <div data-controller='nested-form' data-nested-form-wra\
pper-selector-value='.nested-blocker'>
    <template data-nested-form-target="template">
      <%= f.fields_for :blockers, Blocker.new, child_inde\
x: 'NEW_RECORD' do |blocker| %>
        <%= render "task_field", f: blocker, placeholder:\
 'A thing I was blocked me' %>
      <% end %>
    </template>
    <%= f.fields_for :blockers do |blocker| %>
      <%= render 'task_field', f: blocker, placeholder: '\
A thing that blocked me' %>
    <% end %>
    <div data-nested-form-target="target"></div>

    <button type="button" class="text-green-700 hover:tex\
t-white border border-green-700 hover:bg-green-800 focus:\
ring-4 focus:outline-none focus:ring-green-300 font-mediu\
m rounded-lg text-sm px-5 py-2.5 text-center my-3" data-a\
ction="nested-form#add">
      <%= heroicon 'plus' %>
    </button>
  </div>
  <%= f.submit "Save", class: "text-white bg-blue-700 hov\
er:bg-blue-800 focus:ring-4 focus:outline-none focus:ring\
-blue-300 font-medium rounded-lg text-sm w-full sm:w-auto\
 px-5 py-2.5 text-center" %>
<% end %>
```

This form is actually pretty straightforward except for one thing. This form utilizes a StimulusJS based package called `stimulus-rails-nested-form` . We'll get that set up and installed in just a second. The form is essentially a nested form with each of

the three `Task` types. Each of those nested sections will allows `Task` types to be submitted for the standup.

As you can see, there is a partial used for each field(and new fields). One partial for each nested attribute type.

```
<div class="flex nested-<%= f.object.class.name.downcase \
%>" data-new-record="<%= f.object.new_record? %>">
  <%= f.text_field :title, class: 'inline bg-gray-50 bord\
er border-gray-300 text-gray-900 text-sm rounded-lg focus\
:ring-blue-500 focus:border-blue-500 block w-full p-2.5',\
 placeholder: placeholder %>
  <span class="input-group-append">
    <button type="button" class="ml-2 align-middle inline\
 text-red-700 hover:text-white border border-red-700 hove\
r:bg-red-800 focus:ring-4 focus:outline-none focus:ring-r\
ed-300 font-medium rounded-lg text-sm px-5 py-2.5 text-ce\
nter mr-2" data-action="nested-form#remove">
      <%= heroicon 'trash', class: 'inline'%>
    </button>

    <%= f.hidden_field :_destroy %>
  </span>
</div>
```

To get `stimulus-rails-nested-form` in your project, you will need to do two things:

1. Pin the javascript through ImportMaps ‘ bin/importmap pin stimulus-rails-nested-form‘.
2. Add the following lines to your Application Controller (`app/javascript/controllers/application.js`):

```
import NestedForm from 'stimulus-rails-nested-form'

application.register('nested-form', NestedForm)
```

Restart your Rails server to be sure the changes are picked up.

Next, we will need to modify the `Standup` model to allow the form submitted attributes to be saved as nested data.

```
class Standup < ApplicationRecord
...

  has_many :dids,
           -> { where(type: 'Did') },
           through: :task_memberships,
           source: :task
  accepts_nested_attributes_for :dids,
                                reject_if: :all_blank,
                                allow_destroy: true

  has_many :todos,
           -> { where(type: 'Todo') },
           through: :task_memberships,
           source: :task
  accepts_nested_attributes_for :todos,
                                reject_if: :all_blank,
                                allow_destroy: true

  has_many :blockers,
           -> { where(type: 'Blocker') },
           through: :task_memberships,
           source: :task
  accepts_nested_attributes_for :blockers,
                                reject_if: :all_blank,
                                allow_destroy: true
end
```

The `accepts_nested_attributes_for` syntax will allow you to tell the model to enable this nested saving. Furthermore, it will skip any nested records sent with `reject_if: :all_blank`. Lastly, this syntax allows the deletion of nested attributes with a `_destroy` key with `allow_destroy: true`. The great thing here is that most of the intricacies are hidden through the cocoon gem. Thus, only needs you to set up some simple forms.

With all that said, after clicking `Submit a Standup`, you should see a form that will allow you to add the three task types to a Standup for a particular day.

5.1.4.2 Viewing Standups with Contextual Filtering

You will want to view these standups that you are adding. A few different contexts will be used in filtering as well.

- Viewing your own `Standups`, sorted by date
- Viewing `Standups by user` in a user profile, sorted by date
- Viewing `Standups` by Team, one day at a time

Let's throw these changes on a new branch that we will merge into main when done:

```
git checkout -b standup-markup
```

We can start by updating the `My Activity` dashboard to show your own standups sorted by date, with the most recent first. You will want to update your `ActivityController`'s mine method to achieve this. This change will include the `standups` and changing the template to loop over this as well.

Let's start with the controller:

```
...

  def mine
    @standups =
      current_user
      .standups
      .includes(:dids, :todos, :blockers)
      .references(:tasks)
      .order('standup_date DESC')
  end

...
```

Here we collect the `current_user` standups for iteration inside the template. Though, you may notice some new methods being chained in here. First, you may notice `.includes(:dids, :todos, :blockers)`. When loading a collection of records and then intending to fetch collections of associated records, you will want to `preload` those association records. This way, the template will not be making a SQL request every iteration of the main collection.

** TIP: This is one of the quickest and easiest ways to get performance gains within a Rails application. Use this anytime you can. **

Next, you will notice `.references(:tasks)`. This method is a little more obscure and less used. Due to the `has_many through` approach taken with the model association, this line allows the `includes` method to work.

Lastly, `.order('standup_date DESC')` instructs the query to sort the collection of `standups` by the `standup_date`, descending(or most recent first).

Even though the controller is ready, there is still one relationship not hooked up and would cause the page to error if the controller action is used. Currently, there is no `.standups` relation on the `User`

model. However, this is a quick fix; you can simply add `has_many :standups` to the `User` model.

Now we are ready for the updated template file: Here, we collect the `current_user` standups for iteration inside the template. Though, you may notice some new methods being chained in here. First, you may notice `.includes(:dids, :todos, :blockers)`. When loading a collection of records and then intending to fetch collections of associated records, you will want to `preload` those association records. This way, the template will not be making a SQL request every iteration of the main collection.

** TIP: This is one of the quickest and easiest ways to get performance gains within a Rails application. Use this anytime you can. **

Next, you will notice `.references(:tasks)`. This method is a little more obscure and less used. Due to the `has_many through` approach taken with the model association, this line allows the `includes` method to work.

Lastly, `.order('standup_date DESC')` instructs the query to sort the collection of `standups` by the `standup_date`, descending(or most recent first).

Even though the controller is ready, there is still one relationship not hooked up and would cause the page to error if the controller action is used. Currently, there is no `.standups` relation on the `User` model. However, this is a quick fix; you can simply add `has_many :standups` to the `User` model.

```
<div class="flex p-5" aria-label="Breadcrumb">
  <ol class="inline-flex items-center space-x-1 md:space-\
x-3">
    <li class="inline-flex items-center">
      <h2 href="#" class="inline-flex items-center font-m\
edium text-gray-700 hover:text-gray-900">
        My Activity
      </h2>
    </li>
  </ol>
</div>

<div class="flex justify-center">
  <div class="p-4">
    <div class="grid grid-cols-3 auto-cols-auto gap-4">
      <%= render @standups %>
    </div>
  </div>
</div>
```

This is now a pretty standard type of template file for us. First, split a collection into sets of three for the row/grid templating, then iterate over the items. In the particular case of this template, you are also iterating over the `Dids`, `Todos`, and `Blockers` in unordered lists.

Next, we will update the partial that the loop of standups uses to match the rest of the application:

```
<div id="<%= dom_id standup %>" class="w-96 block p-6 bg-\
white rounded-lg border border-gray-200 shadow-md">
  <div>
    <div class="float-right">
      <%= link_to edit_standup_path(standup) do %>
        <%= heroicon "pencil-alt", class: 'inline' %>
      <% end %>
    </div>
    <h3 class="font-bold leading-tight text-gray-900 text\
-xl"><%= standup.standup_date %></h3>
  </div>
  <h4 class="mt-4 font-bold leading-tight text-gray-900 t\
ext-lg">Dids</h4>
  <ul>
    <% standup.dids.each do |did| %>
      <li class="bg-gray-300 border border-gray-100 p-2 r\
ounded shadow-sm"> <%= did.title %> </li>
    <% end %>
  </ul>

  <h4 class="mt-4 font-bold leading-tight text-gray-900 t\
ext-lg">Todos</h4>
  <ul>
    <% standup.todos.each do |todo| %>
      <li class="bg-gray-300 border border-gray-100 p-2 r\
ounded shadow-sm"> <%= todo.title %> </li>
    <% end %>
  </ul>

  <h4 class="mt-4 font-bold leading-tight text-gray-900 t\
ext-lg">Blockers</h4>
  <ul>
    <% standup.blockers.each do |blocker| %>
      <li class="bg-gray-300 border border-gray-100 p-2 r\
ounded shadow-sm"> <%= blocker.title %> </li>
```

```
    <% end %>
  </ul>
</div>
```

The following few sections will be more involved but will not be terribly difficult. First, we will spruce up the team page to show the current day's standups. While providing another page will function as a day-by-day view of standups, per that team.

The first place we will want to start is in the routes file. This way, we can make sure all the paths are available and working.

```
Rails.application.routes.draw do
...

  get 't/new', to: 'teams#new'
  get 't/:id/edit', to: 'teams#edit'
  get 't/:id/s', to: 'teams/standups#index', as: 'team_st\
andups'
  get 't/:id/s/(:date)', to: 'teams/standups#index', as: \
'team_standups_by_date'
  get 't/:id/(:date)', to: 'teams#show'
  resources :teams, path: 't'

...

  scope 'account', as: 'account' do
    resources :users do
      member do
        get 's', to: 'users/standups#index', as: 'standup\
s'
      end
    end
  end

...
```

```
end
```

Here, I am pointing out two sections of the `config/routes.rb` file. First, there are a few manual routes listed for teams, which I'll explain. Typically, you can add multi-level routes right through the Rails Routes DSL. However, using option path parameters, like we are for `(:date)`, makes it a lot tougher to do so. As Rails uses routes in order, by specifying a few existing routes, routes without the optional path, and finally with the optional path, all of the `team`-based paths.

The second section uses the more traditional way of adding multi-level routes to existing resources. Resources can take a block(`do...end`) as their last argument to provide this nesting. There are a few different ways to add these sub-level routes:

1. `resources` - You can use yet another level of resources that will provide a complete CRUD set of routes to your second level. This could be useful in situations where you have a top-level `teams` resource and then sub-level that with different `projects`.
2. `collection` - This is when you need a sub-level route that does not require a specific resource. It will be in the format `collection/route`. This is useful when using a route to provide a scoped view of some sort of resources.
3. `member` - You can use this block when you want your sub-route to be based on a specific resource. Like above, it would be `/user/:id/standups`, which will automatically include a `params[:id]` to the controller's action parameters.

As you can see, we are using a block to sub-route into the `users` resource block. Then, using `member do` to base the resource on specific users. In this case the full route will be `/account/users/:id/s`. You might notice another new thing is the `users/standups#index` in

the `to` for the route. The slash signifies a nested controller would be used. In this case, it will be a `Users::StandupsController` controller, allowing us to use an `index` action and will allow us to stay in a more RESTful resource pattern.

Since this route will actually be called from the standup listing on the team pages, let's go ahead and add the User contextual standups now:

```
class Users::StandupsController < ApplicationController

  def index
    @standups = user
      .standups
      .includes(:dids, :todos, :blockers)
      .references(:tasks)
      .order('standup_date DESC')
  end

  private

  def user
    @user ||= User.find(params[:id])
  end

end
```

Here we are loading the user record as the template, and we will include some user-based data such as `name`, `email`, and timezone. Since the primary focus will be standups, we will also set up the `standups` and their `task`-based associations to be eagerly loaded. Thus, preventing the `N+1` queries from slowing down the application.

Now you can add the template file at `app/views/users/standups/index.html.erb`:

```
<div class="flex p-5" aria-label="Breadcrumb">
  <ol class="inline-flex items-center space-x-1 md:space-\
x-3">
    <li class="inline-flex items-center">
      <h2 href="#" class="inline-flex items-center font-m\
edium text-gray-700 hover:text-gray-900">
        Users
      </h2>
    </li>
    <li class="inline-flex items-center">
      <h2 href="#" class="inline-flex items-center font-m\
edium text-gray-700 hover:text-gray-900">
        <svg class="w-6 h-6 text-gray-400" fill="currentC\
olor" viewBox="0 0 20 20" xmlns="http://www.w3.org/2000/s\
vg"><path fill-rule="evenodd" d="M7.293 14.707a1 1 0 010-\
1.414L10.586 10 7.293 6.707a1 1 0 011.414-1.414l4 4a1 1 0\
 010 1.414l-4 4a1 1 0 01-1.414 0z" clip-rule="evenodd"></\
path></svg>
        <%= @user.name %>
      </h2>
    </li>
  </ol>
</div>

<div class="flex justify-center">
  <div class="p-4 w-4/5 bg-white rounded-lg border border\
-gray-200 shadow-md sm:p-6 lg:p-8">
    <h5 class="text-xl font-bold leading-none text-gray-9\
00">Information:</h5>
      <p>
        <strong> Name</strong>
        <%= ": #{@user.name}" %>
     </p>
     <p>
      <strong> Email</strong>
```

```
      <%= ": #{@user.email}" %>
    </p>
    <p>
      <strong> Timezone</strong>
      <%= ": #{@user.time_zone || 'not set'}" %>
    </p>
  </div>
</div>

<div class="flex justify-center">
  <div class="p-4 w-4/5 sm:p-6 lg:p-8">
    <div class="grid grid-cols-3 auto-cols-auto gap-4">
      <% if @standups.empty? %>
        <div class="row">
          <div class="col">
            <p>No standups are available to display</p>
          </div>
        </div>
      <% end %>
      <%= render partial: 'standups/standup', collection:\
 @standups %>
    </div>
  </div>
</div>
```

This is a pretty standard view template. Near the top, we display some relevant information about the user, including their time zone. Followed by the user's standups sorted by most recent.

With that added, we have two of the three different contexts needed. Since the routes were already added, let's go ahead and jump straight to the changes to the `TeamsController`:

```
class TeamsController < ApplicationController
  load_and_authorize_resource except: [:create]
  around_action :use_time_zone, only: [:edit]

  ...

  def show
    set_teams_and_standups(Date.today.iso8601)
  end

  ...


class Teams::StandupsController < ApplicationController

  def index
    set_teams_and_standups(current_date)
  end

end
```

Since both actions(this and one in the new `Teams::StandupsController`) will require the `team` to be loaded and then a list of each user in the team's standups, we can move this to a method in the `ApplicationController`, and the action will call it with its own date argument.

```
  def set_teams_and_standups(date)
    @team = Team.includes(:users).find(params[:id])
    @standups = @team.users.flat_map do |u|
      u.standups.where(standup_date: date)
      .includes(:dids, :todos, :blockers)
      .references(:tasks)
    end
  end
```

In the case of the `show` action, we will just want to pass on today's date. However, in the case of the `standups#index` action, the `current_date` method from the `ApplicationController` will be triggered. Which will first try and check for a `params[:date]`, such as when a URL with an optional data URL path is entered into the browser. Otherwise, it will check for a date in a session or finally use the current date.

With those actions all set up, we can now follow up with the two templates we will use. Both will use an identical iteration over the collection of standups; we will move that template out into its own partial teamplate that we can call right in the action-based templates.

First, here is the `show`'s template, which is actually an updated version from earlier in the chapter as it no longer will contain a date picker:

```
<div class="flex p-5" aria-label="Breadcrumb">
  <ol class="inline-flex items-center space-x-1 md:space-\
x-3">
    <li class="inline-flex items-center">
      <h2 href="#" class="inline-flex items-center font-m\
edium text-gray-700 hover:text-gray-900">
        Teams
      </h2>
    </li>
```

```
    <li class="inline-flex items-center">
      <h2 href="#" class="inline-flex items-center font-m\
edium text-gray-700 hover:text-gray-900">
        <svg class="w-6 h-6 text-gray-400" fill="currentC\
olor" viewBox="0 0 20 20" xmlns="http://www.w3.org/2000/s\
vg"><path fill-rule="evenodd" d="M7.293 14.707a1 1 0 010-\
1.414L10.586 10 7.293 6.707a1 1 0 011.414-1.414l4 4a1 1 0\
 010 1.414l-4 4a1 1 0 01-1.414 0z" clip-rule="evenodd"></\
path></svg>
        <% @team.name %>
      </h2>
    </li>
  </ol>
</div>

<div class="flex justify-center">
  <div class="p-4 w-4/5 bg-white rounded-lg border border\
-gray-200 shadow-md sm:p-6 lg:p-8">
    <div class="float-right">
      <% if current_user.has_role? :admin, current_accoun\
t %>
        <%= link_to edit_team_path(@team), class: 'inline\
' do %>
          <%= heroicon "pencil-alt", class: 'inline' %>
        <% end %>
        <%= button_to team_path(@team), {form_class: 'inl\
ine', method: :delete, form: {data: { turbo_confirm: "Are\
 you sure? Deleting this team is permanent"}}} do %>
          <%= heroicon "trash", class: 'inline' %>
        <% end %>
      <% end %>
    </div>
    <h3 id="team-settings"><%=@team.name %> Team Settings\
</h3>
    <p>
```

```
      <span>Users:</span>
      <%= @team.users.map(&:email).join(", ") %>
      <% if @team.has_reminder %>
        <span>| Reminder at: </span>
        <%= "#{@team.reminder_time.in_time_zone(current_u\
ser.time_zone)\
          .strftime("%I:%M %p %Z")}\
          (#{@team.reminder_time.in_time_zone(current_use\
r.time_zone)\
          .strftime("%H:%M")})" %>
      <% end %>
      <% if @team.has_recap %>
        <span>| Recap at: </span>
        <%= "#{@team.recap_time.in_time_zone(current_user\
.time_zone)\
        .strftime("%I:%M %p %Z")}\
        (#{@team.recap_time.in_time_zone(current_user.tim\
e_zone)\
        .strftime("%H:%M")}) " %>
      <% end %>
    </p>
  </div>
</div>

<div class="flex justify-center mt-6">
  <div class="px-4 py-1 w-4/5 bg-white rounded-lg border \
border-gray-200 shadow-md">
    <div class="float-right">
      <%= link_to "See More", team_standups_path(@team) %>
    </div>
    <h3 class="card-title"><%= "\"#{@team.name}\"'s Stand\
ups Today" %></h3>
  </div>
</div>
```

```
<div class="flex justify-center">
  <div class="p-4 w-4/5 sm:p-6 lg:p-8">
    <div class="grid grid-cols-3 auto-cols-auto gap-4">
      <% if @standups.empty? %>
        <div class="row">
          <div class="col">
            <p>No standups are available to display</p>
          </div>
        </div>
      <% end %>
      <%= render partial: 'standups/standup', collection:\
 @standups %>
    </div>
  </div>
</div>
```

Now, to use this `standups` partial with multiple different entry points and where the individual standup header will change based on the entry point mentioned above, we will need to create a helper method that the view will use to determine the header output.

```
module StandupHelper
  def standup_header(standup)
    caller = [controller_path, action_name].join("_")
    case caller
    when "activity_mine", "users/standups_index"
      standup.standup_date
    when "teams/standups_index"
      link_to standup.user.name, standups_account_user_pa\
th(standup.user)
    else
      "..."
    end
  end
end
```

Here the helper concatenates the controller path and action name to create a unique string from which view calls the helper. It then uses a case statement to detemine what standup attribute to use to display in the markup. Here is the update markup for the `standup` partial.

```
<div id="<%= dom_id standup %>" class="w-96 block p-6 bg-\
white rounded-lg border border-gray-200 shadow-md">
  <div>
    <div class="float-right">
      <%= link_to edit_standup_path(standup) do %>
        <%= heroicon "pencil-alt", class: 'inline' %>
      <% end %>
    </div>
    <h3 class="font-bold leading-tight text-gray-900 text\
-xl"><%= standup_header standup%></h3>
  </div>
  <h4 class="mt-4 font-bold leading-tight text-gray-900 t\
ext-lg">Dids</h4>
  <ul>
    <% standup.dids.each do |did| %>
      <li class="bg-gray-300 border border-gray-100 p-2 r\
ounded shadow-sm"> <%= did.title %> </li>
    <% end %>
  </ul>

  <h4 class="mt-4 font-bold leading-tight text-gray-900 t\
ext-lg">Todos</h4>
  <ul>
    <% standup.todos.each do |todo| %>
      <li class="bg-gray-300 border border-gray-100 p-2 r\
ounded shadow-sm"> <%= todo.title %> </li>
    <% end %>
  </ul>

  <h4 class="mt-4 font-bold leading-tight text-gray-900 t\
```

```
ext-lg">Blockers</h4>
  <ul>
    <% standup.blockers.each do |blocker| %>
      <li class="bg-gray-300 border border-gray-100 p-2 r\
ounded shadow-sm"> <%= blocker.title %> </li>
    <% end %>
  </ul>
</div>
```

Specifically, that `<%= standup_header standup%>` replaces the current ERB markup to display the date.

We can finally finish this chapter off by writing some more tests to ensure all of the functionality added to this point has been covered.

First, we will update some failing tests that we know need to be changed. In the first version of the `date_change` system specs, we relied on the date-changing functionality on the `activity_mine_path`. However, now we can find that on the `team_standups_path`. Here is the updated spec:

```
require 'rails_helper'

RSpec.describe "DateChanges", type: :system do
  login_admin

  let(:team) { FactoryBot.create(:team, account_id: @admi\
n.account.id) }

  it "changes the date when the back arrow is pressed" do
    visit team_standups_path(team)

    click_on(id: 'date_backwards')

    expect(page).to have_content(
      (Date.today - 1.day)
```

```
      .strftime("%a, %b #{(Date.today - 1.day).day.ordina\
lize}")
    )
  end

  it "changes the date when the forwards arrow is pressed\
" do
    visit team_standups_path(team)

    click_on(id: 'date_forward')

    expect(page).to have_content(
      (Date.today + 1.day)
      .strftime("%a, %b #{(Date.today + 1.day).day.ordina\
lize}")
    )
  end

  it 'should be able to visit standups and move date from\
 picker', js: true do
    visit team_standups_path(team)

    date = (Time.now + 1.day).strftime("%B %-d, %Y")

    find('#datePicker').click

    expect(page).to have_selector(".flatpickr-days")
    find(".flatpickr-calendar .flatpickr-day[aria-label='\
#{date}']").click

    expect(page).to have_content(
      (Time.now + 1.day)
      .strftime(
        "%a, %b #{(Time.now + 1.day).day.ordinalize}"
      )
```

```
    )
  end
end
```

The changes here are pretty simple. First, updating the path helper to the `team_standups_path`, we want to use as well as setting the expectations to expect `team_standups` based paths.

Now, another spec to fix will be the view `activity_mine` spec as it will need a mocked instance variable:

```
require 'rails_helper'

RSpec.describe 'activity/mine.html.erb', type: :view do
  before do
    allow(view)
      .to receive(:current_date)
      .and_return(Date.today.strftime('%a %-d %b %Y'))
    assign(:standups, [FactoryBot.create(:standup)])
  end

  it 'renders the word mine' do
    render template: 'activity/mine.html.erb'
    expect(rendered).to match(/My Activity/)
  end
end
```

Now, the `standups_controller_spec` is pretty close to what we needed by default. But, it requires some changes to the attribute setup and testing the correct action use case. Here is the full new spec file:

```
require 'rails_helper'

RSpec.describe "/standups", type: :request do
  login_admin

  let(:valid_attributes) do
    {
      user_id: @admin.id,
      standup_date: Date.today.iso8601,
      todos_attributes: [
        { title: 'YE Old Thingy', "_destroy": 'false' }
      ]
    }
  end

  let(:invalid_attributes) do
    { user_id: nil, standup_date: Date.today.iso8601, tod\
os_attributes: [] }
  end

  let(:valid_session) { {} }

  describe 'GET #index' do
    it 'assigns a new standup as @standup' do
      get standups_path
      expect(response).to redirect_to(root_path)
    end
  end

  describe 'GET #new' do
    it 'redirects without a date' do
      get new_standup_path, params: {}
      expect(response).to have_http_status(302)
    end
  end
```

```
  describe 'GET #new' do
    it 'assigns a new standup as @standup' do
      get new_standup_path, params: { date: Date.today.is\
o8601 }
      expect(assigns(:standup)).to be_a_new(Standup)
    end
  end

  describe 'GET #new redirect to edit on existance' do
    it 'it redirects' do
      standup = FactoryBot.create(:standup, valid_attribu\
tes)
      get new_standup_path, params: { date: standup.stand\
up_date }
      expect(response).to redirect_to(edit_standup_path(d\
ate: standup.standup_date))
    end
  end

  describe 'GET #edit' do
    it 'assigns the requested standup as @standup' do
      standup = FactoryBot.create(:standup, valid_attribu\
tes)
      get edit_standup_path(standup), params: { date: sta\
ndup.standup_date }
      expect(assigns(:standup)).to eq(standup)
    end
  end

  describe 'GET #edit redirect to new on non-existance' do
    it 'it redirects' do
      date = Date.today.iso8601
      get edit_standup_path(), params: { date: date }
      expect(response).to redirect_to("/s/new/#{date}")
```

```
    end
  end

  describe 'POST #create' do
    context 'with valid params' do
      it 'creates a new Standup' do
        expect do
          post(
            standups_path,
            params: { standup: valid_attributes }
          )
        end.to change(Standup, :count).by(1)
      end

      it 'assigns a newly created standup as @standup' do
        post(
          standups_path,
          params: { standup: valid_attributes }
        )
        expect(assigns(:standup)).to be_a(Standup)
        expect(assigns(:standup)).to be_persisted
      end

      it 'redirects to the created standup' do
        post(
          standups_path,
          params: { standup: valid_attributes }
        )
        expect(response).to redirect_to(root_path)
      end
    end
  end

  describe 'PUT #update' do
    context 'with valid params' do
```

```
      let(:new_attributes) do
        { standup_date: Date.tomorrow.iso8601 }
      end

      it 'updates the requested standup' do
        @standup = FactoryBot.create(:standup, valid_attr\
ibutes)
        put(
          standup_path(@standup),
          params: { id: @standup.to_param, standup: new_a\
ttributes }
        )
        @standup.reload
        expect(@standup.updated_at).to be > @standup.crea\
ted_at
      end

      it 'assigns the requested standup as @standup' do
        standup = FactoryBot.create(:standup, valid_attri\
butes)
        put(
          standup_path(standup),
          params: { id: standup.to_param, standup: valid_\
attributes }
        )
        expect(assigns(:standup)).to eq(standup)
      end

      it 'redirects to the standup' do
        standup = FactoryBot.create(:standup, valid_attri\
butes)
        put(
          standup_path(standup),
          params: { id: standup.to_param, standup: valid_\
attributes }
```

```
        )
        expect(response).to redirect_to(root_path)
      end
    end
  end
end
```

Next, an update on the changes and new action in the new `Teams::StandupsController` based request specs:

```
require 'rails_helper'

RSpec.describe "/teams/standups", type: :request do
  login_admin

  describe "GET #index" do
    it 'assigns all standups as @standups' do
      team = FactoryBot.create(:team, account_id: @admin.\
account.id)
      FactoryBot.create(:team_membership, user_id: @admin\
.id, team_id: team.id)
      FactoryBot.create(:standup, user_id: @admin.id)
      get team_standups_path(team.id)
      expect(assigns(:team)).to eq(team)
      expect(assigns(:standups)).to eq([])
    end
  end
end
```

Finally, an update on the changes and new action in the new `Users::StandupsController`:

```
require 'rails_helper'

RSpec.describe "/users/standups", type: :request do
  login_admin

  describe "GET #index" do
    it 'assigns all standups as @standups' do
      standup = FactoryBot.create(:standup, user_id: @adm\
in.id)
      get standups_account_user_path(@admin.id)
      expect(assigns(:user)).to eq(@admin)
      expect(assigns(:standups)).to eq([standup])
    end
  end
end
```

Additionally, as there are new resources and actions, we will want to update the `app/models/ability.rb` file with new permissions:

```
class Ability
  include CanCan::Ability

  def initialize(user)
    user ||= User.new # guest user (not logged in)
    if user.has_role?(:user, user.account || Account.new)
      can [:me, :password, :update_me, :update_password, \
:standups], User
      can [:feed, :mine], ActivityController
      can [:index, :show, :standups], Team
      cannot :manage, Account
    elsif user.has_role?(:admin, user.account || Account.\
new) #is an admin
      can :manage, :all
    else
      can [:new, :create], Account
```

```
    end
  end
end
```

The `:standup` action and the `Team` line are new. This will allow non-admin users access to needed paths and resources to view standups, team information, etc.

You should be able to run `rspec` at this point and have all tests passing. Additionally, test coverage is still very close to 100%.

```
rspec
.....................................................\
.....................................................\
.............

Finished in 9.89 seconds (files took 4.71 seconds to load)
127 examples, 0 failures

Coverage report generated for RSpec to standup_app/covera\
ge. 310 / 330 LOC (93.94%) covered.
```

Now let's add, commit and merge the changes into the main branch from the `standup-markup` branch you have been working on.

```
git add .
git commit -am "Add Markup for Standups in various contex\
ts"
git checkout main
git merge standup-markup
git push
```

You will now have all of your standup markup code in main, squashed into one commit, and pushed up to your remote main branch.

After all of this in the chapter, your SaaS application is really beginning to shape up. In the next chapter, you will be shown how to keep the features that are added to your app cleanly and elegantly written to keep your app easy to read and maintain.

5.2 What did you learn in this chapter?

- You learned how to use a scaffold to create generic default CRUD files throughout the Rails stack.
- You learned what default files are created within the context of this chapter's `Standup` scaffold
- You learned how to use the session, a controller action, and a controller helper to keep the state of a date for use in the application.
- You learned about `has_many through:` with `TeamMembership`.
- You learned about Single Table Inheritance, using a `Task` base class to hold the `Did`, `Todo`, and `Blocker` types.
- You learned how to create different controllers and templates for similar but different contextual collection information.

5.3 Exercises

Exercise answers can be found with any package or bundle purchases of Build A SaaS App in Ruby on Rails 7[1]!

1. Right now, each `Task` is a static string of text. Most times, when end-users add links to a task, they will want that link turned into an actual clickable anchor tag. Create a method with regular expressions to parse the task field and save a

[1] https://buildasaasappinrails.com

modified string. The user would now have an HTML anchor tag when rendering a modified string with an anchor tag in the `standup` edit form. Write a helper that will search for an anchor, grab the `href` attribute and replace the HTML.

2. The `activity#mine` template could use some extra information. Look up the `current_user`'s teams and list out any teams that have a standup today, with its reminder time.

6. Service Objects



Note:

This chapter may be a little different than the others. It begins with a little bit of theory and simple examples before diving straight into application relevant code, and Service Objects can be a big part of a clean Rails application. Spending a little time understanding them can go a long way. Additionally, there is more digging into the Ruby specifics than in other chapters. This approach helps understand Service Objects and their benefits.

6.1 What is a Service Object?

Service Objects are becoming a staple in the tool belt to slim down Controllers and Models. Welcome to the world of fat services [folder]!

A Service Object is a PORO(Plain Old Ruby Object) to decompose business objects into manageable classes and methods. Without us getting too deep into a discussion on the sound OOP principles and how they can apply to Ruby, a Service Object should have a Single Responsibility, be testable, and a few other aspects in which I go further in-depth.

This idea can be a departure from the Rails design principles of yesteryear. You may have gone straight to adding another method to your *fat* model or, more recently, added logic into a mixin or model concern. While this may help with code separation, it does

not make the activities a Service Object can perform as simple, organized, and discoverable. Additionally, as they are "mixed into" the model class, the model still becomes enlarged and ties everything into the model class.

6.1.1 So...a service Object?

Before going deep on Service Objects, it may be an excellent time to discuss the name of Service Objects. As I have, in fact, seen this come up more and more recently in places such as Reddit, Twitter, etc.

There are a few alternatives here. One of the biggest would be using the name "Operations" taken from the TrailBlazer framework(which sits on the Rails framework).

Another approach I have seen is calling them Objects and removing the services from the class names, folders, etc.

I don't mind the Service Object name and feel it describes that the Class/Object's purpose is to perform an action or service. Though, Operations comes in a close second as it describes Service Objects similarly. Again, though, not breaking some of the current best practices for Service Objects is one good enough reason to stick with "Services".

Here are some great approaches to Service Objects.

Try REALLY hard only to have one public method.

I feel this goes hand in hand with the Single Responsibility Principle for classes. If the class/object is only responsible for one thing, there should only be one public method. Thus, making the object provide that single service operation. The most common method for the one public method constraint is `.call`.

Now, while this is one of the most agreed-upon aspects of Service Object best practices, the method of Service Object instantiation can be formed into two broad groups.

First:

```
#services/thing_service.rb
class ThingService

  def initialize(param1)
    @param1 = param1
  end
  def call
    private_method
  end
  private
  attr_reader :param1
...
end
### and calling by ThingService.new(param1).call
```

or Two:

```
#services/thing_service.rb
class ThingService

  def self.call(param1)
    new(param1).call
  end

  def call
    private_method
  end
  private
  def initialize(param1)
    @param1 = param1
  end

  attr_reader :param1
```

```
...
end
### and calling by ThingService.call(param1)
```

I lean more towards the first way, as it fits my experience and most other Ruby developers' experience creating POROs. However, the second method can be more concise and readable when called.

The main idea here is that you are working to restrict your object to one public method called throughout your application(typically controllers).

An added benefit to one or minimal public methods is that you have an easier time refactoring the service object. You do not need to change much of the Service Object calling throughout your application.

Managing Dependencies can be challenging. Do it privately and optimally.

It is nearly impossible outside the simplest of Service Objects that you won't need to include another dependent Service Object. So what is the best way to add them?

One method that may come to mind is instantiating the objects and including them as parameters. While it can work, this approach would most likely lead to refactoring headaches and changing method signatures in a multitude of places.

Another method would instantiate the dependent service somewhere in the Service Objects public method or primary private method(if you have one). While this method is better than the first, it starts to break down the SRP of a method and makes that method responsible and knowledgeable of the dependency directly.

In my experience, the best approach here would be to create a private method that returns a memoized version of the injected dependency. Here is an example:

```ruby
#services/thing_service.rb
class ThingService

  def initialize(param1)
    @param1 = param1
  end

  def call
    result = other_service.call
    private_method(result)
  end

  private

  attr_reader :param1

  def other_service
    @other_service ||= OtherService.new(param1)
  end
...
end
```

While this isn't a huge change from other dependency injection methods, this can make it easier to change the public method as you refactor as it is no longer directly responsible for another service's instantiation. However, a downside could be that a previously instantiated or memoized object being instantiated again within the service could cause possible duplication of objects or SQL queries. Again, though, it is not enough to make a difference most times.

Return something meaningful. An object.

While your gut instinct would be to return a boolean as a result of a Service Object, it may not be your best option. While yes, the default idiom in Rails is to run a conditional off of an ActiveRecord update/create, a rich response object will pay dividends in the long run.

Think about it this way. There are more than just two outcomes for a service object. 1. Success. 2. Failure. 3. Unhandled Exception (s). It would be a disservice to your service to handle such outcomes just in a boolean response.

When taking these outcomes more in-depth, there is even more a reason for a response object. Let's take the success example. Firstly, a result object that can hold a success? predicate method to handle conditional branching for controller responses and more. Additionally, since it's your response object, you could name the success/failure method as you would like. If `signed_up?` makes more sense, it's as simple as defining it in your object.

Service objects may also have multiple sets of data as a response. For example, suppose you were creating a Credit Card processing service. You may need to record various different tokens from the processor or other data sets that you want to display to the user. But, again, this pattern could be cumbersome without some response object.

Now let's take the other two outcomes of a Service Object: failure and exceptions. At first glance, it seems simple. Just make the results of either result as falsey and force some error/failure browser response. However, that may not deliver full intent or context to the end-user or developer. For example, suppose an exception is triggered due to missing data from a form. That Exception can be rescued, a message set, or passed through a result object and used to correctly inform the end-user in a browser. Now, you have a more meaningful request/response cycle with just a little more information from a response object. More than you would get from just a true or false.

So, how can you use a response object? The most straightforward use case will usually be using a Struct or OpenStruct from the Ruby library. However, that can sometimes leave little to be desired on syntax, and there have been previous concerns on `OpenStruct`

performance. Therefore, I mostly reach for `ImmutableStruct`[1] when creating simple classes to define Result Objects.

```
#services/thing_service.rb
class ThingService

  Result = ImmutableStruct.new(:success?, :things, :error)

  def initialize(param1)
    @param1 = param1
  end

  def call
    private_method
  end

  private

  attr_reader :param1

  def private_method
    things = do_something(param1)
  rescue SomeSpecificException => exception
    Result.new(success: false, things: nil, error: except\
ion.message)
  else
    Result.new(success: true, things: things, error: nil)
  end
end
```

[1] https://github.com/stitchfix/immutable-struct

6.2 Building and using a service object in this app

Rails does not create a services folder by default, so it is our job to create one. A little side-note, though, depending on the size of your app or the complexity of business logic, you may need to break services down even further by domain, type, or other qualifiers. You can just create a services folder with `mkdir app/services` in your app folder.

Restart your rails server to pick up the new folder as Rails will autoload directories in the app directory. Now we can create a file for a user registration service `app/services/new_registration_service.rb` and fill it with our business logic.

Ok! Let's go over the Service Object section by section!

```
class NewRegistrationService

  Result = ImmutableStruct.new(:success?, :things, :error)

  def initialize(params)
    @user = params[:user]
    @account = params[:account]
  end
```

Here we are creating the class and then adding the initialize method. This method makes the needed instance variables upon the object being instantiated. We will pass the Service Object, the User model record object, and the `Account` record object previously created in the `AccountsController`.

Additionally, you will want to add `gem 'immutable-struct'` to your Gemfile and run `bundle install` if it has not already been added (and installed).

```
def process_registration
  account_create
  send_welcome_email
  notify_slack
  Result.new(success: true, user: user, account: account,\
 error: nil)
rescue ActiveRecord::RecordInvalid, Slack::Notifier => ex\
ception
  Result.new(
    success: false, user: user, account: account, error: \
exception.message
  )
end
```

Here we add an instance method to be called on the instantiated object. This object will call the designated methods to run bits of operations. Additionally, if all the methods run uninterrupted, the last line will implicitly return the result object. Otherwise, if we rescue an exception caused by the following methods, it will return a result object that will signify failure.

```
private

attr_reader :user, :account
```

It's important to note the `private` keyword here. This keyword signifies anything below `private` is only allowed to be accessed by the object's instance. This approach reduces the public interface and provides a safer object for the codebase.

Another Ruby language idiom is the use of `attr_` in a few different contexts. Let's take, for example, `attr_reader`. In other languages, you may need to write `getter` functions/methods to get the value of an object's particular attribute. Ruby cuts out some boilerplate by allowing you to use the keyword `attr_reader` followed by a

list of attribute names as symbols in conjunction with setting a corresponding instance variable in the `initialize` method. Ruby also has an `attr_writer` to set attribute values. Lastly, Ruby has `attr_accessor` to handle both getting and setting.

With that out of the way, we can start digging into the logic of this Service.

```
def account_create
  post_account_setup if account.save!
end

def post_account_setup
  user.account_id = account.id
  user.save!
  user.add_role :admin, account
end
```

Here we create the account (and allow the upstream `begin/rescue` block to catch an invalid record). Then, make the necessary updates to the user, such as adding the account_id needed for the belongs_-to relation. Finally, assigning a role for the authorization system scoped to the created account.

```
def send_welcome_email
    WelcomeEmailMailer.welcome_email(user).deliver_later
end
```

This will call a Rails ActionMailer (..and will be covered in Chapter 8). To skip this, for now, you can simply comment out the line:

```
def send_welcome_email
  ## NO-OP
  # WelcomeEmailMailer.welcome_email(user).deliver_later
end
```

Once you create the Mailer in a later chapter, you will simply come back into this Service Object and uncomment that line. Easy Peazy!

Lastly, you can use the latter method here to add a few notifications to yourself when a user signs up. If you use Slack and are familiar with setting up a Webhook (or want to learn, as it's easy enough), make sure you have added `gem 'slack-notifier'` to your Gemfile if you have not already and ran `bundle install`.

"ruby
def notify_slack
notifier = Slack::Notifier.new(

"https://hooks.slack.com/services/some_hashed_id"
)
notifier.ping(

"A New User has appeared! # - #\
|| ENV: #")
end

```
Now, you may be saying to yourself, "this makes sense", b\
ut how and where do you call this Service Object? The fol\
lowing is the code that can be used to replace the `creat\
e` action in the `AccountsController`:

```ruby
def create
  @account = Account.new(account_params)
  result = NewRegistrationService
    .new(account: @account, user: current_user)
    .process_registration
```
```

```
  if result.success?
    redirect_to root_path, success: 'Your account has bee\
n created!'
  else
    @account = result.account
    render :new
  end
end
```

We would be remiss in creating a brand new file without at least some unit testing. While the functionality and behavior are tested through the save feature and controller specs, testing the Service Object as a unit would still be wise.

```
require 'rails_helper'

describe NewRegistrationService do
  subject(:registration_service) { NewRegistrationService\
 }

  describe 'create an accounts' do
    context 'from form input' do
      let(:account)         { FactoryBot.build(:account) }
      let(:bad_account)         { FactoryBot.build(:accou\
nt, {name: nil}) }
      let(:user)         { FactoryBot.build(:user) }

      before do
        allow(Slack::Notifier).to receive_message_chain(:\
new, :ping) { nil }
      end

      it 'creates an account' do
        expect{
          registration_service
```

```
          .new(account: account, user: user)
          .process_registration
        }.to change(Account, :count).by(1)
      end

      it 'does not create an account on invalid object' do
        expect{
          registration_service
          .new(account: bad_account, user: user)
          .process_registration
        }.to change(Account, :count).by(0)
      end

      it 'creates a user' do
        expect{
          registration_service
          .new(account: account, user: user)
          .process_registration
        }.to change(User, :count).by(1)
      end

      it 'sends a message to be passed to welcome email m\
ethod' do
        expect_any_instance_of(
          NewRegistrationService
        ).to receive(:send_welcome_email).once
        registration_service
        .new(account: account, user: user)
        .process_registration
      end

      it 'sends a message to be passed to slack method' do
        expect_any_instance_of(
          NewRegistrationService
        ).to receive(:notify_slack).once
```

```
        registration_service
        .new(account: account, user: user)
        .process_registration
      end
    end
  end
end
```

The spec is pretty straightforward and close to what would be used when testing a model. In this case, we're testing the outcome of the service by making sure an account and user were created in the first three specs. That is followed by two specs checking that methods are called on the success workflow.

A notable piece of code that you might see is the following block of Ruby, which stubs the third party Slack gems' code to return nil (implicitly) instead of letting it run the code, hit Slack's system and return an error:

```
before do
  allow(Slack::Notifier).to receive_message_chain(:new, :\
ping)
end
```

If you run your whole test suite, you will see this Slack issue come up in a few other places. Thus you will want to add the before block to your`accounts_spec.rb`, and the "allow" line to your `sign_up_processes_spec.rb` system spec.

Now would be an excellent time to commit your file changes if you have not been so far, and depending on your git management style, you may have been doing this work on a feature branch. If so, you would be able to `commit` and `merge` like so:

```
git add .
git commit -am "Registration Service Object"
git checkout main
git merge service-object
git push
```

6.3 More uses for Service Objects

In their most pure form, Service Objects are ideal for keeping controller actions looking clean and skinny. For example, with a simple nested service class structure, you can maintain something such as notifications reduced to a narrow set of small objects.

Let's take our current use case. Sure right now, the `slack-notifier` gem makes it easy to send a short string of text to a webhook. But imagine if you need to send notifications through other providers or even different methods within a provider. In this case, we will extract the Slack Webhook Notification workflow and move it into a service class.

One of the easiest ways to have long-term maintainability within a Rails application is to break up complex code into smaller, extendable, and reusable classes or modules. You can use class hierarchy and namespaces, providing sensible navigation through your file structure, much like Rails offers. How might we achieve this for the Notification Services? Here is what we have:

```
NotificationServices(folder services/notification_service\
s)

-- SlackWebhooks(class SlackWebhooks and /services/notifi\
cation_services/slack_webhooks)

---- NewAccount(class NewAccount and extends SlackWebhook\
s)
```

We'll go deeper into each particular file and its use in a second, but I wanted to give a sense of how services and classes can be composed. `NotificationServices` itself is used to namespace all Notification based Service Objects into a shared directory. This is for navigation and maintainability. `SlackWebhooks` is the first real class and will contain a method used to send a webhook message. The method will accept a few arguments, such as the `webhook_url` and `message`. This allows the message to be ambiguous for its callers. Finally, `NewAccount` is the first specific and mostly concrete class within this structure. This class will be instantiated and contain the logic/message that belongs to the `new account` workflow.

With the general overview out of the way, here is what is needed to set this Service Object Structure up:

Create a folder for `NotificationServices`:

```
mkdir app/services/notification_services
```

Now, at this level, for `SlackWebhooks`, you will create a folder and a file. First, the folder:

```
mkdir app/services/notification_services/slack_webhooks
```

Now to create a file that will serve as a parent class for all use case-specific Slack Webhook Notification classes:

```
module NotificationServices
  class SlackWebhooks
    private

    def send_message(webhook, message)
      notifier = Slack::Notifier.new webhook
      notifier.ping message
    end
  end
end
```

That's it! That's all the code needed for the parent class. Since `NotificationServices` is merely our namespace, it is just declared as a module. Next, we declare the class `SlackWebhooks` with a private method for the children's classes to call with arguments. The code in this method matches what is in the `NewRegistrationService` slack method. Again, though, now used with generic arguments(`webhook, message`).

Now we can create the use case, specific class. A class that will be instantiated from the `NewRegistrationService`, passing in the `account` and `user` objects. As a use case class, the `webhook_url` and `message` to be sent can be supplied and do not need to be passed in as arguments.

```
module NotificationServices
  class SlackWebhooks
    class NewAccount < SlackWebhooks
      WEBHOOK_URL =
        'https://hooks.slack.com/services/some_hashed_id'\
.freeze

      def initialize(params)
        @user = params[:user]
        @account = params[:account]
      end
```

```
      def send_message
         super(WEBHOOK_URL, message)
      end

      private

      attr_reader :user, :account

      def message
         "A New User has appeared! #{account.name} - \
          #{user.name} || ENV: #{Rails.env}"
      end
    end
  end
end
```

There are a few concepts you may be used to from this chapter in this class and some new ones you may not have seen before. Let's start with the class/module declarations. Since this file is nested in `notification_services/slack_webhooks`, it will also nest those class declarations. Thus, the name and placement of `class NewAccount < SlackWebhooks` may be no surprise. However, the `< SlackWebhooks` may catch you off guard. What is happening is that this child class is extending the functionality of the parent class. For example, many books may use the `Animal < Dog` example for Class Inheritance. This extending will allow you to call the parent's class method of the same name to achieve the more comprehensive `SlackWebhooks` message sending functionality.

Another change you may notice is that `WEBHOOK_URL` is in all CAPS. No, we're not shouting for no good reason here. When a variable starts with a capital letter, it is not a variable but a `Constant`. The idea of a `Constant` is that its value should remain constant for the execution duration. However, since Ruby can play it a little loose on rules, this isn't entirely true, and Constants are technically able

to be mutated. This problem is where the `.freeze` comes in. This method, in the end, will now add just a bit more immutability to the constant and some Ruby execution optimizations. Lastly, the widely accepted format for `Constant` names is all capital letters with underscores (SCREAMING_SNAKE_CASE). This helps the constant's declarations stand out just a little bit.

```
def initialize(params)
  @user = params[:user]
  @account = params[:account]
end

def send_message
   super(WEBHOOK_URL, message)
end

...

def message
   "A New User has appeared! #{account.name} - \
    #{user.name} || ENV: #{Rails.env}"
end
```

The `initialize` method and public method used to trigger operations should be pretty standard to you so far in this chapter. This class makes no exceptions for their use. The primary public method of this class is what makes this class interesting.

At first, the message string creation is nothing crazy. An interpolated string, as it was before in the `NewRegistrationService`. Just now is being created inside a method for good class hygiene. You will see something new, `super`. As mentioned earlier, the extending syntax allows this child class to have unique interactions with its parent class. In simplest terms, we're calling the parent classes `send_message` method with the needed arguments:

```
def send_message(webhook, message)
  notifier = Slack::Notifier.new webhook
  notifier.ping message
end
```

To round this all out, you can now replace the `notify_slack` method in `app/services/new_registration_service.rb` with a call to the `NewAccount` service:

```
def notify_slack
  NotificationServices::SlackWebhooks::NewAccount
    .new(
      account: account,
      user: user
    )
    .send_message
end
```

If you re-run the current service specs, you should see they still pass, which signifies the behavior remains the same, even after refactoring the Notifications into their Service Object structure.

Here are the two test files you should create, though. Stop reading here if you want to try and test it yourself, as it will be very similar to how the `NewRegistrationService` was checked.

First, testing the `SlackWebhooks` class and making sure the `send_message` method responds:

```
require 'rails_helper'

describe NotificationServices::SlackWebhooks do
  describe 'the SlackWebhooks service object' do
    context 'from form input' do
      let(:account) { FactoryBot.build(:account) }
      let(:user) { FactoryBot.build(:user) }

      it 'responds to send_message' do
        expect_any_instance_of(
          NotificationServices::SlackWebhooks
        ).to receive(:send_message).once
        NotificationServices::SlackWebhooks::NewAccount
          .new(account: account, user: user)
          .send_message
      end
    end
  end
end
```

Then followed by a spec for the instantiated class `NotificationServices::SlackWebhooks::NewAccount`:

```
require 'rails_helper'

describe NotificationServices::SlackWebhooks::NewAccount \
do
  describe 'the SlackWebhooks::NewAccount service object'\
 do
    context 'from form input' do
      let(:account) { FactoryBot.build(:account) }
      let(:user) { FactoryBot.build(:user) }

      it 'responds has the correct message' do
        expect(
```

```
          NotificationServices::SlackWebhooks::NewAccount\
.new(
            account: account, user: user
          ).send(:message)
        ).to eq(
          "A New User has appeared! #{account.name} - \
          #{user.name} || ENV: #{Rails.env}"
        )
      end

      it 'responds to send_message' do
        expect_any_instance_of(
          NotificationServices::SlackWebhooks::NewAccount
        ).to receive(:send_message).once
        NotificationServices::SlackWebhooks::NewAccount
          .new(account: account, user: user)
          .send_message
      end
    end
  end
end
```

Now that you have some Service Object experience under your belt, you can now begin to add more features and functionality. Now in an even more clear and concise way. You will no longer worry about abundant methods on your model or 15-line controller actions. Instead, you can whip up a single Service Object with a few methods to support your workflow when you need a quick workflow. If you have a more considerable breadth of functionality, you can create a Service Object Structure containing multiple levels of namespacing and/or class inheritance.

Lastly, you can go ahead and `git` your latest changes. If you're following the branch and merge technique, you can use the following to merge the branch back in:

```
git add .
git commit -am "Notification Service Object Structure"
git checkout main
git merge notification-service-object
git push
```

6.4 What did you learn in this chapter?

- You learned about what a Service Object is and why it is used.
- You learned more about how Ruby classes can be used and configured.
- You created a Service Object to handle New Registration Accounts.
- You created a Service Object Structure for handling Notifications

6.5 Exercise

1. Now that you have a Slack Webhook notification under your belt try adding another third-party communication service such as Microsoft Teams.

7. Background Jobs, ActiveJob and Sidekiq

7.1 What is a Background Job, and why use one?

You may or may not have heard the term “background job” before. If you haven’t, what is a “background job.” A background job, sometimes called an asynchronous job(or task), is a small amount of code running off the primary web requests cycle. When attempting to serve web requests in the most performant way possible, you will need to unload long-running tasks into this asynchronous pattern.

Imagine that you need to send an email after a specific task is completed within your web application. You wouldn’t and don’t need the web request to wait for the email to be sent. Instead, you will want to drop the email processing(creation and delivery) into a background job.

Any work being done in a background job could be designed to process inline(such as right inside the controller action). There are two reasons why a task may be unloaded into a background job. First, the code may rely on a third-party API/Service/tool to complete long-running tasks. If the code was in line, the whole web request could hang. Thus, causing usability issues and consuming all the available threads on the application server. The code that lives in these background jobs should not have any bearing on the requests/response cycle results. Suppose, after adding the user to the database, we send a “Welcome” email with all the correct meta information. The end-user would likely not want this change to hinge upon the application sending a “Welcome” email out

immediately.

As you can see, there are two indicators that "some code" may be a suitable candidate for a background job:

1. That "code" relies heavily on a third-party service where the request/ response cycle should not be responsible for that task's success/failure pattern(i.e., Email sending, Slack notifications, payment processing).
2. That "code" will perform a long-running task or queue of jobs that will need to run independently of the web's request/response cycle. This scenario can typically be bulk operations or actions.

Lastly, another feature of background job processing is tuning. In most use cases, you can fine-tune the background processor to match the available resources. Additionally, each task's class will be able to apply specific settings to particular job instances. These parameters can include retry patterns or queue associations.

7.2 Right Tools for the "background" job

As you continue to build your SaaS application into a sound, production-ready product, background processing will be yet another great tool in your toolbox. Now, with most open-source tools, there are options. In fact, with Rails, you can use one of many different background job processing frameworks. Frameworks include using Rails' built-in framework, which can delegate background processing to another framework. Meta, huh?

7.2.1 ActiveJob

A book about building a SaaS application in Rails would never go without a discussion of ActiveJob.

ActiveJob is Rails' built-in framework for creating job classes and providing methods to run or enqueue the jobs. The meta aspect of ActiveJob is that it presents an API for other gems to use. Gems such as Sidekiq, Sneakers, or SuckerPunch can use a consistent set of methods or API instead of possible library-specific methods. ActiveJob allows you to use these background job libraries in a "plug and play" fashion.

That isn't to say there aren't disadvantages to using ActiveJob only when creating and running jobs. For example, many background job libraries will have beneficial job settings that may not be available through the ActiveJob API. Thus, you would need to write your job class in that library's syntax and specification to use those.

7.2.1.1 Using ActiveJob

Since ActiveJob is built into Rails' you do get some niceties. One of those is the ability to create a job file(along with a test file) from a Rails generator. To get your application up and running with ActiveJob, we will create a job to send a slack notification when a user signs up. If you do not use Slack, you can substitute another service or follow the pattern in your jobs.

The job will be created in the `app/jobs` folder and will resemble this stubbed out job file:

```
class SlackNotificationJob < ApplicationJob
  queue_as :default

  def perform(*args)
    # Do something later
  end
end
```

Anywhere else in your code, you can enqueue the job by using:

```
SlackNotificationJob.perform_later(user)
```

To continue implementing this notification job, we will now fill in the `# Do something later` portion of the code in the job file:

```
class SlackNotificationJob < ApplicationJob
  queue_as :default

  def perform(user)
    NotificationServices::SlackWebhooks::NewAccount
    .new(
      account: user.account,
      user: user
    )
    .send_message
  end
end
```

Now, you can add the enqueuing syntax to your service object's method for sending a notification:

```
class NewRegistrationService

  Result = ImmutableStruct.new(:success?, :user, :account\
, :error)

  def initialize(params)
    @user = params[:user]
    @account = params[:account]
  end

  def process_registration
    account_create
    send_welcome_email
    notify_slack
    Result.new(success: true, user: user, account: accoun\
t, error: nil)
  rescue ActiveRecord::RecordInvalid, Slack::Notifier => \
exception
    Result.new(
      success: false, user: user, account: account, error\
: exception.message
    )
  end

  private

  attr_reader :user, :account

  def account_create
    post_account_setup if account.save!
  end

  def post_account_setup
    user.account_id = account.id
    user.save!
```

```
    user.add_role :admin, account
  end

  def send_welcome_email
    # WelcomeEmailMailer.welcome_email(user).deliver_later
  end

  def notify_slack
    SlackNotificationJob.perform_later(user)
  end
end
```

With your notification job added, when you create an account, it will send a notification to Slack or another notification service.

ActiveJob will use an in-memory queuing system to process background jobs by default. While requiring no setup out of the box, it does not offer advanced features or recovery patterns.

Here is a comparison of all adapters ActiveJob can use and their features:

Name	Async	Queues	Delayed	Priorities	Timeout	Retries
Backburner	Yes	Yes	Yes	Yes	Job	Global
Delayed Job	Yes	Yes	Yes	Job	Global	Global
Qu	Yes	Yes	No	No	No	Global
Que	Yes	Yes	Yes	Job	No	Job
queue_classic	Yes	Yes	Yes	No	No	No
Resque	Yes	Yes	Yes(gem)	Queue	Global	Yes
Sidekiq	Yes	Yes	Yes	Queue	No	Job
Sneakers	Yes	Yes	No	Queue	Queue	No
Sucker Punch	Yes	Yes	Yes	No	No	No
Active Job Async	Yes	Yes	Yes	No	No	No

Name	Async	Queues	Delayed	Priorities	Timeout	Retries
Active Job In-line	No	Yes	N/A	N/A	N/A	N/A

Now that we know about ActiveJob's abstractions, benefits, and shortcomings, we can decide which library to use to fit our needs best.

7.2.2 Sidekiq

Sidekiq is now the most well-known and widely used of the Ruby background job frameworks. Sidekiq is based on large portions of the Concurrent::Ruby library. It also uses Redis as a memory store to store queue data, statistics, and more.

Using Redis allows the job store to be hosted in a separate service, giving more reliability, resource control, and fault tolerance. Redis is an in-memory key/value store that allows Sidekiq to quickly manage background job data. Sidekiq version 4.0.0 can process around 100,000 jobs in 22 seconds and 4500 jobs/sec throughput. It's fast.

7.2.3 Sneakers

`Sneakers` is very similar to Sidekiq in that it is a background job processor that uses an external store. However, instead of Redis, Sneakers uses RabbitMQ. RabbitMQ is a message broker system, or more precisely, a system built specifically for passing messages around to other processes and servers.

While Sneakers may be in a similar place in performance, the ecosystem and knowledge base for SideKiq provide a better developer experience. The author of Sneakers states, "if you do not need

long-running background jobs, there's really no difference between Sneakers and SideKiq".

7.2.4 SuckerPunch

Sucker Punch is a single process in-memory background job processing library. The benefit here is that there is no external service, data store, or additional server needed to run background jobs.

Like the others, it allows you to queue up a job and let it run asynchronously in the background. However, being in memory, any server process restarts will erase the memory used by the Rails application, which will likely delete any pending jobs that had not been completed. Therefore, per the library's documentation, SuckerPunch could only be recommended for non-mission-critical jobs such as logs, emails, etc.

7.2.5 Other Notable Mentions

There are some other background processing libraries out there. For example, you may have seen Delayed::Job at some point. Delayed::Job was created by Shopify and works by using a "jobs" table in your current database that your Rails application is using. The upside is that there is nearly no extra work plugging this library into your application and ActiveRecord. However, Delayed::Job was known to add additional stress to the database service/server used for the Rails application.

Another valuable library is Resque. Resque was the Redis-backed predecessor to SideKiq. The main difference between Resque and SideKiq is that SideKiq is multithreaded. With a similar setup to Resque, Sidekiq is merely more efficient.

7.3 Using SideKiq with ActiveJob

In this book, we will choose to use SideKiq for its ease of use and vast ecosystem. First, however, we will use Sidekiq through its ActiveJob API for almost all cases.

Let's ensure the SideKiq gem is included in your Rails app by adding `gem 'sidekiq` if it is not already in there and running `bundle install`. Next, we will make sure you have Redis on your system. Run the `brew install redis` command if you are on OSX(a quick google search of "install Redis *your OS and version* should help you out otherwise). Once finished installing Redis through brew, you will be able to use the brew provided command to add Redis to your startup and start the service.

To add SideKiq to ActiveJob is also relatively simple in itself. You will need to add one line to your `config/application.rb` file to tell your Rails app to use SideKiq as an ActiveJob adapter:

```
class Application < Rails::Application
  # ...
  config.active_job.queue_adapter = :sidekiq
end
```

To run the Sidekiq process(es), you will want to run the following command in another terminal tab/pane/window:

```
bundle exec sidekiq -q default
```

Lastly, to get this running as part of the `Procfile` in Foreman, just update your `Procfile.dev` to look like:

```
web: bin/rails server -p 3000
css: bin/rails tailwindcss:watch
worker: bundle exec sidekiq -q default
```

After a quick restart of `bin/dev`, you will be able to run the previous `SlackNotificationJob`. Now background jobs process through ActiveJob using Sidekiq.

7.4 Other ActiveJob tips and tricks

One setting that may help you organize and/or prioritize background jobs is using multiple queues. As your codebase grows, you may start splitting out your background jobs into different queues with queue names such as mailers, notifications, or data imports. With jobs divided into different queues, you can run queues under separate processes. Even just as different priorities if your library supports it(SideKiq does).

To set a queue, you will need to apply this setting to the job:

```
class SlackNotificationJob < ApplicationJob
  queue_as :notifications

  def perform(user)
    NotificationServices::SlackWebhooks::NewAccount
    .new(
      account: user.account,
      user: user
    )
    .send_message
  end
end
```

However, this can be an advanced technique and is usually not needed until background job performance becomes an issue.

With that being said, there is one place a queue will come into play without you specifying any particular queue settings. First, I'll explain what that is and then a simple fix.

ActiveJob integrates tightly with ActionMailer. This integration gives you a shortcut to asynchronously send any email you are using a Rails Mailer to send. To send an email immediately in the current thread/request you can use `YourMailer.send_welcome(arg).deliver_now`. If you would like ActiveJob to pick up the email and send it in the background, just use `YourMailer.send_welcome(arg).deliver_later`.

The change here tells SideKiq to watch two different queues within its worker process. One of those queues is the default job queue, and the other is Rails' mailer queue.

When testing ActiveJob jobs, you can use some of the built-in RSpec helpers as such:

```
require "rails_helper"

RSpec.describe SlackNotificationJob do
  it "matches with enqueued job" do
    ActiveJob::Base.queue_adapter = :test
    SlackNotificationJob.perform_later
    expect(SlackNotificationJob).to have_been_enqueued
  end

  it 'calls the Slack service' do
    user = FactoryBot.create(:user)
    ActiveJob::Base.queue_adapter = :test
    expect_any_instance_of(NotificationServices::SlackWeb\
hooks::NewAccount).to receive(:send_message)
    SlackNotificationJob.perform_now(user)
  end
end
```

Here you are testing whether the job gets enqueued when asked to perform later, and the second test checks whether or not the `SlackWebhooks` is called as expected. This syntax is provided by `rspec-mocks` which allows you to test against objects and how they are called, including mocking out responses (which will be covered in later chapters).

As stated before, you can write a background job in the native libraries API. The cases I have seen at this point are few and far between, but if you wanted to manage the job's retry limits, you could rewrite it like so:

```
class SlackNotificationJob
  include Sidekiq::Worker
  sidekiq_options :retry => 5 # Only five retries and the\
n to the Dead Job Queue

  def perform(user_id)
    user = User.find(user_id)
    NotificationServices::SlackWebhooks::NewAccount
    .new(
      account: user.account,
      user: user
    )
    .send_message
  end
end
```

And it would only retry five times instead of the typical exponential backoff algorithm used by SideKiq. This process will retry less and less frequent periods until six months have passed or the job is deleted.

> ActiveJob allows you to pass objects through as parameters to make it easier to handle passing around the

objects you are going to need to act upon. However, Sidekiq does *not* allow this and requires you to pass an ID, then look it up in the worker.

7.5 Scheduled Jobs

Sometimes you don't need a background task to run on demand, but more on a schedule. Just when you were worried that we would be adding a bunch of code to solve this problem, there is a 3rd party gem that will take care of this. The gem is called sidekiq-cron[1]. Sidekiq-cron will help us run sidekiq/ActiveJob workers on a schedule.

So, how will this help us? To be able to send emails or notifications for a `Teams` standups, we will need to run a background job at the same intervals as the choices they had :00, :15, :30, and :45. At those times, the background job would query for Teams that:

- have a `true` set for reminders or recaps(depending on which job)
- the time matches the current time.
- have the Standup set for the particular day of the week.

The results of that query will be iterated upon, firing individual jobs for each `team` to gather information. A list of users will be sent an email reminder or recap in that information.

Let's start by adding the following to the `Gemfile` and running `bundle install`:

```
gem "sidekiq"
gem "sidekiq-cron" # <-- new gem
```

[1] https://github.com/ondrejbartas/sidekiq-cron

We will delay any further setup for that gem until we create the Job files we will need. Let's start by adding a file that will query for Teams that have `has_reminder` set to `true` set for reminders, and the time matches the current time.

In this case, it is wise to namespace and put files into folders to keep the jobs directory relatively clean as we begin to increase job files.

In this case, create a folder `app/jobs/reminders`, then add a file `app/jobs/reminders/find_teams_job.rb`:

```
module Reminders
  class FindTeamsJob < ApplicationJob

    def perform(*_args)
      teams.each { |team| Reminders::EmailUserOnTeamJob.p\
erform_later(team) }
    end

    private

    def teams
      @teams ||=
        Team
        .includes(:days)
        .where(
          has_reminder: true,
          reminder_time: Time.at(last_fifteen_marker).utc
        )
        .where('days_of_the_week_memberships.day = ?', cu\
rrent_day)
        .references(:days_of_the_week_memberships)
    end

    def current_day
      DaysOfTheWeekMembership.days[Time.now.utc.strftime(\
'%A').downcase.to_sym]
```

```
    end

    def last_fifteen_marker
      Time.now.utc.to_i - (Time.now.utc.to_i % 15.minutes)
    end
  end
end
```

Let's go over the file. First, as noted earlier, jobs will automatically call the `.perform`-method with any parameters passed in. In the case of a scheduled job, we are currently not anticipating using any sent arguments. Using the '_syntax' on ignored arguments is common.

`teams` will call the private `teams` method, which will query the database for `teams` that have reminders turned on, have the time set to the current 15-minute interval, and are set for the current day of the week. With that said, this query mirrors some of those seen in Chapter 5. It included association data and used `references` to ensure ActiveRecord has access to the joined data.

The first bit of unique code may be the `last_fifteen_marker` method. This method will return the most recent past 15-minute marker. As the scheduled job may run shortly after, we need to make sure the queried time is in a format that is saved in the `team`'s record.

The intriguing part is that we calculate the integer saved to the database in the `DaysOfTheWeekMemberships` table. Since we're using the `enum` feature from Rails on the model, we often interact with the textual representation of a day. But in the case of this lookup, we need to get the integer value of the current day.

Now, after adding `FindTeamsJob`, we will want to add a file that will iterate over the user in each particular team. Then send an email reminder, `app/jobs/reminders/email_user_on_team_job.rb`:

```
module Reminders
  class EmailUserOnTeamJob < ApplicationJob

    def perform(team)
      team.users.each do |user|
        EmailReminderMailer.reminder_email(user, team).de\
liver_later
      end
    end
  end
end
```

We will go in-depth with Mailers and ActionMailer in the next chapter. But, for now, we will go through the quickest possible setup to get this whole scheduled cron running.

7.5.0.1 Getting The Mailer Running

You will want to update the `app/mailers/application_mailer.rb` file to have a reasonable default `from` address. For example, the value could be `default from: "'Standup App' <Standups@standup-app.com>"`.

Now we are down to a few more files to get this working! First, we will be adding an actual mailer file and the particular email's template. So let's get started with the mailer!

First is the mailer file, which is responsible for the action of telling Rails how to send a particular piece of email. Add the file `app/mailers/email_reminder_mailer.rb`:

```
class EmailReminderMailer < ApplicationMailer

  def reminder_email(user,team)
    @user = user
    @team = team
    make_bootstrap_mail(to: @user.email, subject: "#{team\
.name} Standup Reminder!")
  end
end
```

The mailer is pretty generic itself. We are turning the arguments into instance variables to use in the email template. Then, it uses a `make_bootstrap_mail` that looks like the same `mail` method inherited by ActionMailer to start the ActionMailer delivery process.

We will use Bootstrap Email to make creating email templates and markup as easy as using Bootstrap on the web. Now, I know we are using TailwindCSS for our application's styling, but there does not seem to be a Rails mailer equivalent that has been built. The good news is that the scope for learning Bootstrap for email will be straightforward and will use a very simplified Bootstrap 4 Grid format.

First, we will add the following line in `Gemfile` (anywhere in the general gem list area) to add the gem to our project:

```
gem 'bootstrap-email'
```

and run

```
bundle install
```

Next will be a few pieces of general setup for the gem's use. We will add a layout, set the layout to be the default, and add the gem's styles to the Rails application.

First up is the layout:

Create a file `/app/views/layouts/bootstrap-mailer.html.erb` and paste this HTML into it.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" \
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html lang="en">
  <head>
    <meta http-equiv="Content-Type" content="text/html; c\
harset=utf-8">
    <meta name="x-apple-disable-message-reformatting">
    <meta http-equiv="x-ua-compatible" content="ie=edge">
    <meta name="viewport" content="width=device-width, in\
itial-scale=1">
    <meta name="format-detection" content="telephone=no, \
date=no, address=no, email=no">
  </head>
  <body class="bg-light">
    <%= yield %>
  </body>
</html>
```

Now we will adjust the `ApplicationMailer` (`app/mailers/application_mailer.rb`) one more time to set the default layout (what we just added):

```
class ApplicationMailer < ActionMailer::Base
  default from: "'Standup App' <Standups@standup-app.com>"
  layout 'bootstrap-mailer'
end
```

Last, we will add the actual email template. For brevity regarding this chapter's content, we will let the Bootstrap Email gem do its magic with a simple email template. To do so, just create the file at `app/views/email_reminder_mailer/reminder_email.html.erb` and add the following markup:

```
<div class="container">
  <h1 class="text-center mb-4">
    Standup App
  </h1>
  <div class="card mb-4">
    <div class="card-body">
      <h3 class="text-center mb-4"><%= @team.name%> Remin\
der! </h3>
      <p class="mb-4">
        Just wanted to remind you to add your standup for
        the team: <%= @team.name %>
      </p>
      <%= link_to "Add Your Standup", new_standup_url(), \
{class: "btn btn-primary btn-lg mx-auto mt-2", style: "wi\
dth:95%"} %>
    </div>
  </div>
</div>
```

7.5.0.2 Getting the Job Scheduled

Now that the job files and mailer files have been added, we can add the following setup information to schedule the job:

```
jobs_hash = {
  'reminder' => {
    'class' => 'Reminders::FindTeamsJob',
    'cron'  => '0,15,30,45 * * * *',
    'active_job' => true
  }
}

if Sidekiq.server?
  Rails.application.config.after_initialize do
    Sidekiq::Cron::Job.load_from_hash! jobs_hash
```

```
  end
end
```

The file is added to the initializer folder as these files get loaded at the application boot up. Aside from starting the Rails server, whenever any Sidekiq server processes startup, the application is booted.

In the file, a hash is used to add the settings for a particular scheduled job. First, it needs a unique hash key for a name. Then, you supply the job's class name, cron schedule and state that it is an ActiveJob job.

It may be essential to mention the cron format if you have never seen it before. The quickest explanation is that columns or space separated value represents one of the time levels. For example, the first column is minutes, then hours, the day of the month, month, and day(s) of the week.

If you reexamine the cron schedule used, `0,15,30,45 * * * *`, it is saying run this at :00, :15, :30, and :45 of every hour, day, and month.

Now would also be a great time to mention that you can enable a Web UI for Sidekiq. Additionally, a good percentage of 3rd party gems that plug into Sidekiq will also have UIs. You can allow the UI by adding the following to your `config/routes.rb` file, right before the `root to:` line:

```
require "sidekiq/web"
require 'sidekiq/cron/web'
mount Sidekiq::Web, at: "/sidekiq"
```

You should see an interface like this:



Sidekiq UI

That wraps up setting up your first scheduled job. But, first, let's write a few tests to ensure the added functionality works as intended and continues to work as expected.

Like many other types of files, job specs will go in their directory, which helps RSpec infer the test type. In the case here, you will create a `jobs` directory in the `specs/` directory. Plus, in the event of these `reminder` namespaced jobs, you will create another folder within `spec/jobs`, called `reminder`. So let's first write a test for the `Reminders::FindTeamsJob`:

```
require 'rails_helper'
include ActiveJob::TestHelper

RSpec.describe Reminders::FindTeamsJob do
  it 'matches with enqueued job' do
    ActiveJob::Base.queue_adapter = :test
    Reminders::FindTeamsJob.perform_later
    expect(Reminders::FindTeamsJob).to have_been_enqueued
  end

  it 'finds a team with reminders' do
    team = FactoryBot.create(
      :team,
      has_reminder: true,
      reminder_time: Time.at(
        Time.now.utc.to_i - (Time.now.utc.to_i % 15.minut\
es)
      ).utc
    )
    team.update(
      days: [DaysOfTheWeekMembership.new(
```

```
        team_id: team.id,
        day: Time.now.utc.strftime('%A').downcase
      )]
    )
    job = Reminders::FindTeamsJob.new
    job.perform_now
    expect(job.instance_variable_get(:@teams)).to eq([tea\
m])
  end
end
```

In this spec file, we're testing two things. First, by using the regular job enqueuing syntax, a job will be enqueued correctly. Second, after setting up a `team` with all of the needed attributes, a test that the team is found in the `Reminders::FindTeamsJob`'s querying logic.

Next, let's create a spec for `Reminders::EmailUserOnTeamJob`:

```
require 'rails_helper'
include ActiveJob::TestHelper

RSpec.describe Reminders::EmailUserOnTeamJob do
  let(:user) { FactoryBot.create(:user) }
  let(:team) do
    team = FactoryBot.create(
      :team,
      user_ids: [user.id],
      has_reminder: true,
      reminder_time: Time.at(
        Time.now.utc.to_i - (Time.now.utc.to_i % 15.minut\
es)
      ).utc
    )
    team.update(
      days: [DaysOfTheWeekMembership.new(
        team_id: team.id,
```

```
        day: Time.now.utc.strftime('%A').downcase
      )]
    )
    team
  end

  it 'matches with enqueued job' do
    ActiveJob::Base.queue_adapter = :test
    Reminders::EmailUserOnTeamJob.perform_later
    expect(Reminders::EmailUserOnTeamJob).to have_been_en\
queued
  end

  it 'enqueues a mailer based job' do
    ActiveJob::Base.queue_adapter = :test
    job = Reminders::EmailUserOnTeamJob.new(team)
    expect { job.perform_now }
      .to have_enqueued_job
  end

  it 'reminder email is sent' do
    expect {
      perform_enqueued_jobs do
        EmailReminderMailer.reminder_email(user, team).de\
liver_later
      end
    }.to change { ActionMailer::Base.deliveries.size }.by\
(1)
  end
end
```

In this job_spec, we're running three specs. The first is just like the first spec in the previous spec file. We are testing the job gets enqueued. The following spec proves that if a team with a user is sent to the job, it will enqueue a mailer job. Finally, the last spec

tests the mailer, creates a mailer instance, and delivers the mail. Running your test suite should result in a passing suite:

```
rspec
.....................................................\
.....................................................DEBUGG\
ER: Attaching after process 31454 fork to child process 3\
1471
.............................

Finished in 10.63 seconds (files took 3.88 seconds to loa\
d)
139 examples, 0 failures

Coverage report generated for RSpec to standup_app/covera\
ge. 383 / 403 LOC (95.04%) covered.
```

7.6 What did you learn in this chapter?

Even though this chapter did not go into painful detail on creating dozens of background jobs, it covered the basics used in subsequent chapters concerning ActiveCable and payment processing.

- You learned what a background job is and when you would use it.
- You learned how to create a background job in Rails
- You learned about a handful of different background job libraries

You learned how to integrate Sidekiq into your Rails application, get it up and running, and finally use the native API to write a job.

- How to schedule jobs to be run on a specific schedule.
- The very basics of a minimal ActionMailer mailer
- What a `Procfile` is, how it is used, and some basic configuration options.

7.7 Exercises

Exercise answers can be found with any package or bundle purchases of Build A SaaS App in Ruby on Rails 7[2]!

1. Create a job that would handle updating the user after account creation. While this may be a little redundant, it will give you more experience creating and using background jobs.

[2] https://buildasaasappinrails.com

8. Mailers and You

8.1 What is a Mailer?

`ActionMailer` is the built-in Rails system for sending and receiving emails through your application. As usual with Rails, ActionMailer provides a quick and easy way to get up and running with a complex system with a series of generators and conventions.

When sending an email, ActionMailer will help you by providing a way to configure an email-level set of settings. Plus, it will assist you in preparing your email template(which will be used very much like standard templates) with instance variables to access information from your application's database.

On the template side, you can specify both an HTML template and a plain text template. Plain text will use ERB, Slim, or any other templating engine you will use to add specific information to your outbound email, just as if it were a template used to provide information inside your web application.

Mailers live in the `app/mailers` directory and the respective templates will be placed in folders, `app/views/*mailer_name*_mailer/*method_name*.html.erb`. Not surprisingly, the actual mailer classes will inherit from the base Rails Mailer class called `ActionMailer::Base`.

Mailers can easily be created with a Rails CLI command, and in this case, its syntax would be:

```
bin/rails generate mailer SomeMailer
```

This command will generate a few files, such as any top-level Mailer files. For example, `ApplicationMailer` will act like

`ApplicationController` does for controllers and layout files for mailers.

Additionally, the Mailer will generate a nearly blank Mailer file, in which you will include a method that performs the mail action with a mail method. Lastly, the Mailer generates a folder in `app/views`.

After creating content for the templates, the Mailer will be ready to be called from your application. Though this has been briefly mentioned in earlier chapters, it bears being reiterated. You can chain two methods to your Mailer's sending method, `deliver_now` and `deliver_later`. `deliver_now` is simple and does what it says it would do. That method will send the email inline to where it is included. For example, if you had `SomeMailer.email_method(arg).deliver_now`, the request/response will wait for the Mailer to process the outgoing email and then continue with the action method's other duties. However, suppose you used `SomeMailer.email_method(arg).deliver_later`. In that case, it will use the background job processor of your choice, ActiveJob, if you used the setup from the previous chapter to asynchronously perform the outgoing email sending process outside of the request/response cycle of the particular Rails controller and action.

Finally, you will want to test your mailers, as you do any other code you have in your codebase. As mailers are inherited from ActionMailer::Base, you will have a bunch of Rails and RSpec-based testing methods and tools at your disposal. Like most scenarios, there are two approaches to testing mailers. First, you will be able to create Unit level specs by writing mailer specs that test that mail get's queued, sent, or has the correct necessary delivery attributes. Additionally, you can write a system-level specs test where the email came from(such as the sign-in process) to see that the appropriate browser behavior triggered the sending and delivery of the Mailer's email.

8.2 Using a Mailer in your application

8.2.1 Adding a Welcome Email

The best way to jump into ActionMailer and its use cases will be to create a new mailer for an outgoing email that has been skipped a few times in this book, the `Welcome Email`.

Before we get going on the mailer, let's go ahead and spin up another branch off of master:

```
git checkout -b welcome-mailer
```

Per the usual, Rails has a CLI command to generate the mailer, templates, and a few other files to get the process going:

```
bin/rails generate mailer WelcomeEmailMailer
      create  app/mailers/welcome_email_mailer.rb
      invoke  tailwindcss
      create    app/views/welcome_email_mailer
      invoke  rspec
      create    spec/mailers/welcome_email_mailer_spec.rb
      create    spec/mailers/previews/welcome_email_maile\
r_preview.rb
```

A few things in there match what was mentioned before, such as the file and folder naming conventions. However, a few more things may have caught your eye.

```
create    spec/mailers/welcome_email_mailer_spec.rb
create    spec/mailers/previews/welcome_email_mailer_prev\
iew.rb
```

The first file listed isn't a surprise, as Rails will usually try and create a test/spec file in almost any generator. The second file, though is odd, right? `spec/mailers/previews/welcome_email_mailer_preview.rb`? What is a preview file? This file allows you to define a simple scoped method that will allow you to preview the email in the browser, using information such as the `current_user` or a predefined user(`User.first`) to build and display the email. This process can usually be easier than trying to trigger the email manually or through the Rails console. Though, one could argue that it's nearly as easy with the `letter_opener` gem and Rails' console. I'll leave it up to you to pick your preferred route after showing you how to use the `preview.rb` method.

Let's take a step back and mention mailer layouts again. A mailer's layout template function is just like other layout templates that expect to `=yield` the current rendered template inside. However, as you noticed, we left the file with only one line to `=yield` out the content from the individual mailer templates. I find that a personal preference as outgoing emails vary as they go and are built for various reasons.

If there is an area of convergence, you can set the mailer layout on a case-by-case basis inside the mailer file.

Now, when you think about it, the creation of a mailer is very similar to the creation of controllers. In fact, at a high level, they are not so dissimilar. They are handling the collection of business logic, database information, and finally presenting that through a template. The main difference stems from the controller responding to incoming router-based requests and the mailer usually being provoked programmatically and preparing an email to be sent.

When the Rails mailer generator creates a new mailer, it generates

a new empty class, with nothing more than a class name matching the file name for autoloading to work correctly.

title="app/mailers/welcome_email_mailer.rb BEFORE"}

```
class WelcomeEmailMailer < ApplicationMailer
end
```

Then, to begin to turn this into a functional mailer, we will add a few lines of configuration and a method:

```
class WelcomeEmailMailer < ApplicationMailer
  default from: "'Standup App' <welcome@standup-app.com>"

  def welcome_email(user)
    @user = user
    make_bootstrap_mail(to: @user.email, subject: "Welcom\
e to Standup App, #{user.name}!!")
  end
end
```

Up near the top, we are setting the `from` email address to be one that will be perceived as sending the welcome email. The `default` syntax can take a hash of a few mailer-specific settings. Some of the settings you can change are `mime_version`, `charset`, `content_type`, or any other static value that can usually be passed into the `mail` method. One such case could be a static subject line that does not change on an email-by-email basis in a mailer. Also, the `default` hash takes two more things. One would be that it can use any header you anticipate using for an email as long as the key is set as a string(a header such as "SomeWeirdGmailThingIMadeUp": "12yay". Lastly, a proc(`Proc.new {}`) can also be passed to have a method evaluated as the mailer is being evaluated. This could be useful if a special header with a very dynamic value is needed, such as `default 'X-Special-Header' => Proc.new { my_method }`.

The one(and only) public method in the file will also be used to match the template in just a few moments. The method will expect a user to be passed in, such as the first admin user on the account from the `NewRegistrationService`. Once inside the method, the incoming user argument is set to an aptly named `@user` instance variable that can be accessed in the template. The mail method, in most cases, will take two specific keyword arguments: `to:` and `subject:`. As the email address that the mailer email is being delivered to is usually dynamic to the argument passed in, the `to:` is specified in the mail method. As mentioned earlier, if the subject is not static, then you can also dynamically set the `subject` by sending an interpolated string in as the `subject:` argument. The mail method takes the same options as the `default` has provided a few minutes ago. You will want to specify those settings in the `mail` method instead of the `default` hash if they are specific to the dynamic `user` passed into the method.

Now that the mailer is set up, you can move on to the mail template(s). The best route here is to set both the HTML and Plain-text versions of the email. We can start with the HTML version:

```
<div class="container">
  <h1 class="text-center mb-4">
    Standup App
  </h1>
  <div class="card mb-4">
    <div class="card-body">
      <h3 class="text-center mb-4">Welcome <%= @user.name\
 %>!</h3>
      <p class="mb-4">
        Thank you for signing up to Standup App and we're
        glad to have you with us! Standup App was created\
 to
        help user's like yourself have an easy way to tra\
ck
        progress with other team members and users.
```

```
      </p>
      <p class="mb-4">
        The best way to get started is to invite more use\
rs
        to Standup App and set up some teams!
      </p>
      <%= link_to "Add Users to Standup App", account_use\
rs_url(), {class: "btn btn-primary btn-lg mx-auto mt-2", \
style: "width:95%"} %>
    </div>
  </div>
</div>
```

As you can see, the email itself is much like that created in the previous chapter. Again, specify a title, other HTML settings, and some very Bootstrap HTML elements to structure the email. This time, the messaging contains some verbiage to welcome the user, and I suggest you tailor that to the app or voice you use with your customers. Finally, a button link to the user's page will quickly guide users to an area to invite even more users.

Now, when we create our plain text version of the email, you will use the ERB template file extension. There is a good reason for that. `ERB` will merely spit out whatever you give it, much like an HTML file, and allow you to throw in some dynamic ruby content where needed, even if it is the text version of an email.

```
** <%= "Welcome #{@user.name}!" %>
---------------------------

Thank you for signing up for Standup App, and we're
glad to have you with us! Standup App was created to
help users like yourself have an easy way to track
progress with other team members and users.

The best way to get started is to invite more users
to Standup App and set up some teams!"

<%= "Add Users to Standup App: #{account_users_url()}"%>
```

As you can see, the content itself is very much like the HTML version, with a few characters added to help separate content. Finally, the `link_to` help is no longer used as it generates an HTML anchor; instead, we just use the path created by the `url()` based view helper.

Now to introduce the two ways to test a mailer's output. First, you can use the built-in mailer preview created from the Rails mailer generator. Otherwise, you can use the Rails console and manually call the WelcomeMailer, to deliver, which will be caught by the `letter-opener` gem to display the output in your browser.

To use the ActionMailer preview system, you will only need to make a few changes to the preview loader file in the `/spec` folder at `spec/mailers/previews/welcome_email_mailer_preview.rb`:

```
# Preview all emails at http://localhost:3000/rails/maile\
rs/
# welcome_email_mailer
class WelcomeEmailMailerPreview < ActionMailer::Preview
  def welcome_email
    WelcomeEmailMailer.welcome_email(User.first)
  end
end
```

The trick here is that you are creating a method with the same name as the method from your Mailer. Then, inside that, create an instance of the Mailer. In this case, it is sending in a user as an argument, just as the mailer users in the real world. For the sake of the preview, it only grabs the first user from the database. You can use anything here, a specific `User` from a `.find` query, etc.

The second way to preview an email is to generate an email for delivery and have it intercepted by the `letter-opener` gem. This gem shims into your development environment to provide a browser-based view of any mail sent. If you have `letter-opener` in your Gemfile, as you should from the third chapter, you will need to verify a `development` setting. If you don't have `letter-opener`, add the following to your Gemfile's `:development` group, `gem 'letter-opener`, and run a `bundle install`. You can add the following settings to the bottom of the `development` settings file:

```
Rails.application.configure do

...

  config.action_mailer.default_url_options = { host: 'loc\
alhost', port: 3000 }
  config.action_mailer.delivery_method = :letter_opener
end
```

The first setting adds an option allowing the URL view helpers to

generate a URL correctly. The second setting lets `letter-opener` function as the development environment's mailer system. Once completed, you can fire up a Rails console to manually trigger an email to be delivered through `letter-opener`:

```
rails c

:001 > WelcomeEmailMailer.welcome_email(User.first).deliv\
er_now
```

You should see the console output some information about the loading and rendering of the email, while your default browser will open a new window/tab with the email preview.

To finish the setup for this `WelcomeEmailMailer`, you will need to uncomment the Mailer's delivery invocation from the `NewRegistrationService`. From:

```
  def send_welcome_email
    ## No Op
    # WelcomeEmailMailer.welcome_email(user).deliver_later
  end
```

to:

```
  def send_welcome_email
    WelcomeEmailMailer.welcome_email(user).deliver_later
  end
```

Testing this mailer will fall into two areas, as discussed earlier. First, we will fill out the generated but empty mailer spec as more of a unit-level spec. Then we will add a few extra lines to the `spec/system/sign_up_processes_spec.rb`, to test that it works from the sign-up browser workflow.

The mailer spec is actually rather straightforward:

```
require 'rails_helper'
include ActiveJob::TestHelper

RSpec.describe WelcomeEmailMailer, type: :mailer do
  let(:user) { FactoryBot.create(:user, email: 'awesome@d\
abomb.com') }

  it 'job is created' do
    ActiveJob::Base.queue_adapter = :test
    expect do
      WelcomeEmailMailer.welcome_email(user).deliver_later
    end.to have_enqueued_job.on_queue('default')
  end

  it 'welcome_email is sent' do
    expect do
      perform_enqueued_jobs do
        WelcomeEmailMailer.welcome_email(user).deliver_la\
ter
      end
    end.to change { ActionMailer::Base.deliveries.size }.\
by(1)
  end

  it 'welcome_email is sent to the right user' do
    perform_enqueued_jobs do
      WelcomeEmailMailer.welcome_email(user).deliver_later
    end

    mail = ActionMailer::Base.deliveries.last
    expect(mail.to[0]).to eq user.email
  end
end
```

This spec file houses three specs. The first will ensure the mailer's job is correctly placed in the mailer queue. This means that it is

invoked and sent for background processing. The second one takes the process a little further and uses an ActionMailer method to check the `deliveries` array size to ensure it has increased. Finally, the last one checks the email itself was sent to the correct email address in the test environment.

We can follow up with an addition to the system spec mentioned a few minutes ago. So here we are looking at the declarations, a new include, and a revised spec:

```
require 'rails_helper'
include ActiveJob::TestHelper
ActiveJob::Base.queue_adapter = :test

RSpec.describe "SignUpProcesses", type: :system do
  before do
    driven_by(:rack_test)
  end

  it "should require the user to sign up and successfully\
 sign up" do

    visit root_path

    click_on 'Sign up'

    within "#new_user" do
      fill_in "user_email", with: 'test@test.com'
      fill_in "user_password", with: 'password123'
      fill_in "user_password_confirmation", with: 'passwo\
rd123'
    end

    click_button "Sign Up"

    expect(current_path).to eql(new_accounts_path)
```

```
    within "#new_account" do
      fill_in "account_name", with: "Test Co"
    end

    expect do
      click_button "Save"
      expect(ActionMailer::Base.deliveries.last.to).to eq\
 ["test@test.com"]
      expect(current_path).to eql(root_path)
    end
...
```

A few things that have changed:

- `include ActiveJob::TestHelper` and `ActiveJob::Base.queue_adapter = :test` have been added to allow and use ActiveJob based helpers and methods properly.
- The `click_button` action was moved inside of an expect block. This is done to allow `ActionMailer::Base.deliveries` to be used and inspected for the spec's assertion.

The remainder of the system spec will remain the same.

8.2.2 Email Recap

As we have gone over a mailer in-depth and a mailer based on teams in a previous chapter, we are going to breeze through the setup of the `email_recap` jobs and mailer. We'll stop along the way and go over anything that hasn't seemed covered in the book so far.

We will start with a job similar to the reminder's `FindTeamsJob`:

```
module Recaps
  class FindTeamsJob < ApplicationJob
    def perform(*_args)
      teams.each do |team|
        Recaps::EmailUserOnTeamJob.perform_later(team)
      end
    end

    private

    def teams
      @teams ||=
        Team
        .includes(:days)
        .where(
          has_recap: true,
          recap_time: Time.at(last_fifteen_marker).utc
        )
        .where('days_of_the_week_memberships.day = ?', cu\
rrent_day)
        .references(:days_of_the_week_memberships)
    end

    def current_day
      DaysOfTheWeekMembership.days[Time.now.utc.strftime(\
'%A').downcase.to_sym]
    end

    def last_fifteen_marker
      Time.now.utc.to_i - (Time.now.utc.to_i % 15.minutes)
    end
  end
end
```

This class looks very much like the reminders' job. However, this job is looking for teams with `has_recap` set to true and a `recap_-`

`time`. For each team that is found, it sends the `team` into another job, which will get the appropriate users.

That job is much like the reminder's `EmailUserOnTeamJob` as well:

```
module Recaps
  class EmailUserOnTeamJob < ApplicationJob
    def perform(team, date = Date.today.iso8601)
      standups = team.users.flat_map do |u|
        u
          .standups
          .where(standup_date: date)
      end

      return if standups.empty?
      team.users.each do |user|
        EmailRecapMailer.recap_email(user, team, standups\
).deliver_later
      end
    end
  end
end
```

What is new in the `recap` version of this job is that it looks up a user's standups based on the list of the team's users. The date is an optional argument and uses the current date's iso format if the argument is not passed into the method's invocation. In the case of the previous `FindTeamsJob`, it is not given, and the standups' lookup will try and find standups based on the current date. Otherwise, the method will run through users on the team and invoke the `EmailRecapMailer`.

The mailer is also very similar:

```
class EmailRecapMailer < ApplicationMailer

  def recap_email(user, team, standups)
    @user = user
    @team = team
    @standups = standups

    make_bootstrap_mail(to: @user.email, subject: "#{team\
.name} Standups Recap!")
  end
end
```

The main difference here is the lack of the reply-to field being set(this email does not need replies managed) and the standups as a `@standups` instance variable to be used in the template.

To be able to use web template looking Standup lists we will want to add the following CSS to the `boostrap-email.scss` file:

```
//= @import bootstrap-email;

.box.box-primary {
  border-top-color: #3c8dbc;
}

.box {
  position: relative; border-radius: 3px;
  background: #ffffff;
  border-top: 3px solid #d2d6de;
  margin: 20px 5px;
  box-shadow: 0 1px 1px rgba(0, 0, 0, 0.1);
}

.box-header.with-border {
  border-bottom: 1px solid #f4f4f4;
}
```

```
.box-header {
  color: #444; display: block;
  padding: 10px; position: relative;
}

.box-header .box-title {
  display: inline-block; font-size: 18px; margin: 0; line\
-height: 1;
}

.box-body {
  border-top-left-radius: 0; border-top-right-radius: 0;
  border-bottom-right-radius: 3px;
  border-bottom-left-radius: 3px;
  padding: 10px;
}

.todo-list {
  margin: 0;
  padding: 0;
  list-style: none;
  overflow: auto;
}

.todo-list > li {
  border-radius: 2px;
  padding: 10px;
  background: #f4f4f4;
  margin-bottom: 2px;
  border-left: 2px solid #e6e7e8;
  color: #444;
}
```

Speaking of the template:

```
<div class="container">
  <h1 class="text-center mb-4">
    Standup App
  </h1>
  <div class="card mb-4">
    <div class="card-body">
      <h3 class="text-center mb-4"><%= @team.name%> Stand\
ups Recap! </h3>

        <% if @standups.empty? %>
          <p class="mb-4">
            No standups are available to display
          </p>
        <% else %>
          <% @standups.each_slice(3) do |standups| %>
            <div class="row">
              <% standups.each do |standup| %>
                <div class="col-4">
                  <div class="content-block">
                    <div class="box box-primary">
                      <div class="box-header with-border">
                        <h3 class="box-title">
                          <%= link_to standup.user.name, \
standups_account_user_url(standup.user) %>
                        </h3>
                      </div>
                      <div class="box-body">
                        <h4> Dids</h4>
                        <ul class="todo-list">
                          <% standup.dids.each do |did| %>
                            <li><%= did.title %></li>
                          <% end %>
                        </ul>
                        <h4> Todos</h4>
                        <ul class="todo-list">
```

```
                        <% standup.todos.each do |todo|\
 %>
                          <li><%= todo.title %></li>
                        <% end %>
                      </ul>
                      <h4> Blockers</h4>
                      <ul class="todo-list">
                        <% standup.blockers.each do |bl\
ocker| %>
                          <li><%= blocker.title %></li>
                        <% end %>
                      </ul>
                    </div>
                  </div>
                </div>
              </div>
              <% if standups.size < 3 %>
                <% (3 - standups.size).times.each do %>
                  <div class="col-4"></div>
                <% end %>
              <% end %>
            <% end %>
            </div>
          <% end %>
        <% end %>
      <%= link_to "View Your Standups", activity_mine_url\
(), {class: "btn btn-primary btn-lg mx-auto mt-2", style:\
 "width:95%"} %>
    </div>
  </div>
</div>
```

This template very closely matches other standup listing templates. However, as Bootstrap grids are unavailable here, the Bootstrap Email gem's grid is used to achieve a similar effect. In addition, due to email templates ultimately using tables, on the last slice of

three, any difference between the standup count and three will be filled with extra `empty columns` cells to ensure everything lines up correctly.

No new code is complete without its corresponding tests. In this case, the specs also closely match the `Reminder` versions.

Here is the `FindTeamsJob` spec:

```
require 'rails_helper'
include ActiveJob::TestHelper

RSpec.describe Recaps::FindTeamsJob do
  it 'matches with enqueued job' do
    ActiveJob::Base.queue_adapter = :test
    Recaps::FindTeamsJob.perform_later
    expect(Recaps::FindTeamsJob).to have_been_enqueued
  end

  it 'finds a team with recaps' do
    team= FactoryBot.create(
      :team,
      has_recap: true,
      recap_time: Time.at(
        Time.now.utc.to_i - (Time.now.utc.to_i % 15.minut\
es)
      ).utc
    )
    team.update(
      days: [DaysOfTheWeekMembership.new(
        team_id: team.id,
        day: Time.now.utc.strftime('%A').downcase
      )]
    )
    job = Recaps::FindTeamsJob.new
    job.perform_now
    expect(job.instance_variable_get(:@teams)).to eq([tea\
```

```
m])
  end
end
```

No magic here, just updating a couple of specs to use `recap` instead of `reminder`.

However, the `EmailUserOnTeamJob` spec has a little more changes than it's `Reminder`'s counterpart:

```
require 'rails_helper'
include ActiveJob::TestHelper

RSpec.describe Recaps::EmailUserOnTeamJob do
  let(:user) { FactoryBot.create(:user) }
  let(:team) do
    team = FactoryBot.create(
      :team,
      user_ids: [user.id],
      has_recap: true,
      recap_time: Time.at(
        Time.now.utc.to_i - (Time.now.utc.to_i % 15.minut\
es)
      ).utc
    )
    team.update(
      days: [DaysOfTheWeekMembership.new(
        team_id: team.id,
        day: Time.now.utc.strftime('%A').downcase
      )],
      users: [user]
    )
   team
  end
  let!(:standups) do
    [
```

```
      FactoryBot.create(
        :standup,
        standup_date: Date.today.iso8601,
        user_id: user.id
      )
    ]
  end

  it 'matches with enqueued job' do
    ActiveJob::Base.queue_adapter = :test
    Recaps::EmailUserOnTeamJob.perform_later
    expect(Recaps::EmailUserOnTeamJob).to have_been_enque\
ued
  end

  it 'enqueues a mailer based job' do
    ActiveJob::Base.queue_adapter = :test
    job = Recaps::EmailUserOnTeamJob.new(team, Date.today\
.iso8601)
    expect { job.perform_now }
      .to have_enqueued_job.on_queue('mailers')
  end

  it 'recap email is sent' do
    expect do
      perform_enqueued_jobs do
        EmailRecapMailer.recap_email(user, team, standups\
).deliver_later
      end
    end.to change { ActionMailer::Base.deliveries.size }.\
by(1)
  end
end
```

For the spec to work correctly, we must ensure standups under the correct user exist. Additionally, the user will need to be the team's

user. After getting past those changes, the spec is testing the job's enqueuing, queuing onto the mailer's queue, and the mailer gets delivered successfully.

Lastly, we will need to add a new hash to the `jobs_hash`. This one will be called `recap` and only have one difference. You will simply need to use the new `Recaps::FindTeamsJob` class name:

```
jobs_hash = {
  'reminder' => {
    'class' => 'Reminders::FindTeamsJob',
    'cron'  => '0,15,30,45 * * * *',
    'active_job' => true
  },
  'recap' => {
    'class' => 'Recaps::FindTeamsJob',
    'cron'  => '0,15,30,45 * * * *',
    'active_job' => true
  }
}

...
```

There you have it...a quick run of the test suite should show all tests passing and nearly 100% test coverage!

```
rspec
.....................................................\
.....................................................\
..............................

Finished in 18.02 seconds (files took 6.06 seconds to loa\
d)
147 examples, 0 failures

Coverage report generated for RSpec to standup_app/covera\
ge. 415 / 435 LOC (95.4%) covered.
```

8.2.3 Mail in Production

While this handles the development and testing side of things, there are a few more pieces to get into a production environment. So, while we will go a bit further with production differences and changes in a later deployment chapter, we will go over the production email to set up the following section regarding receiving mail into the application.

First, you will need to choose a transactional email provider. A few providers are available to use with the library for parsing incoming mail: Sendgrid, Cloudmailin, Mandrill, Mailgun, Postmark, Sparkpost, SES (amazon).

Sendgrid has an account that allows 100 outgoing emails for free a month, which should suffice to get a new SaaS up and running.

1. Sign up for a Sendgrid account Here[1]
2. Once signed up, add a domain name to the sending email address. For example `app.buildasaasappinrails.com` through this link[2].
3. Set up DNS records according to Sendgrid's domain name interface.
4. Add the following seeings to your production environment configuration file, `config/environments/production.rb`:

[1] https://signup.sendgrid.com/
[2] https://app.sendgrid.com/app/domains

```
config.action_mailer.delivery_method = :smtp
config.action_mailer.perform_deliveries = true
ActionMailer::Base.smtp_settings = {
  user_name: '**your username**',
  password: '**your password**',
  domain: '**yourdomainname**',
  address: 'smtp.sendgrid.net',
  port: 587,
  authentication: :plain
}
```

You can also add the same settings below `config.action_mailer.delivery_method = :letter_opener` in your development file. Then you could either comment out the above or the `:letter_opener` line to change which local delivery mechanism you will be using. In particular, it may be a good time to comment out the `:letter_opener` line now to try and send the welcome email through the rails console, `WelcomeEmailMailer.welcome_email(User.first).deliver_now`. If you got all the settings correct, you should see an email delivered to your inbox(assuming your email address is in `User.first.email`)!

8.3 Receiving mail or replies

Another aspect of email within a SaaS application is receiving mail. While this is far less normal or used than sending, it can be a great way to make end users' responses to emails or action items quicker.

At a high level, there are a few different layers to this. The topmost layer is the email service, and in this book's case, Sendgrid. This service handles sending outbound emails and routing incoming emails to an address/domain name specified in their interface. Once the email is routed, it will be redirected to your application's route and processor file. This file will be responsible for parsing the

incoming email address and using logic to decide what to do with it.

In the case of the Standup App we are building, we can have directions to respond to an Email Reminder to create a new standup right from their email response! Some of the tools we will be using are Sendgrid's email routing service and ngrok, an HTTP tunneling service. `ngrok` is very useful for a few solutions throughout this book. First, it allows you to have a web-accessible URL, which tunnels(connects) to ports within your local machine. You will have a `http://somesubdomain.ngrok.com` that will forward to your computer and a specific port specified when you start `ngrok`. This allows you to test with external services such as Sendgrid, Stripe(later), Github(later), and more!

Let's get started with some setup:

- Download and use ngrok[3]
  - Download ngrok[4]
  - Unzip and move the executable file to where you would like.
  - In *nix OS's, open ngrok with: `path/to/ngrok HTTP start 3000`. This will depend on the port you use for your Rails server. ngrok will now fire up a tunnel service with a randomly generated URL.
  - Add `config.hosts << "yoursubdomain.ngrok.io"` to `config/environments/development.rb`
  - Optionally, if you upgrade to a paid version of `ngrok`, you can set a subdomain so you do not have to change your settings elsewhere every time you restart `ngrok`.
- Follow your domain's DNS provider's instructions for adding an MX record for the subdomain or domain name you have chosen to use for inbound email. For example, the MX record will be `mx.sendgrid.net`, with a priority of `10`.

[3] https://ngrok.com
[4] https://ngrok.com

Before we head to the next part, we must create and store a password for incoming mail. You can use any method you like to make a password, and if you just can't think of anyway, you can run `SecureRandom.alphanumeric` in the Rails console.

Now, to store this password you just came up with, we will need to introduce a new small but mighty feature, `Rails Credentials`. Since Rails 5.2, Rails has provided a built-in way for storing secure credentials that are encrypted.

To start using Rails Credentials, you will want to fire off the edit command:

```
EDITOR="code --wait" rails credentials:edit
```

The editor I am using in this example is VSCode. However, you can use any editor you like and its command line caller. Also, `--wait` is used to ensure that the editor will wait for Rails to finish decrypting the file before opening it.

Now that we have the encrypted file open, add the following lines:

```
action_mailbox:
  ingress_password: *password you just made*
```

Save the file and close the tab/editor to ensure Rails saves and encrypts the file. You will notice when Rails first opened the file, there was some output text in your terminal. Which includes the master key, DO NOT LOSE THIS KEY. It will need to be used as the only text in a `config/master.key` to be able to encrypt and decrypt the credentials.

Now, with that out of the way, there is a little bit more setup we need to do for Action Mailbox.

First, two commands to get Rails and some tables set up for inbound email:

```
bin/rails action_mailbox:install
bin/rails db:migrate
```

Lastly, add the following line to the `config/application.rb` file to let Rails know we're using Sendgrid:

```
config.action_mailbox.ingress = :sendgrid
```

Ok, back to Sendgrid's setup:

- In Sendgrid, go to the Inbound Parse[5] link under settings to create the email route to send the email. Enter the following settings:
- Enter `app.yourdomain.com` for the domain field. You can optionally choose an additional subdomain to receive emails within the Inbound Parse.
- Enter `http://actionmailbox:THEPASSWORDFROMBEFORE@yoursubdomain.ngrok.` `mailbox/sendgrid/inbound_emails` as the destination URL.
- Check `POST the raw, full MIME message`, before pressing the `submit` button.

Ok, we're getting nearly done with the setup here. Next, we will need to set up the application's inbound email route and run a command to generate that mailbox class. First, in the `application_mailbox.rb` file created just a few commands ago, you will add `routing /standups\.[0-9a-fA-F]{8}\-[0-9a-fA-F]{4}\-[0-9a-fA-F]{4}\-[0-9a-fA-F]{4}\` `=> :standups`. This says when an incoming mail matches that email address syntax, it will be routed to the `standups` mailbox class for processing.

After that, to create the mailbox file, you will just need to run a Rails generator:

[5] https://app.sendgrid.com/settings/parse

```
bin/rails generate mailbox standups
```

A few changes will be needed to allow Action Mailbox and its files to capture text from incoming replies.

First, we will update the `EmailReminderMailer` to create a unique reply-to address and include that email address as part of the outgoing email:

```
class EmailReminderMailer < ApplicationMailer
  def reminder_email(user, team)
    @user = user
    @team = team
    reply_to = "'Standups App' <standups.#{@user.id}@app.\
yourdomain.com>"
    make_bootstrap_mail(
      to: @user.email,
      subject: "#{team.name} Standup Reminder!",
      reply_to: reply_to
    )
  end
end
```

Here we are adding the `reply_to` string so that when someone replies to a reminder email, the reply will be routed through Sendgrid, to Action Mailbox, and then finally by the Standup Mailbox class.

Next, we will update the mailer template to have `##- Please type your reply above this line -##` and some text letting the email recipient know they can add a standup by replying:

```
<div class="container">
  <span class="text-secondary text-center" style="font-si\
ze:8px">##- Please type your reply above this line -##</s\
pan>
  <h1 class="text-center mb-4">
    Standup App
  </h1>
  <div class="card mb-4">
    <div class="card-body">
      <h3 class="text-center mb-4"><%= @team.name%> Remin\
der! </h3>
      <p class="mb-4">
        Just wanted to remind you to add your standup for
        the team: <%= @team.name %>
      </p>
      <%= link_to "Add Your Standup", new_standup_url(), \
{class: "btn btn-primary btn-lg mx-auto mt-2", style: "wi\
dth:95%"} %>
    </div>
  </div>
</div>
```

Lastly, we will need to add an extra column to the `Standups` table to track the `Message-ID` coming from the Mailgun routed emails. As you can not count on an email service to provide "just once" delivery, we will need to track these unique IDs ourselves on the Standup table.

```
rails g migration AddMessageIdToStandups message_id
```

Next, before the end of the newly created migrations `change` method, you will want to add `add_index :standups, :message_id`. This index will allow quick lookups as the `Standups` table grows. Then, finally, migrate the actual change:

```
rails db:migrate
```

Lastly, to make sure sidekiq picks up the queues that Rails uses for Action Mailbox, we will need to adjust the `worker` in the `Procfile.dev` to:

```
worker: bundle exec sidekiq -q default -q mailers -q acti\
on_mailbox_routing -q active_storage_analysis
```

With those changes out of the way we can now add the new `StandupsMailbox` class that will parse the incoming email:

```
class StandupsMailbox < ApplicationMailbox
  TASK_TYPE_HASH = {
    '[d]' => 'Did',
    '[t]' => 'Todo',
    '[b]' => 'Blocker'
  }

  def process
    # Get a user id from reply-to or bail
    reply_user = mail.to.first&.split('<')&.last&.split('\
@')&.first&.split('.')&.last
    return if reply_user.blank?

    # Find a user by the id or bail
    user = User.find_by(id: reply_user)
    return if user.nil?

    # Bail if standup with incoming message-id exists
    return if Standup.exists?(message_id: inbound_email.m\
essage_id)

    # Bail if a standup for today exists
    today = Date.today.iso8601
```

```
    return if Standup.exists?(standup_date: today)

    # Get content or bail
    safe_body = Rails::Html::WhiteListSanitizer.new.sanit\
ize(mail_body)
    tasks_from_body = safe_body.scan(/(\[[dtb]{1}\].*)$/)
    return if tasks_from_body.blank? || tasks_from_body.e\
mpty?

    build_and_create_standup(
      user: user,
      tasks: tasks_from_body,
      date: today,
      message_id: inbound_email.message_id
    )
  end

  private

  def build_and_create_standup(user:, tasks:, date:, mess\
age_id:)
    standup = Standup.new(
      user_id: user.id,
      standup_date: date,
      message_id: message_id
    )

    tasks.each do |task|
      task_type, task_body = task.first.scan(/(\[[dtb]\])\
(.*)$/).flatten
      standup.tasks << Task.new(type: TASK_TYPE_HASH[task\
_type], title: task_body)
    end

    standup.save
```

```
  end

  def mail_body
    @mail_body ||= begin
      if mail.multipart?
        mail.parts[0].body.decoded
      else
        mail.decoded
      end
    end
  end
end
```

The class is relatively simple, but let's go over it section by section:

```
class StandupsMailbox < ApplicationMailbox
  TASK_TYPE_HASH = {
    '[d]' => 'Did',
    '[t]' => 'Todo',
    '[b]' => 'Blocker'
  }

  ...
```

Here we are setting up the `StandupsMailbox` to inherit from the `ApplicationMailbox` class, which handles the routing. Additionally, we are creating a hash to later use in the text content to `Task` type conversion.

```
...

def process
  # Get a user id from reply-to or bail
  reply_user = mail.to.first&.split('<')&.last&.split('@'\
)&.first&.split('.')&.last
  return if reply_user.blank?

  # Find a user by the id or bail
  user = User.find_by(id: reply_user)
  return if user.nil?

  # Bail if standup with incoming message-id exists
  return if Standup.exists?(message_id: inbound_email.mes\
sage_id)

  # Bail if a standup for today exists
  today = Date.today.iso8601
  return if Standup.exists?(standup_date: today)

  # Get content or bail
  safe_body = Rails::Html::WhiteListSanitizer.new.sanitiz\
e(mail_body)
  tasks_from_body = safe_body.scan(/(\[[dtb]{1}\].*)$/)
  return if tasks_from_body.blank? || tasks_from_body.emp\
ty?

...
```

Here we are grabbing some information used in the parsing and giving the `process` method chances to exit early if the incoming email is insufficient for processing and `Standup` creation. In the first section, the incoming email address is parsed to find the user's `id`. That string is then used to find a User. If there is no user, the method returns without adding a Standup.

The following line will exit the method early if there is already a standup with the current Message-ID. Again, this is guarding against email providers, not guaranteeing "just once" delivery. That is followed up by generating a variable for the current date and ensuring there is no standup with the `current_user` and `current_date`.

To get the mail's content, we will want to do two things, make sure we get the correct body from the mail object, as email clients can send both an HTML and plain text version. Then we will want to make sure we only grab the content above the `##- Please type your reply above this line -##` string by using the `split` method (which splits a string into as many parts based on the separator specified in the argument):

```
def mail_body
    @mail_body ||= begin
      body = if mail.multipart?
        mail.parts[0].body.decoded
      else
        mail.decoded
      end
      body.split('##- Please type your reply above this l\
ine -##').first
    end
  end
```

Finally, the actual email content is parsed with a regular expression. Regular Expression is a programming language that allows you to pattern match on a string and even capture parts of the pattern matching. This particular pattern(which you can get a more thorough syntax explanation here[6]) searches for lines that begin with `[d]`, `[t]` or `[r]`. If those are present, it captures the content to the end of the line. The `.scan` method on the content's body allows it

[6] https://regex101.com/r/Bg9EIy/2

to catch all occurrences of the above pattern. The' process' method exits if the scan's output is empty.

```
  build_and_create_standup(
      user: user,
      tasks: tasks_from_body,
      date: today,
      message_id: email.headers["Message-ID"]
    )
  end

  private

  def build_and_create_standup(user:, tasks:, date:, mess\
age_id:)
    standup = Standup.new(
      user_id: user.id,
      standup_date: date,
      message_id: message_id
    )

    tasks.each do |task|
      task_type, task_body = task.first.scan(/(\[[dtb]\])\
(.*)$/).flatten
      standup.tasks << Task.new(type: TASK_TYPE_HASH[task\
_type], title: task_body)
    end

    standup.save
  end
end
```

The last section is a culmination of the stored information so far to be saved into a new `Standup`. The `user`, `tasks_from_body`, `today`, and `message_id` are all passed into a method that will hand the

actual save. The `build_and_create_standup` method creates a new `Standup`, with the user's ID, date, and `message_id`. Once the object is created, the task's strings are iterated to build the `Task` with a type and assigned as child objects with the `<<` syntax. Finally, the new `Standup` object with children `Tasks` will be saved with `standup.save`

Lastly, you can test this all works if you reply back to an email(that was sent through Sendgrid SMTP and not letter_opener) with the following text:

```
[d] Did a thing
[d] And Another
[t] Something to do
[b] Something in the way. Some really long line about som\
ething or another
```

Testing this will require a new spec file with a few `it` blocks to try all the branches the `StandupMailbox` may encounter.

First, we will need to add one smile file to the `rails_helper.rb` file to ensure RSpec has access to the Rails Action Mailbox test helpers. Then, I'll add the new lines and a few lines above each so you know where to put the new stuff:

```
...

require 'rspec/rails'
require 'devise'
require 'action_mailbox/test_helper' # <-- new line

...

RSpec.configure do |config|
  config.include Devise::Test::IntegrationHelpers, type: \
:request
  config.include Devise::Test::IntegrationHelpers, type: \
```

```
:system
  config.include Devise::Test::ControllerHelpers, type: :\
view
  config.include ActionMailbox::TestHelper, type: :mailbo\
x # <-- new line

...
```

...and now the actual spec file itself:

```
require 'rails_helper'
include ActiveJob::TestHelper

RSpec.describe StandupsMailbox, type: :mailbox do
  let(:user) { FactoryBot.create(:user) }
  let(:to_email) { "standups.#{user.id}@app.buildasaasapp\
inrails.com" }
  let(:body) do
    %Q(
      [d] Did a thing
      [d] And Another
      [t] Something to do
      [b] Something in the way. Some really long line abo\
ut something or another

      ##- Please type your reply above this line -##
    )
  end

  subject do
    receive_inbound_email_from_mail(
      from: user.email,
      to: to_email,
      subject: 'Re: Reminder',
      body: body
```

```
    )
  end

  before do
    ActiveJob::Base.queue_adapter = :test
  end

  it 'saves standup record' do
    expect { subject }.to change(Standup, :count).by(1)
  end

  it 'saves task records' do
    expect { subject }.to change(Task, :count).by(4)
  end

  context 'fails on bad to email' do
    let(:to_email) {"standups@app.buildasaasappinrails.co\
m"}

    it 'does not save standup record' do
      expect { subject rescue nil }.to_not change(Standup\
, :count)
    end
  end

  context 'fails on no user' do
    let(:to_email) {"standups.9a859af8-30ca-4473-b073-471\
05352d936@app.buildasaasappinrails.com"}

    it 'does not save standup record' do
      expect { subject rescue nil }.to_not change(Standup\
, :count)
    end
  end
```

```
  context 'fails on useless body' do
    let(:body) { 'asdj;oisaduaskd' }

    it 'does not save standup record' do
      expect { subject rescue nil }.to_not change(Standup\
, :count)
    end
  end

  context 'fails on empty body' do
    let(:body) { '' }

    it 'does not save standup record' do
      expect { subject rescue nil }.to_not change(Standup\
, :count)
    end
  end
end
```

While long and containing six examples, this spec is actually pretty straightforward. It first tests the happy path where everything is set up and working. Then checks each failing path that doesn't create a standup, as those paths appear in the `StandupsMailbox`'s `.process` method.

A quick run of the whole rspec suite should show no failing tests and nearly perfect test/code coverage:

```
rspec
.....................................................\
.....................................................\
.....................................

Finished in 18.34 seconds (files took 6.23 seconds to loa\
d)
153 examples, 0 failures

Coverage report generated for RSpec to standup_app/covera\
ge. 443 / 464 LOC (95.47%) covered.
```

Lastly, let's merge the branch we have been running with all chapter into master:

```
git add .
git commit -am "Add Welcome Mailer, Recap Mailer and Acti\
onMailbox"
git checkout master
git merge welcome-mailer
git push
```

8.4 What did you learn in this chapter?

- What a mailer is and what a Rails generated ActionMailer comes with.
- You created a single use mailer called `WelcomeMailer`
- You created a mailer with it's supporting files manually.
- You learned how to parse incoming mail through `ActionMailbox`.

8.5 Exercises

Exercise answers can be found with any package or bundle purchases of Build A SaaS App in Ruby on Rails 7[7]!

1. Since the reply input is being sanitized by Rails, it would be great to write a test that makes sure bad input is removed, and good input is left in.
2. Even, better, you may want to make sure that all incoming input is stripped of everything but it's string representation.
3. Verify through the console that you can generate a Recap email.

[7] https://buildasaasappinrails.com

9. HotWire

9.1 What is HotWire?

From the HotWire[1] website, you will find the following explanation:

> Hotwire is an alternative approach to building modern web applications without using much JavaScript by sending HTML instead of JSON over the wire. This makes for fast first-load pages, keeps template rendering on the server, and allows for a more straightforward, more productive development experience in any programming language without sacrificing any speed or responsiveness associated with a traditional single-page application.

But what does that mean? In simpler terms, it is a set of tools (Turbo, StimulusJS, and the yet-to-be-released Strada) that allows you to write interactive applications, which may include real-time updates, with little to no custom JavaScript.

9.1.1 What is Turbo?

Turbo is the primary tool that is going to drive your interactive application. However, there are a few components within Turbo that deal with those different parts of an interactive site:

- Turbo Drive: This is mostly what Turbolinks of past Rails versions have turned into. It keeps all the same functionality of

[1] https://hotwired.dev

partial page updates, quick loading, and more while updating it to support all of the HotWire needs.
- Turbo Frames: These are custom components you drop into your HTML, allowing you to easily manage partial page navigation. This even includes a simple way to lazy load content!
- Turbo Stream: This is where all of the real-time magic happens! This tool (along with some help from the `turbo-rails`) gem allows your application to come alive by updating your application's HTML in real-time. It utilizes websockets and HTML rendering to send updates around the app in real-time!

9.2 Using HotWire in your application

Like usual, let's not get too carried away in theory and get right to putting a Rails library to use.

To get started, let's create a new branch:

```
git checkout -b hotwire-standups
```

The next step to install HotWire is...nothing. There is nothing left to do, as HotWire was installed and enabled when the Rails application was created a few chapters back. It has installed the `turbo-rails` gem and added the lines needed to all the required Javascript files.

9.3 Streaming Updates

As we get into a bit of theory here, the streaming workflow will work at a high level:

- An update happens on a model object. In our case, it will be the `Standup` object.
- Once the update is finished and "committed" to the database, a bit of code will run to prompt Rails to send an update.
- The update is a bit of HTML passed over a WebSocket. This will correspond to a few other HTML tags added to some views.
- Once the JavaScript receives the WebSocket market, it will look at some of the data, such as the IDs and "turbo event type," to determine where it goes and how it should be modified into the DOM.
- The markup appears updated to the user using the application.

The first thing we will do to introduce real-time updates in the application is to add the needed markup for updates and creations for the "mine" view:

```
<div class="flex p-5" aria-label="Breadcrumb">
  <ol class="inline-flex items-center space-x-1 md:space-\
x-3">
    <li class="inline-flex items-center">
      <h2 href="#" class="inline-flex items-center font-m\
edium text-gray-700 hover:text-gray-900">
        My Activity
      </h2>
    </li>
  </ol>
</div>

<div class="flex justify-center">
  <div class="p-4">
    <%= turbo_stream_from current_user, "standups" %>
      <div class="grid grid-cols-3 auto-cols-auto gap-4" \
id="<%= dom_id(current_user, 'standups') %>">
```

```
        <%= render @standups %>
      </div>
  </div>
</div>
```

9.3.1 Streaming Standup Create

Specifically adding `turbo_stream_from current_user, "standups"` to add a TurboStream to the page allows Turbo to know it accepts updates from WebSockets. This also adds a signed token into the markup to prevent misusing Websockets to gain access to data the end user may not have access to.

`turbo_stream_from` accepts multiple arguments that will be concatenated and used to create the signed token. This will be replicated on the front end to ensure only streams that subscribe to that user and the string "standups" can listen to the stream.

Next, `id="<%= dom_id(current_user, 'standups') %>"` adds a DOM ID to the standup list container. `dom_id` accepts two arguments: A resource that will have an ID and a string or symbol to use as a "prefix". This will allow us to target this element from the broadcast.

Next up is adding the functionality to the model. However, knowing that this will grow in size over the 3 modifications that can be made to a Standup (create, update and delete) and the possibility that it will be needed for multiple different steams which may use other arguments or attributes, we will create a separate file. Luckily, Rails has a way to easily included model code into a model from another plain ruby file. It is called a `concern` and has another module, `ActiveSupport::Concern`, to add in to make the syntax even simpler:

```
module StandupBroadcasts
  extend ActiveSupport::Concern

  included do
    scope :users_standups, lambda { |user|
      user
        .standups
        .includes(:dids, :todos, :blockers)
        .references(:tasks)
        .order('standup_date DESC')
    }

    after_create_commit do
      broadcast_update_to(
        [user, :standups],
        target: dom_id(user, :standups),
        html: render(
          partial: 'standups/standup', collection: Standu\
p.users_standups(user), locals: { caller: 'activity_mine'\
, user: user }
        )
      )
    end
  end
end
```

First, the `extend ActiveSupport::Concern` allows us to use the `included do` block to add code to this file as if it were right inside the module itself. Next up is the use of a "named scope". As we will need a list of a user's standup, just as we were using inside the `ActivityController`, we need an easy way to get this within the model. A named scope is the answer to this. It also allows you to pass an argument in, which in this case, will be the user we want the standups for. Otherwise, this method chain looks just like we had used in the controller. Lastly, we use the `after_create_commit` hook

to "do a thing" after "the new standup is committed" to the database. Within that block, we have access to the broadcast methods and, in this case, `broadcast_update_to`.

There are a handful of broadcast types:

- `append` to add elements after a specified ID
- `prepend` to add elements before a specified ID
- `replace` to replace the contents of a `specified` ID
- `update`, like replace, but replaces the content inside the element, thus leaving all existing attributes and associated events on the element.

`broadcast_update_to`, like most broadcast methods, takes a few arguments. The first argument is the broadcast stream identifier, which is what you want to match up with the TurboStream tag you created earlier. Here though, it is an array `[user, :standups]`. Next is the target, which is the HTML ID you are looking to broadcast a stream change to, and again, we are trying to match the ID we created in the view. Lastly is the content that is being sent to the front end; in this case, it is `html` generated by the Rails renderer.

If you were to add this code now, you would notice that `dom_id` and `render` are not currently available as they are in classes that are usually only automatically included in controllers and views. However, to add them to every model, we can add the following lines to the `ApplicationRecord`:

```
delegate :render, to: :ApplicationController
delegate :dom_id, to: 'ActionView::RecordIdentifier'
```

This tells `ApplicationRecord` that it can add these methods and the classes they come from. Since `ApplicationRecord` is the parent class to all models, it will now add these methods to all models!

Now that those methods have been added, we can use `dom_id` to easily match the ID for the target argument and then `render` to

generate the HTML. The render function takes a few arguments itself:
It takes a `partial` to specify what view file to use.
`Collection` passes the array of objects included in each partial rendering.
`Locals` allows us to pass any more extra data to the partial to be used. `caller` will allow us to specify the content for the standup's header already being used in the `StandupHelper`.
However, as of right now, it depends on the normal rendering cycle to determine the controller and action calling the partial, which is not present when rendering from the model.

To make use of the `locals` variable, make the following change in the `_standup` partial:

```
<h3 class="font-bold leading-tight text-gray-900 text-xl"\
><%= standup_header standup%></h3>
```

to

```
<h3 class="font-bold leading-tight text-gray-900 text-xl"\
><%== standup_header standup, local_assigns[:caller]%></h\
3>
```

Then, update the `StandupHelper` to check for a caller first and fall back to the controller and action next:

```
module StandupHelper
  def standup_header(standup, caller)
    caller ||= [controller_path, action_name].join("_")
    case caller
    when "activity_mine", "users/standups_index"
      standup.standup_date
    when "teams/standups_index", "teams_show"
      link_to standup.user.name, standups_account_user_pa\
th(standup.user), data: { turbo_frame: '_top' }
    else
      "..."
    end
  end
end
```

By passing in the caller, which will be populated or `nil`, `caller ||=` will only set caller if it is not nil when passed into the helper method. The other change, `data: { turbo_frame: '_top' }`, will be used later in the chapter when implementing Turbo Frames.

Lastly, we can write a quick system test to check if the Standup shows up on the page when the record is created:

```
require 'rails_helper'

RSpec.describe "standup/new", type: :system do
  login_admin

  context 'show a new standup' do

    it 'on the activity#mine page when record is created'\
, js: true do
      visit activity_mine_path
      expect(page).to have_no_css('.w-96')
      expect(page).to have_selector('turbo-cable-stream-s\
ource', visible: false)
```

```
      FactoryBot.create(:standup, standup_date: Date.toda\
y, user_id: @admin.id)

      expect(page).to have_css('.w-96')
    end
  end
end
```

This is all standard RSpec syntax with just two callouts. First, `js: true` allows the test to run with a web-based browser to execute javascript, which is needed to test Turbo functionality. Second, by using the selector `expect(page).to have_selector('turbo-cable-stream-source', visible: false)`, it tells RSpec to wait for the turbo HTML element to exist, which makes sure the Standup is created after a WebSocket connection is established. Otherwise, the page tests that no standup element exists, then does exist after creating the Standup in the spec.

9.3.2 Streaming Standup Update

With all of the changes in the previous section and the inclusion of `dom_id` in the `_standups` partial, the only update needed to handle streaming updates is inside the `StandupBroadcasts` file.

```
module StandupBroadcasts
  extend ActiveSupport::Concern

  included do
    scope :users_standups, lambda { |user|
      user
        .standups
        .includes(:dids, :todos, :blockers)
        .references(:tasks)
```

```
        .order('standup_date DESC')
    }

    after_create_commit do
      broadcast_update_to(
        [user, :standups],
        target: dom_id(user, :standups),
        html: render(
          partial: 'standups/standup', collection: Standu\
p.users_standups(user), locals: { caller: 'activity_mine'\
, user: user }
        )
      )
    end

    after_update_commit do
      broadcast_replace_to [user, :standups], target: sel\
f, locals: { caller: 'activity_mine', user: user }
    end
  end
end
```

To handle sending updates after an update to the Standup record, we will use `after_update_commit`, fired after the record change is committed into the database. Once again, a broadcast is used, and there are a few differences, but due to inference and conventional defaults. The first is the target, which uses `self` in this case. In the case of being inside a model, self refers to the object that has just been updated. Thus, the broadcast will know it is a Standup, will be able to look for the default resource-based `dom_id`, and knows to use the `_standup` partial. Easy, right? Now the stream identifier and locals passed to the template are the same.

We can jump right to a spec now!

```
require 'rails_helper'

RSpec.describe "standup/edit", type: :system do
  login_admin

  context 'show an edited standup' do

    it 'on the activity#mine page when record is edited',\
 js: true do
      standup = FactoryBot.create(:standup, standup_date:\
 Date.today, user_id: @admin.id)

      visit activity_mine_path
      expect(page).to have_css('.w-96')
      expect(page).to have_content(Date.today.iso8601)
      expect(page).to have_selector('turbo-cable-stream-s\
ource', visible: false)

      standup.update(standup_date: Date.tomorrow.iso8601)

      expect(page).to have_css('.w-96')
      expect(page).to have_content(Date.tomorrow.iso8601)
    end
  end
end
```

This spec is very similar to the one for the new Standup, except it creates the Standup before visiting the page to ensure it is listed. Then the spec updates the date and expects the change is reflected on the page.

9.3.3 Streaming Standup Removal

Once again, all of the markup and supporting code are already written, so we will just need to add the callback block with the

broadcast inside of it:

```
module StandupBroadcasts
  extend ActiveSupport::Concern

  included do
    scope :users_standups, lambda { |user|
      user
        .standups
        .includes(:dids, :todos, :blockers)
        .references(:tasks)
        .order('standup_date DESC')
    }

    after_create_commit do
      broadcast_update_to(
        [user, :standups],
        target: dom_id(user, :standups),
        html: render(
          partial: 'standups/standup', collection: Standu\
p.users_standups(user), locals: { caller: 'activity_mine'\
, user: user }
        )
      )
    end

    after_update_commit do
      broadcast_replace_to [user, :standups], target: sel\
f, locals: { caller: 'activity_mine', user: user }
    end

    after_destroy_commit do
      broadcast_remove_to [user, :standups]
    end
  end
end
```

Here we use `after_destroy_commit` to fire a broadcast after the object is destroyed and then use `broadcast_remove_to` to broadcast the removal. This method is even shorter as `self` is inferred when arguments are omitted. Thus it knows it is looking for the `dom_id` for the recently destroyed Standup. We can jump right over to a destroy spec!

```
require 'rails_helper'

RSpec.describe "standup/destroy", type: :system do
  login_admin

  context 'show a destroyed standup' do

    it 'on the activity#mine page when record is destroye\
d', js: true do
      standup = FactoryBot.create(:standup, standup_date:\
 Date.today, user_id: @admin.id)

      visit activity_mine_path
      expect(page).to have_css('.w-96')
      expect(page).to have_content(Date.today.iso8601)
      expect(page).to have_selector('turbo-cable-stream-s\
ource', visible: false)

      standup.destroy

      expect(page).to have_no_css('.w-96')
      expect(page).to have_no_content(Date.tomorrow.iso86\
01)
    end
  end
end
```

Lastly, if you run the whole test suite, you may notice that your view spec is failing. This is because `current_user` is now in the

codepath for this view and needs to be stubbed in the test. Here is the updated spec:

```
require 'rails_helper'

RSpec.describe "activity/mine.html.erb", type: :view do
  before do
    allow(view).to receive(:current_date)
      .and_return(Date.today.strftime('%a %d %b %Y'))
      allow(view).to receive(:current_user)
      .and_return(FactoryBot.create(:user))
    assign(:standups, [FactoryBot.create(:standup)])
  end
  it "renders the word feed" do
    render template: 'activity/mine'
    expect(rendered).to match(/My Activity/)
  end
end
```

There you have it, you are now streaming broadcasts for the user's dashboard!

9.3.4 Streaming Standup Changes onto Team pages

After laying a bunch of groundwork to add real-time broadcasts to the `Activity#mine` page, adding it to other pages will be an exercise of making sure the templates at `turbo_stream` tags with relevant sources and making sure the model callbacks can correctly stream to those channels.

First, we can edit `app/views/teams/show.html.erb` to add a Turbo Stream and an ID to use in the target. Change...

```
<div class="grid grid-cols-3 auto-cols-auto gap-4">
```

to

```
<%= turbo_stream_from @team, Date.today.iso8601, "standup\
s" %>
<div class="grid grid-cols-3 auto-cols-auto gap-4" id="<%\
= "#{@team.id}_#{Date.today.iso8601}_standups" %>">
```

The changes to note here are that the identifier is a concatenation of the team, the current date (in ISO8601 format, as it is stored in the database), and the string standups. Second, as `dom_id` does not accept multiple identifiers, it is easy to simply use string concatenation to form the ID for the team's standups.

Now we're off to the `StandupBroadcasts` file to add a new named scope and a few more callbacks!

```
module StandupBroadcasts
  extend ActiveSupport::Concern

  included do
    scope :users_standups, lambda { |user|
      user
        .standups
        .includes(:dids, :todos, :blockers)
        .references(:tasks)
        .order('standup_date DESC')
    }

    scope :teams_standups_by_date, lambda { |team, date|
      team
        .users
        .flat_map { |u| u.standups.where(standup_date: da\
te).includes(:dids, :todos, :blockers).references(:tasks)\
```

```
 }
    }

    after_create_commit do
      broadcast_update_to(
        [user, :standups],
        target: dom_id(user, :standups),
        html: render(
          partial: 'standups/standup', collection: Standu\
p.users_standups(user), locals: { caller: 'activity_mine'\
, user: user }
        )
      )
    end

    after_update_commit do
      broadcast_replace_to [user, :standups], target: sel\
f, locals: { caller: 'activity_mine', user: user }
    end

    after_destroy_commit do
      broadcast_remove_to [user, :standups]
    end
  end
end
```

As you can see, the scope takes a team and a date to map over the users from the team and select a standup that matches the user id and the `standup_date`.

```
module StandupBroadcasts
  extend ActiveSupport::Concern

  included do
    scope :users_standups, lambda { |user|
      user
        .standups
        .includes(:dids, :todos, :blockers)
        .references(:tasks)
        .order('standup_date DESC')
    }

    scope :teams_standups_by_date, lambda { |team, date|
      team
        .users
        .flat_map { |u| u.standups.where(standup_date: da\
te).includes(:dids, :todos, :blockers).references(:tasks)\
 }
    }

    after_create_commit do
      broadcast_update_to(
        [user, :standups],
        target: dom_id(user, :standups),
        html: render(
          partial: 'standups/standup', collection: Standu\
p.users_standups(user), locals: { caller: 'activity_mine'\
, user: user }
        )
      )
      user.teams.each do |team|
        broadcast_update_to(
          [team, standup_date, :standups],
          target: "#{team.id}_#{standup_date}_standups",
          html: render(
```

```
            partial: 'standups/standup',
            collection: Standup.teams_standups_by_date(te\
am,
                                                       sta\
ndup_date),
            locals: { caller: 'teams_show', user: user }
          )
        )
      end
    end

    after_update_commit do
      broadcast_replace_to [user, :standups], target: sel\
f, locals: { caller: 'activity_mine', user: user }
    end

    after_destroy_commit do
      broadcast_remove_to [user, :standups]
    end
  end
end
```

In the `after_create_commit` hook, we are iterating over all the teams the user who just created a standup is on. From there, we broadcast to the identifier that matches what was used in the view and the element's ID matching the target parameter. Lastly, a partial is rendered with the usual `_standup` partial, a collection from the earlier named scope, and this time using the `caller` of `teams_show` to make sure the user's name shows in the standup's header.

The changes for updating and removing a standup are even simpler!

```
module StandupBroadcasts
  extend ActiveSupport::Concern

  included do
    scope :users_standups, lambda { |user|
      user
        .standups
        .includes(:dids, :todos, :blockers)
        .references(:tasks)
        .order('standup_date DESC')
    }

    scope :teams_standups_by_date, lambda { |team, date|
      team
        .users
        .flat_map { |u| u.standups.where(standup_date: da\
te).includes(:dids, :todos, :blockers).references(:tasks)\
 }
    }

    after_create_commit do
      broadcast_update_to(
        [user, :standups],
        target: dom_id(user, :standups),
        html: render(
          partial: 'standups/standup', collection: Standu\
p.users_standups(user), locals: { caller: 'activity_mine'\
, user: user }
        )
      )
      user.teams.each do |team|
        broadcast_update_to(
          [team, standup_date, :standups],
          target: "#{team.id}_#{standup_date}_standups",
          html: render(
```

```
            partial: 'standups/standup',
            collection: Standup.teams_standups_by_date(te\
am,
                                                       sta\
ndup_date),
            locals: { caller: 'teams_show', user: user }
          )
        )
      end
    end

    after_update_commit do
      broadcast_replace_to [user, :standups], target: sel\
f, locals: { caller: 'activity_mine', user: user }
      user.teams.each do |team|
        broadcast_replace_to [team, standup_date, :standu\
ps], target: self, locals: { caller: 'teams/standups_inde\
x', user: user }
      end

    end

    after_destroy_commit do
      broadcast_remove_to [user, :standups]
      user.teams.each do |team|
        broadcast_remove_to [team, standup_date, :standup\
s]
      end
    end
  end
end
```

Here the teams are iterated over, like in create. Then, the identifier and caller are updated to reflect the team page streaming location.

Lastly, because we are tracking updates here, we also want to make

sure that the `Tasks` will also cause the `Standup` broadcasts to trigger, such as when a `Did`, `Todo`, or `Blocker` is added or removed from the `Standup`. Since everything runs through a `TaskMembership`, which `belongs_to` a `:standup`, there is a simple parameter that can be passed to the `belongs_to` setting called `touch: true`. When one of those tasks is added, the standup will update its `updated_at` timestamp and cause any `after_update_commit` callbacks to run. Here is the change:

```
class TaskMembership < ApplicationRecord
  belongs_to :task
  belongs_to :standup, touch: true
end
```

Finally, we can write some tests to make sure this all works!

```
require 'rails_helper'

RSpec.describe "standup/new", type: :system do
  login_admin

  context 'show a new standup' do

    it 'on the activity#mine page when record is created'\
, js: true do
      visit activity_mine_path
      expect(page).to have_no_css('.w-96')
      expect(page).to have_selector('turbo-cable-stream-s\
ource', visible: false)

      FactoryBot.create(:standup, standup_date: Date.toda\
y, user_id: @admin.id)

      expect(page).to have_css('.w-96')
    end
```

```
    it 'on the teams#show page when record is created', j\
s: true do
      team = FactoryBot.create(:team)
      team.users << @admin
      visit team_path(team)

      expect(page).to have_no_content('Dids')
      expect(page).to have_selector('turbo-cable-stream-s\
ource', visible: false)

      FactoryBot.create(:standup, standup_date: Date.toda\
y, user_id: @admin.id)

      expect(page).to have_content(@admin.name)
    end
  end
end
```

The main change in this new second spec is that we want to make sure a team exists, has the current logged-in admin as a member, and that the standup is created under that user. From there, we check for `dids` and the admin's name to adjust for the different content on the `teams#show` page.

Here is how we would add a new edit spec:

```
require 'rails_helper'

RSpec.describe "standup/edit", type: :system do
  login_admin

  context 'show an edit standup' do

    it 'on the activity#mine page when record is updated'\
, js: true do
      standup = FactoryBot.create(:standup, standup_date:\
 Date.today, user_id: @admin.id)

      visit activity_mine_path
      expect(page).to have_css('.w-96')
      expect(page).to have_content(Date.today.iso8601)
      expect(page).to have_selector('turbo-cable-stream-s\
ource', visible: false)

      standup.update(standup_date: Date.tomorrow.iso8601)

      expect(page).to have_css('.w-96')
      expect(page).to have_content(Date.tomorrow.iso8601)
    end

    it 'on the teams#show page when record is updated', j\
s: true do
      team = FactoryBot.create(:team)
      team.users << @admin

      standup = FactoryBot.create(:standup, standup_date:\
 Date.today, user_id: @admin.id)

      visit team_path(team)

      expect(page).to have_content('Dids')
```

```
      expect(page).to have_selector('turbo-cable-stream-s\
ource', visible: false)

      standup.dids << Did.new(title: 'Hello there')

      expect(page).to have_css('.w-96')
      expect(page).to have_content('Hello there')
    end
  end
end
```

We are adding a new `Did` to ensure this triggers an update and that update shows up in real-time. Finally, the destroy spec:

```
require 'rails_helper'

RSpec.describe "standup/destroy", type: :system do
  login_admin

  context 'show a destroyed standup' do

    it 'on the activity#mine page when record is destroye\
d', js: true do
      standup = FactoryBot.create(:standup, standup_date:\
 Date.today, user_id: @admin.id)

      visit activity_mine_path
      expect(page).to have_css('.w-96')
      expect(page).to have_content(Date.today.iso8601)
      expect(page).to have_selector('turbo-cable-stream-s\
ource', visible: false)

      standup.destroy

      expect(page).to have_no_css('.w-96')
```

```
      expect(page).to have_no_content(Date.tomorrow.iso86\
01)
    end

    it 'on the teams#show page when record is destroyed',\
 js: true do
      team = FactoryBot.create(:team)
      team.users << @admin
      standup = FactoryBot.create(:standup, standup_date:\
 Date.today, user_id: @admin.id)

      visit team_path(team)

      expect(page).to have_content('Dids')
      expect(page).to have_selector('turbo-cable-stream-s\
ource', visible: false)

      standup.destroy

      expect(page).to have_no_content('Dids')
    end
  end
end
```

This spec uses changes from the past two specs to track that the destroyed standup is removed from the page.

Now, to add the team-based standup updates to the `teams/standups#index` page or the "standalone standup list for a team by date" page, the only change needed at this point will be a view change. In fact, the only difference from the `teams#show` page is the use of the `current_date` helper method instead of using `Date.today` in the stream identifier and element ID. So, in `app/views/teams/standups/index.html.erb` view, change:

```
<div class="grid grid-cols-3 auto-cols-auto gap-4">
```

to

```
<%= turbo_stream_from @team, current_date, "standups" %>
<div class="grid grid-cols-3 auto-cols-auto gap-4" id="<%\
= "#{@team.id}_#{current_date}_standups" %>">
```

It will use the same broadcast code, so we can jump to specs!

```
require 'rails_helper'

RSpec.describe "standup/new", type: :system do
  login_admin

  context 'show a new standup' do

    it 'on the activity#mine page when record is created'\
, js: true do
      visit activity_mine_path
      expect(page).to have_no_css('.w-96')
      check_turbo_and_sleep

      FactoryBot.create(:standup, standup_date: Date.toda\
y, user_id: @admin.id)

      expect(page).to have_css('.w-96')
    end

    it 'on the teams#show page when record is created', j\
s: true do
      team = FactoryBot.create(:team)
      team.users << @admin
      visit team_path(team)
```

```
      expect(page).to have_no_content('Dids')
      check_turbo_and_sleep

      FactoryBot.create(:standup, standup_date: Date.toda\
y, user_id: @admin.id)

      expect(page).to have_content(@admin.name)
    end

    it 'on the teams/standups#index page when record is c\
reated', js: true do
      team = FactoryBot.create(:team)
      team.users << @admin
      visit team_standups_path(team)

      expect(page).to have_no_content('Dids')
      check_turbo_and_sleep

      FactoryBot.create(:standup, standup_date: Date.toda\
y, user_id: @admin.id)

      expect(page).to have_content(@admin.name)
    end
  end
end
```

The spec is just like the second spec, except it is asserting on a different page in the app. However, you may have also noticed there is a replacement for the expectation of the Turbo tag, `check_turbo_and_sleep`. As the headless browser in the test is loading up the Javascript, markup, and connecting to websockets, it appears that while the tag may exist on the page, the WebSocket has not connected before the Standup is created in the test. Adding a small sleep gives the browser enough time to connect and not have a test that succeeds inconsistently.

Add the following method to the bottom of your `rails_helper.rb` file:

```
def check_turbo_and_sleep
  expect(page).to have_selector('turbo-cable-stream-sourc\
e', visible: false)
  sleep 0.25
end
```

Hopefully, in the future, there will be a longer-term solution for this problem of the Websocket not being ready in time.

On to the next two spec files:

```
require 'rails_helper'

RSpec.describe "standup/edit", type: :system do
  login_admin

  context 'show an edit standup' do

    it 'on the activity#mine page when record is updated'\
, js: true do
      standup = FactoryBot.create(:standup, standup_date:\
 Date.today, user_id: @admin.id)

      visit activity_mine_path
      expect(page).to have_css('.w-96')
      expect(page).to have_content(Date.today.iso8601)
      check_turbo_and_sleep

      standup.update(standup_date: Date.tomorrow.iso8601)

      expect(page).to have_css('.w-96')
      expect(page).to have_content(Date.tomorrow.iso8601)
    end
```

```
    it 'on the teams#show page when record is updated', j\
s: true do
      team = FactoryBot.create(:team)
      team.users << @admin

      standup = FactoryBot.create(:standup, standup_date:\
 Date.today, user_id: @admin.id)

      visit team_path(team)

      expect(page).to have_content('Dids')
      check_turbo_and_sleep

      standup.dids << Did.new(title: 'Hello there')

      expect(page).to have_css('.w-96')
      expect(page).to have_content('Hello there')
    end

    it 'on the teams/standups#index page when record is u\
pdated', js: true do
      team = FactoryBot.create(:team)
      team.users << @admin

      standup = FactoryBot.create(:standup, standup_date:\
 Date.today, user_id: @admin.id)

      visit team_standups_path(team)

      expect(page).to have_content('Dids')
      check_turbo_and_sleep

      standup.dids << Did.new(title: 'Hello there')
```

```
      expect(page).to have_css('.w-96')
      expect(page).to have_content('Hello there')
    end
  end
end
```

And

```
require 'rails_helper'

RSpec.describe "standup/destroy", type: :system do
  login_admin

  context 'show a destroyed standup' do

    it 'on the activity#mine page when record is destroye\
d', js: true do
      standup = FactoryBot.create(:standup, standup_date:\
 Date.today, user_id: @admin.id)

      visit activity_mine_path
      expect(page).to have_css('.w-96')
      expect(page).to have_content(Date.today.iso8601)
      check_turbo_and_sleep

      standup.destroy

      expect(page).to have_no_css('.w-96')
      expect(page).to have_no_content(Date.tomorrow.iso86\
01)
    end

    it 'on the teams#show page when record is destroyed',\
 js: true do
      team = FactoryBot.create(:team)
```

```
      team.users << @admin
      standup = FactoryBot.create(:standup, standup_date:\
 Date.today, user_id: @admin.id)

      visit team_path(team)

      expect(page).to have_content('Dids')
      check_turbo_and_sleep

      standup.destroy

      expect(page).to have_no_content('Dids')
    end

    it 'on the teams/standups#index page when record is d\
estroyed', js: true do
      team = FactoryBot.create(:team)
      team.users << @admin
      standup = FactoryBot.create(:standup, standup_date:\
 Date.today, user_id: @admin.id)

      visit team_standups_path(team)

      expect(page).to have_content('Dids')
      check_turbo_and_sleep

      standup.destroy

      expect(page).to have_no_content('Dids')
    end
  end
end
```

The last place we will broadcast changes will be the notification dropdown. This is the dropdown triggered in the navigation bar with the bell icon.

The first step we will take is to move the contents of the dropdown into a partial that will be later used in the broadcast methods.

```
<span class="text-xs text-gray-400"><%= notification_stan\
dups.count %> Notifications</span>
<div class="border border-gray-800"></div>
<% if notification_standups.count > 0 %>
  <% notification_standups.each do |standup| %>
    <a href="#" class="p-2 hover:bg-gray-800 text-white t\
ext-sm no-underline hover:no-underline block"><%= standup\
.user.name %> has a new Standup from <%= standup.standup_\
date %></a>
  <% end %>
<% end %>
```

To include this, you will need to update the `_header` partial with the new TurboStream tag and the call to the partial:

```
<header>
  <!--Nav-->
  <nav aria-label="menu nav" class="bg-gray-800 pt-2 md:p\
t-1 pb-1 px-1 mt-0 h-auto fixed w-full z-20 top-0">

    <div class="flex flex-wrap items-center">
      <div class="flex flex-shrink md:w-1/2 justify-cente\
r md:justify-start text-white">
        <a href="/" aria-label="Home">
          <span class="text-xl pl-2">Standup App</span>
        </a>
      </div>

      <div class="flex w-full pt-2 content-center justify\
-between md:w-1/2 md:justify-end">
        <ul class="list-reset flex justify-between flex-1\
 md:flex-none items-center">
```

```
          <li class="flex-1 md:flex-none md:mr-3">
            <a href="/t" class="inline-block py-2 px-4 te\
xt-white no-underline">Teams</a>
            <a href="/s" class="inline-block py-2 px-4 te\
xt-white no-underline">Standups</a>
          </li>
          <li class="flex-1 md:flex-none md:mr-3">
            <div data-controller="dropdown" class="relati\
ve inline-block">
              <button id="notification_dropdown" data-act\
ion="dropdown#toggle click@window->dropdown#hide" class="\
drop-button text-white py-2 px-2">
                <%= heroicon 'bell' %></button>
              <%= turbo_stream_from current_user, "notifi\
cation_standups" %>
              <div data-dropdown-target="menu" class="w-6\
4 hidden absolute bg-gray-800 text-white right-0 mt-3 p-3\
 overflow-auto z-30" data-transition-enter-from="opacity-\
0 scale-95" data-transition-enter-to="opacity-100 scale-1\
00" data-transition-leave-from="opacity-100 scale-100" da\
ta-transition-leave-to="opacity-0 scale-95" id="notificat\
ion_standups">
                <%= render partial: 'layouts/components/n\
otifications', locals: {notification_standups: notificati\
on_standups}%>
              </div>
            </div>
          </li>
          <li class="flex-1 md:flex-none md:mr-3">
            <div data-controller="dropdown" class="relati\
ve inline-block">
              <button data-action="dropdown#toggle click@\
window->dropdown#hide" class="drop-button text-white py-2\
 px-2">
                <%= heroicon 'cog' %>
```

```
              </button>
              <div data-dropdown-target="menu" class="hid\
den absolute bg-gray-800 text-white right-0 mt-3 p-3 over\
flow-auto z-30" data-transition-enter-from="opacity-0 sca\
le-95" data-transition-enter-to="opacity-100 scale-100" d\
ata-transition-leave-from="opacity-100 scale-100" data-tr\
ansition-leave-to="opacity-0 scale-95" id="nav-dropdown">
                <img
                  class="mx-auto rounded-full"
                  alt="User Image"
                  src='<%="#{user_avatar_url(current_user\
.email, 45 )} "%>'
                 />
                <span class="text-xs">
                  <%= current_user.email %>
                </span>

                <div class="py-4">
                  <div class="w-full border-t border-gray\
-300"></div>
                </div>

                <a href="/user/me" data-action="dropdown#\
toggle" class="p-2 hover:bg-gray-800 text-white text-md n\
o-underline hover:no-underline block"><i class="fa fa-use\
r fa-fw"></i> Profile</a>
                <a href="/account/users" data-action="dro\
pdown#toggle" class="p-2 hover:bg-gray-800 text-white tex\
t-md no-underline hover:no-underline block"><i class="fa \
fa-cog fa-fw"></i> Settings</a>

                <div class="py-2">
                  <div class="w-full border-t border-gray\
-300"></div>
                </div>
```

```
                <%= link_to("Log out", destroy_user_sessi\
on_path, :method => :delete, class: 'p-2 hover:bg-gray-80\
0 text-white text-md no-underline hover:no-underline bloc\
k')%>

              </div>
            </div>
          </li>
        </ul>
      </div>
    </div>

  </nav>
</header>
```

This is pretty simple here. We are taking a view helper (which will be created in a moment) to count and iterate over the standups to populate the line items in the notification dropdown.

There will be two steps to creating the `notification_standups` method for use in the view. First, add the following named scope into the `StandupBroadcasts` concern:

```
scope :notification_standups, lambda { |user|
      includes(user: { teams: { users: { standups: %i[did\
s todos blockers] } } })
        .references(:tasks)
        .where(teams: { users: user.id })
        .order(created_at: :desc)
        .limit(10)
    }
```

Then add the `helper_method :notification_standups` line right under the `:current_date` helper method declaration. This is what allows the view to access the method.

Finally, the `StandupBroadcasts` in its entirity with all three named scopes and all of the broadcasts:

```
module StandupBroadcasts
  extend ActiveSupport::Concern

  included do
    scope :users_standups, lambda { |user|
      user
        .standups
        .includes(:dids, :todos, :blockers)
        .references(:tasks)
        .order('standup_date DESC')
    }

    scope :teams_standups_by_date, lambda { |team, date|
      team
        .users
        .flat_map { |u| u.standups.where(standup_date: da\
te).includes(:dids, :todos, :blockers).references(:tasks)\
 }
    }

    scope :notification_standups, lambda { |user|
      includes(user: { teams: { users: { standups: %i[did\
s todos blockers] } } })
        .references(:tasks)
        .where(teams: { users: user.id })
        .order(created_at: :desc)
        .limit(10)
    }

    after_create_commit do
      broadcast_update_to(
        [user, :standups],
        target: dom_id(user, :standups),
```

```
        html: render(
          partial: 'standups/standup', collection: Standu\
p.users_standups(user), locals: { caller: 'activity_mine'\
, user: user }
        )
      )
      broadcast_update_to(
        [user, 'notification_standups'],
        target: 'notification_standups',
        html: render(
          partial: 'layouts/components/notifications',
          locals: {
            notification_standups: Standup.notification_s\
tandups(user)
          }
        )
      )
      user.teams.each do |team|
        broadcast_update_to(
          [team, standup_date, :standups],
          target: "#{team.id}_#{standup_date}_standups",
          html: render(
            partial: 'standups/standup',
            collection: Standup.teams_standups_by_date(te\
am,
                                                       sta\
ndup_date),
            locals: { caller: 'teams_show', user: user }
          )
        )
      end
    end

    after_update_commit do
      broadcast_replace_to [user, :standups], target: sel\
```

```
f, locals: { caller: 'activity_mine', user: user }
      user.teams.each do |team|
        broadcast_replace_to [team, standup_date, :standu\
ps], target: self, locals: { caller: 'teams/standups_inde\
x', user: user }
      end
    end

    after_destroy_commit do
      broadcast_remove_to [user, :standups]
      broadcast_update_to [user, 'notification_standups']\
, target: 'notification_standups',
                                                          \
  html: render(
                                                          \
    partial: 'layouts/components/notifications',
                                                          \
    locals: {
                                                          \
      notification_standups: Standup.notification_standup\
s(user)
                                                          \
    }
                                                          \
  )
      user.teams.each do |team|
        broadcast_remove_to [team, standup_date, :standup\
s]
      end
    end
  end
end
```

As this is the third time through the broadcasts, it is becoming much of the same, with different values for the parameters. `[user, 'notification_standups']`, for the identifier,

`notification_standups` for the target, and the partial uses the named scope that was just created. Additionally, there is no broadcast when a standup gets edited, as any change would not affect the data or count shown in the dropdown. Thus, only the addition or removal of a standup needs to update that markup!

Let's write a few more specs!

```
require 'rails_helper'

RSpec.describe "standup/new", type: :system do
  login_admin

  context 'show a new standup' do

    it 'on the activity#mine page when record is created'\
, js: true do
      visit activity_mine_path
      expect(page).to have_no_css('.w-96')
      check_turbo_and_sleep

      FactoryBot.create(:standup, standup_date: Date.toda\
y, user_id: @admin.id)

      expect(page).to have_css('.w-96')
    end

    it 'on the teams#show page when record is created', j\
s: true do
      team = FactoryBot.create(:team)
      team.users << @admin
      visit team_path(team)

      expect(page).to have_no_content('Dids')
      check_turbo_and_sleep
```

```
      FactoryBot.create(:standup, standup_date: Date.toda\
y, user_id: @admin.id)

      expect(page).to have_content(@admin.name)
    end

    it 'on the teams/standups#index page when record is c\
reated', js: true do
      team = FactoryBot.create(:team)
      team.users << @admin
      visit team_standups_path(team)

      expect(page).to have_no_content('Dids')
      check_turbo_and_sleep

      FactoryBot.create(:standup, standup_date: Date.toda\
y, user_id: @admin.id)

      expect(page).to have_content(@admin.name)
    end

    it 'in the notifications dropdown', js: true do
      visit activity_mine_path
      expect(page).to have_no_css('.w-96')
      click_button(id: 'notification_dropdown')
      expect(page).to have_content('0 Notifications')
      check_turbo_and_sleep

      FactoryBot.create(:standup, standup_date: Date.toda\
y, user_id: @admin.id)

      expect(page).to have_css('.w-96')
      expect(page).to have_content('1 Notifications')
      expect(page).to have_content("#{@admin.name} has a \
new Standup from #{Date.today.iso8601}")
```

```
    end
  end
end
```

As you can see in the last spec, different expectations check that the dropdown displays the correct text before and after adding the standup.

Finally, let us create a removal spec:

```
require 'rails_helper'

RSpec.describe "standup/destroy", type: :system do
  login_admin

  context 'show a destroyed standup' do

    it 'on the activity#mine page when record is destroye\
d', js: true do
      standup = FactoryBot.create(:standup, standup_date:\
 Date.today, user_id: @admin.id)

      visit activity_mine_path
      expect(page).to have_css('.w-96')
      expect(page).to have_content(Date.today.iso8601)
      check_turbo_and_sleep

      standup.destroy

      expect(page).to have_no_css('.w-96')
      expect(page).to have_no_content(Date.tomorrow.iso86\
01)
    end

    it 'on the teams#show page when record is destroyed',\
 js: true do
```

```
      team = FactoryBot.create(:team)
      team.users << @admin
      standup = FactoryBot.create(:standup, standup_date:\
 Date.today, user_id: @admin.id)

      visit team_path(team)

      expect(page).to have_content('Dids')
      check_turbo_and_sleep

      standup.destroy

      expect(page).to have_no_content('Dids')
    end

    it 'on the teams/standups#index page when record is d\
estroyed', js: true do
      team = FactoryBot.create(:team)
      team.users << @admin
      standup = FactoryBot.create(:standup, standup_date:\
 Date.today, user_id: @admin.id)

      visit team_standups_path(team)

      expect(page).to have_content('Dids')
      check_turbo_and_sleep

      standup.destroy

      expect(page).to have_no_content('Dids')
    end

    it 'in the notifications dropdown', js: true do
      standup = FactoryBot.create(:standup, standup_date:\
 Date.today, user_id: @admin.id)
```

```
      visit activity_mine_path
      expect(page).to have_css('.w-96')
      click_button(id: 'notification_dropdown')
      expect(page).to have_content('1 Notifications')
      expect(page).to have_content("#{@admin.name} has a \
new Standup from #{Date.today.iso8601}")
      check_turbo_and_sleep

      standup.destroy

      expect(page).to have_no_css('.w-96')
      expect(page).to have_content('0 Notifications')
    end
  end
end
```

This is similar to the "new standup" spec but essentially reverse.

There you have it! You have now set up to have changes, additions, and removal of Standups set to broadcast in real time around the app where standups are displayed. Next, we will introduce Turbo Frames to make the modification, removal, and addition of standups scoped to one portion of the page and remove the need to change pages!

9.4 Turbo Frames

As mentioned before, Turbo Frames are a way to break down your app into scoped updatable content. From the HotWire website:

> Turbo Frames allow predefined parts of a page to be updated on request. Any links and forms inside a frame are captured, and the frame contents automatically updated after receiving a response. Regardless of whether

> the server provides a full document, or just a fragment containing an updated version of the requested frame, only that particular frame will be extracted from the response to replace the existing content.

Turbo Frames are created by using `<turbo-frame>` tags around the content your want to be able to "switch out" based on a link/form within the frame. Turbo Frames also need a unique ID to find the content in the response AND replace the content.

We'll start by modifying the `StandupsController` and routes to remove some guarding logic around dealing with the date and the standup form. This will then allow us to set up the views with Turbo Frames!

In the `StandupsController` (`app/controllers/standups_controller.rb`) make the following changes:

```
def new
  return if check_for_blank_date
  return if check_for_existance
  @standup = Standup.new
end
```

to

```
def new
  @standup = Standup.new
end
```

Next.

```
def edit
  return if check_for_blank_date
  return if check_for_existance
  @standup = Standup.find_by(
    user_id: current_user.id,
    standup_date: current_date
  )
end
```

to

```
def edit
  @standup = Standup.find(params[:id])
end
```

Finally, add a new `destroy` method:

```
def destroy
 @standup = Standup.find(params[:id])
 @standup.destroy
 redirect_back(
     fallback_location: root_path,
     notice: 'Standup was successfully removed.'
   )
end
```

Head over to the `config/routes.rb` file to use the default routes with a path option instead of the manually declared new and edit paths:

```
get 's/new/(:date)', to: 'standups#new', as: 'new_standup'
get 's/edit/(:date)', to: 'standups#edit', as: 'edit_stan\
dup'
resources :standups, path: 's', except: [:new, :edit]
```

to

```
resources :standups, path: 's'
```

Now we can jump over to a view! First, we will update the `_standup` partial to handle the extra logic needed for frames:

```
<%= turbo_frame_tag dom_id(standup), class:"w-96 block p-\
6 bg-white rounded-lg border border-gray-200 shadow-md" d\
o %>
  <div>
    <div class="float-right">
      <% if standup.user_id == local_assigns[:user]&.id %>
        <%= link_to edit_standup_path(standup) do %>
          <%= heroicon "pencil-alt", class: 'inline' %>
        <% end %>
        <%= button_to standup_path(standup), {form_class:\
 'inline', method: :delete, form: {data: { turbo_confirm:\
 "Are you sure? Deleting this standup is permanent"}}} do\
 %>
          <%= heroicon "trash", class: 'inline' %>
        <% end %>
      <% end %>
    </div>
    <h3 class="font-bold leading-tight text-gray-900 text\
-xl"><%== standup_header standup, local_assigns[:caller]%\
></h3>
  </div>
  <h4 class="mt-4 font-bold leading-tight text-gray-900 t\
ext-lg">Dids</h4>
```

```
  <ul>
    <% standup.dids.each do |did| %>
      <li class="bg-gray-300 border border-gray-100 p-2 r\
ounded shadow-sm"> <%= did.title %> </li>
    <% end %>
  </ul>

  <h4 class="mt-4 font-bold leading-tight text-gray-900 t\
ext-lg">Todos</h4>
  <ul>
    <% standup.todos.each do |todo| %>
      <li class="bg-gray-300 border border-gray-100 p-2 r\
ounded shadow-sm"> <%= todo.title %> </li>
    <% end %>
  </ul>

  <h4 class="mt-4 font-bold leading-tight text-gray-900 t\
ext-lg">Blockers</h4>
  <ul>
    <% standup.blockers.each do |blocker| %>
      <li class="bg-gray-300 border border-gray-100 p-2 r\
ounded shadow-sm"> <%= blocker.title %> </li>
    <% end %>
  </ul>
<% end %>
```

The changes you will notice are the addition of the turbo_frame_tag instead of a `div` HTML element, the comparison of the user IDs to see if the `current_user` and the user of the standup match (which will allow the edit and delete buttons to show up) and the links to both of those actions.

Next, we will need to update the templates called the partial to ensure they pass in a user.

```
<div class="flex p-5" aria-label="Breadcrumb">
  <ol class="inline-flex items-center space-x-1 md:space-\
x-3">
    <li class="inline-flex items-center">
      <h2 href="#" class="inline-flex items-center font-m\
edium text-gray-700 hover:text-gray-900">
        My Activity
      </h2>
    </li>
  </ol>
</div>

<div class="flex justify-center">
  <div class="p-4">
    <%= turbo_stream_from current_user, "standups" %>
      <div class="grid grid-cols-3 auto-cols-auto gap-4" \
id="<%= dom_id(current_user, 'standups') %>">
        <%= render @standups, user: current_user %>
      </div>
  </div>
</div>
```

Here we are including the `user: current_user` argument to pass in the current user as a user local variable. Next, we will make a few minor changes to the `standups/_form`, `standups/edit`, `standups/new`, and `standups/index` templates to work with Turbo Frames and the controller changes. Starting with the form, we ensure we correctly ID'd forms and get the `standup_date` from the standup object first or from a parameter if the object does not have a date.

(The task part of the form is omitted for brevity as it remains the same):

```
<%= form_with model: @standup, id: 'standup-form', html: \
{class: 'space-y-6'}  do |f| %>
  <%= f.hidden_field :standup_date, value: f.object&.stan\
dup_date || params[:date] %>
  <% if @standup.errors.any? %>
    <div class="text-red-700">
      <h4>
        <%= "#{pluralize(@standup.errors.count, "error")}\
 prohibited this standup from being saved:" %>
      </h4>
      <%== @standup.errors.full_messages.map{|msg|content\
_tag(:li, msg)}.join %>
    </div>
  <% end %>
...
```

The next few template changes will be wrapping content we want to be swapped within the frame with a `turbo_frame_tag`:

```
<div class="flex p-5" aria-label="Breadcrumb">
  <ol class="inline-flex items-center space-x-1 md:space-\
x-3">
  </ol>
</div>
<div class="flex justify-center">
  <div class="p-4 w-3/5 bg-white rounded-lg border border\
-gray-200 shadow-md sm:p-6 lg:p-8">
    <h5 class="text-xl font-bold leading-none text-gray-9\
00">Edit the standup:</h5>
     <%= turbo_frame_tag dom_id(@standup) do %>
      <%= render 'form' %>
    <% end %>
  </div>
</div>
```

Using `turbo_frame_tag` is easy as you just need to provide a unique

ID via the `dom_id` helper.

Next up is the `new` template:

```
<div class="flex p-5" aria-label="Breadcrumb">
  <ol class="inline-flex items-center space-x-1 md:space-\
x-3">
  </ol>
</div>
<div class="flex justify-center">
  <div class="p-4 w-3/5 bg-white rounded-lg border border\
-gray-200 shadow-md sm:p-6 lg:p-8">
    <h5 class="text-xl font-bold leading-none text-gray-9\
00">Add a standup:</h5>
     <%= turbo_frame_tag dom_id(@standup) do %>
        <%= render 'form' %>
    <% end %>
  </div>
</div>
```

Finally, the `index` needs to be deleted as it is only part of the scaffolding from the earlier part of the book! Earlier in the chapter, you replaced the containing element with a stream tag and a unique ID. So now we're going to replace the contents inside the `div`.

Find:

```
<%= turbo_stream_from @team, Date.today.iso8601, "standup\
s" %>
<div class="grid grid-cols-3 auto-cols-auto gap-4" id="<%\
= "#{@team.id}_#{Date.today.iso8601}_standups" %>">
```

Then replace the contents with:

```
<% unless current_user.standups.where(standup_date: Date.\
today).size > 0 %>
  <% if @standups.empty? %>
    <div class="py-8">
      <p>No standups are available to display</p>
    </div>
  <% end %>
  <%= turbo_frame_tag Standup.new, class:"w-96 block p-6 \
bg-white rounded-lg border border-gray-200 shadow-md" do \
%>

    <%= link_to "Add a Standup",
            new_standup_path(date: Date.today.iso8601),
            class: "font-medium rounded-lg text-sm px-5 p\
y-2.5 text-center mr-2 mb-2 border border-gray-400 text-g\
ray-400 hover:text-white hover:bg-gray-600 focus:ring-gra\
y-800",
            data: { turbo_frame: dom_id(Standup.new) } %>
  <% end %>
<% end %>
<%= render partial: 'standups/standup', collection: @stan\
dups %>
```

This does a few things. First, it allows you to control the existence of a button to add a standup. Plus, it will enable you to keep the "no standups" wording when that is the case. If you look at the `turbo_frame_tag`, it uses the classes usually used for the shape, size, and style of a standup. This allows for a seamless interface when the content is swapped. It uses `dom_id` to set an ID on the element. Inside that tag is a `link_to` link to the form for a new standup (formerly its own page). The first three arguments are standard `link_to` options: text, link location, and HTML classes. In the last argument, we are using data attributes to specify that it will use the same `dom_id` for the `turbo_frame`. This allows the wrapping Turbo Frame to take content from the link and replace it with the response,

which will also have a matching Turbo Frame ID.

There is one other template where we will make a similar replacement of the contents.

```
<% unless current_user.standups.where(standup_date: curre\
nt_date).size > 0 %>
  <% if @standups.empty? %>
    <div class="row">
      <div class="col">
        <p>No standups are available to display</p>
      </div>
    </div>
  <% end %>
  <%= turbo_frame_tag Standup.new, class:"w-96 block p-6 \
bg-white rounded-lg border border-gray-200 shadow-md" do \
%>

    <%= link_to "Submit a Standup",
            new_standup_path(date: current_date),
            class: "font-medium rounded-lg text-sm px-5 p\
y-2.5 text-center mr-2 mb-2 border border-gray-400 text-g\
ray-400 hover:text-white hover:bg-gray-600 focus:ring-gra\
y-800",
            data: { turbo_frame: dom_id(Standup.new) } %>
  <% end %>
<% end %>
<%= render partial: 'standups/standup', collection: @stan\
dups %>
```

Lastly, while the user's standups page does not have a Turbo Frame, we need to replace the render method to pass in a user for the _standup partial.

```
<%= render partial: 'standups/standup', collection: @stan\
dups, locals: {user: current_user} %>
```

Regarding templates, with the add/edit standup page no longer in use, we do not need to have the button in the navigation bar anymore.

Remove the following content from `_header.html.erb`:

```
<div class="flex w-full content-center justify-between md\
:w-1/3 md:justify-center">
  <ul class="list-reset flex justify-between flex-1 md:fl\
ex-none items-center">
    <li class="flex-1 md:flex-none md:mr-3">
      <a href='/s/new/<%="#{Date.today.iso8601}"%>' class\
="font-medium rounded-lg text-sm px-5 py-2.5 text-center \
mr-2 mb-2 border border-gray-400 text-gray-400 hover:text\
-white hover:bg-gray-600 focus:ring-gray-800">
        Submit a Standup
      </a>
    </li>
  </ul>
</div>
```

Then to fix the alignment, find the class `md:w-1/3` and change it to `md:w-1/2`. This will put the app name and dropdown buttons back where they were before the standup button was removed.

Alright, if you have run the specs to this point, you would be able to see a few failing request specs that were utilizing the code we removed from the `StandupsController`. So let's go ahead and fix those!

```
require 'rails_helper'

RSpec.describe "/standups", type: :request do
  login_admin

  let(:valid_attributes) do
    {
      user_id: @admin.id,
      standup_date: Date.today.iso8601,
      todos_attributes: [
        { title: 'YE Old Thingy', "_destroy": 'false' }
      ]
    }
  end

  let(:invalid_attributes) do
    { user_id: nil, standup_date: Date.today.iso8601, tod\
os_attributes: [] }
  end

  let(:valid_session) { {} }

  describe 'GET #index' do
    it 'assigns a new standup as @standup' do
      get standups_path
      expect(response).to redirect_to(root_path)
    end
  end

  describe 'GET #new' do
    it 'redirects without a date' do
      get new_standup_path, params: {}
      expect(response).to have_http_status(200)
    end
  end
```

```
  describe 'GET #new' do
    it 'assigns a new standup as @standup' do
      get new_standup_path, params: { date: Date.today.is\
o8601 }
      expect(assigns(:standup)).to be_a_new(Standup)
    end
  end

  describe 'GET #edit' do
    it 'assigns the requested standup as @standup' do
      standup = FactoryBot.create(:standup, valid_attribu\
tes)
      get edit_standup_path(standup), params: { id: stand\
up.id }
      expect(assigns(:standup)).to eq(standup)
    end
  end

  describe 'POST #create' do
    context 'with valid params' do
      it 'creates a new Standup' do
        expect do
          post(
            standups_path,
            params: { standup: valid_attributes }
          )
        end.to change(Standup, :count).by(1)
      end

      it 'assigns a newly created standup as @standup' do
        post(
          standups_path,
          params: { standup: valid_attributes }
        )
```

```
        expect(assigns(:standup)).to be_a(Standup)
        expect(assigns(:standup)).to be_persisted
      end

      it 'redirects to the created standup' do
        post(
          standups_path,
          params: { standup: valid_attributes }
        )
        expect(response).to redirect_to(root_path)
      end
    end
  end

  describe 'PUT #update' do
    context 'with valid params' do
      let(:new_attributes) do
        { standup_date: Date.tomorrow.iso8601 }
      end

      it 'updates the requested standup' do
        @standup = FactoryBot.create(:standup, valid_attr\
ibutes)
        put(
          standup_path(@standup),
          params: { id: @standup.to_param, standup: new_a\
ttributes }
        )
        @standup.reload
        expect(@standup.updated_at).to be > @standup.crea\
ted_at
      end

      it 'assigns the requested standup as @standup' do
        standup = FactoryBot.create(:standup, valid_attri\
```

```
butes)
        put(
          standup_path(standup),
          params: { id: standup.to_param, standup: valid_\
attributes }
        )
        expect(assigns(:standup)).to eq(standup)
      end

      it 'redirects to the standup' do
        standup = FactoryBot.create(:standup, valid_attri\
butes)
        put(
          standup_path(standup),
          params: { id: standup.to_param, standup: valid_\
attributes }
        )
        expect(response).to redirect_to(root_path)
      end
    end
  end

  describe 'DELETE #destory' do
    let(:new_attributes) do
      { standup_date: Date.tomorrow.iso8601 }
    end

    it 'updates the requested standup' do
      standup = FactoryBot.create(:standup, valid_attribu\
tes)
      expect do
        delete( standup_path(standup), )
      end.to change(Standup, :count).by(-1)
    end
```

```
      it 'updates the requested standup' do
        standup = FactoryBot.create(:standup, valid_attri\
butes)
        delete( standup_path(standup), )
        expect(response).to redirect_to(root_path)
      end
  end
end
```

The whole file is included as a few specs were updated to remove the date parameter, some deleted that were no longer relevant, and finally, two destroy specs were added to test the new controller action.

If you run the specs, you will see they all pass, and code coverage in this app thus far is outstanding!

```
rspec
.....................................................\
.....................................................\
..................................................

Finished in 25.63 seconds (files took 3.83 seconds to loa\
d)
164 examples, 0 failures

Coverage report generated for RSpec to standup_app/covera\
ge. 467 / 483 LOC (96.69%) covered.
```

Lastly, let's merge the branch we have been running with all chapter into the main branch:

```
git add .
git commit -am "Add Turbo Streams and Frames"
git checkout main
git merge hotwire-standups
git push
```

9.5 What did you learn in this chapter?

- What is HotWire
- What a Turbo Stream is, how it works and how to use it.
- What a Turbo Frame is, how it works and how to use it.
- All of the above without writing any custom JavaScript and, in this case (any JavaScript at all!)

10. Stripe Payments and Subscriptions

10.1 Stripe

A critical part of running a SaaS application will be to process payments from your users. In years past, this sort of system would take multiple providers, custom setup, and very flakey API wrappers.

Stripe[1] came along to change all of that. Stripe provides a developer-centric payment processing service. They have a simple and understandable API and many features built into their API/system. For instance, you do not need to write any additional logic for monthly charges as Stripe has subscriptions built into their payment system. You simply need to create a customer under a new subscription object, and Stripe will handle the first and all recurring payments for you.

That only begins to scratch the surface regarding tools and features that Stripe lays at your disposal. Pre-made card information forms, coupon codes, events webhooks, tax(if needed), and so on.

This leads to the fact that Stripe will be the payment processor of choice for this chapter and this book. While Stripe is becoming available in many countries, it is not available in all. Understandably, though, Stripe is the best choice when possible. The concepts in this chapter will be mainly transferable to any other payment processor you may choose with a payment processing and subscription API.

[1] https://stripe.com

Before proceeding into the chapter, now would be a good time for you to sign up for Stripe on their Sign Up Page[2].

10.2 Stripe Gem

Before getting started, let's create a new branch:

```
git checkout -b stripe
```

To begin to use Stripe in your Rails application, you will need to install and set up the `stripe` gem. However, adding the `stripe` gem introduces a new complexity not yet seen in this application, secret keys/tokens. When using a third-party service, you will usually be provided with one or more unique strings to identify and authenticate your API calls to their service. These strings are just for you and must be kept as secure as possible. The most prevalent route to do so is to use `Environment Variables`. In most cases, this would mean setting a variable at the beginning of a command `STRIPE_KEY=89udioedkhjed rails s`. While that can be ok, for one or maybe two variables, it would become unusable past that.

However, as we have touched on earlier with ActionMailbox, Rails has a way to store credentials and will be encrypted to be passed through source code and production servers.

To add Stripe to your project, add the following to your `Gemfile`:

```
gem 'stripe'
```

Let's go ahead and add your Stripe keys. Once signed into Stripe, you will see an API link in the sidebar under the "Developers" tree menu if you have not located them yet. Your tokens, a publishable

[2] https://dashboard.stripe.com/register

key, and a secret key will be on that API page. Both are needed in conjunction to make an authenticated API call to stripe securely.

To open up your credentials for editing, you will need to use the Rails CLI with an editor variable, specifying the editor wait for the file to be decrypted:

```
EDITOR="code --wait" rails credentials:edit
```

Once open in an editor, you can add at the end of the file:

```
stripe:
  api_key: sk_test_<<YOUR UNIQUE STRING>>
  pkey: pk_test_<<YOUR UNIQUE STRING>>
```

You can verify that the credentials were saved by using the Rails console:

```
Rails.application.credentials.stripe
 => {:api_key=>"somesecret", :pkey=>"anotherlesssecret"}
```

Next, we will want to create an `initializer` file to load up a few lines of Stripe configuration:

config/initializers/stripe.rb

```
Rails.configuration.stripe = {
  publishable_key: Rails.application.credentials.stripe[:\
pkey],
  secret_key: Rails.application.credentials.stripe[:api_k\
ey]
}

Stripe.api_key = Rails.configuration.stripe[:secret_key]
```

If you are still running a console, `exit`, and open up a new Rails session in `rails c`. You will then be able to test that an API call to Stripe works:

```
2.6.3 :001 > Stripe.api_key
 => "supersecretstring"
2.6.3 :002 > Stripe::Charge.list()
 => #<Stripe::ListObject:0x3fc9c6298dc4> JSON: {
  "object": "list",
  "data": [

  ],
  "has_more": false,
  "url": "/v1/charges"
}
```

Another great thing about Stripe is that they have some of the best documentation. As you can see here[3], their explanations are thorough, code requests and responses are on the right side, and those samples are designed to work directly with your codebase.

If you want to look at some of the API calls, response objects, and details, you may want to look at `Subscriptions`, `Plans`, and `Charges`.

10.3 Products

Products will be the primary grouping mechanism for limits or abilities within the application. This is a relatively standard practice in SaaS applications. For example, suppose you were running a Project Management application. You may want a `Bronze` product that allows one project, a `Silver` for two projects, and so on. But, of course, each product will also have a different price!

In the Stripe world, products are the top-level resource to explain your tiers. In addition, prices will offer granular details such as usage gating, terms, and billing frequency.

To give you an idea about how prices work, I will quote their existing first paragraph regarding prices (products):

[3] https://stripe.com/docs/api/ruby#intro



Products describe the specific goods or services you offer to your customers. For example, you might offer a Standard and Premium version of your goods or service; each version would be a separate Product. They can be used in conjunction with Prices to configure pricing in Payment Links, Checkout, and Subscriptions.

We should create three product levels for this application to understand how products work.

- Free Tier: This will allow two users on the account and one team. Additionally, it does not allow access to a Github integration introduced in the next chapter.
- Starter Tier: Allowing ten users and three teams. Though, it will allow Github access.
- Unlimited Tier: Allowing unlimited users, teams, and all access.

As each product comprises somewhat semantic data, a YAML file is a great way to organize the products and their limits.

First, let's build the YAML file with product information. This will serve as the canonical storage of product details. The details included will include all needed Stripe Product settings.

Currently, with the product and price objects, they recommend creating both through their product builder, then using the IDs within your application. You can find the builder here[4], and we will use the IDs in the YAML file. When creating each, you will need a name (above), a description (you can use your own words but will not be required by the Rails app), and a price, which will be $0, $19, and $39, respectively.

[4] https://dashboard.stripe.com/test/products/create

config/products.yml

```yaml
free:
  users: 2
  teams: 1
  integrations: false
  amount: 0
  price_id: price_1LfWYKHLFfOoaklF2Gwwe
  interval: "month"
  product_id: "prod_MOJAWFBT9TL"
  human_name: "Free"
  active: true
  displayable: true
starter:
  users: 10
  teams: 3
  integrations: true
  amount: 1900
  price_id: price_1LfWZ6HLFfOoaklFVfCek
  interval: "month"
  product_id: "prod_MOJBKncL46E"
  human_name: "Starter"
  active: true
  displayable: true
unlimited:
  users: 9999
  teams: 9999
  integrations: true
  amount: 3900
  price_id: price_1LfWZLHLFfOoaklFMcgBa
  interval: "month"
  product_id: "prod_MOJBb53QT0n"
  human_name: "Unlimited"
  active: true
  displayable: true
```

Each product is a top-level key, then has a key-value pair approach to set user limits, team limits, integrations, and product attributes. Additionally, you can view the products you have just created in the Stripe Dashboard to copy the `product_id` and `price_id` for each. Next, we will want to write a quick initializer to load the YAML into a suitable hash to be used in a model:

config/initializers/product_loader.rb

```
file_path = "config/products.yml"
if File.exist?(file_path)
  PRODUCTS = YAML.load(File.read(file_path)).with_indiffe\
rent_access
end
```

The initializer primarily uses a bunch of Ruby built-in YAML and File methods to load the YAML plan structure into a hash that can be accessed by string(`product["users"]`) or symbol(`product[:users]`).

We will parlay this hash in a `Product` model to give it some valuable methods to make it more like an ActiveRecord model:

app/models/product.rb

```
class Product
  class NotFoundError < StandardError; end
  def self.products
    @products ||= PRODUCTS.keys.map { |k| PRODUCTS[k] }
  end

  def self.all
    products.map { |product| OpenStruct.new(product) }
  end

  def self.where(args)
    conditions = args.map do |arg_key, arg_val|
      proc { |product| product[arg_key] == arg_val }
```

```
    end
    local_products = products.select { |product| conditio\
ns.all? { |c| c.call(product) } }
    return [] if local_products.empty?

    local_products.flatten.map { |product| OpenStruct.new\
(product) }
  end

  def self.find(stripe_id)
    product = products.find { |p| p[:product_id] == strip\
e_id }
    raise NotFoundError, "Could not find Product with str\
ipe_price_id: #{stripe_id}" unless product

    OpenStruct.new(product)
  end

  def self.find_by(args)
    product = Product.where(args)&.first
    return nil if product.nil?

    OpenStruct.new(product)
  end

  def self.find_by!(args)
    product = Product.where(args)&.first
    raise NotFoundError, "Could not find Product with #{a\
rgs.to_a.flatten.first}: #{args.to_a.flatten.last}" unles\
s product

    OpenStruct.new(product)
  end
end
```

A few things here:

- the `self.products` method is a memoized shortcut to a two-dimensional array of the products from the YAML structure. The YAML -> Hash conversion leaves a multi-level hash structure. We are merely remapping the keys with content from that key's hash.
- `self.all` converts the array of hashes to an array of `OpenStruct` objects. `OpenStruct` allows a hash to be accessed more like an object.
- `self.where` is a replacement for the `ActiveRecord where` method. It takes all key-value pairs from the incoming hash and maps them to an array of `procs`. These procs will be dynamically called with a product from the array `self.products`. We're building our own dynamic `and` array selector.
- `self.find` will simply search for a product with the stripe_id passed in. Much like `find` works typically with a primary key.

With this model, you will be able to perform operations in the Rails console, such as:

```
rails c
Product.find("prod_MOJBb53QT0nAS5")
=> #<OpenStruct users=9999, teams=9999, integrations=true\
, amount=3900, price_id="price_1LfWZLHLFfOoakgBa5cI", int\
erval="month", product_id="prod_MOJBQT0nAS5", human_name=\
"Unlimited", active=true, displayable=true>
irb(main):004:0> Product.find_by(amount: 3900)
=> #<OpenStruct users=9999, teams=9999, integrations=true\
, amount=3900, price_id="price_1LfWZLHLFfOoakcgBa5cI", in\
terval="month", product_id="prod_MOJBbT0nAS5", human_name\
="Unlimited", active=true, displayable=true>
irb(main):005:0> Product.where(active: true)
=>
```

```
[#<OpenStruct users=2, teams=1, integrations=false, amoun\
t=0, price_id="price_1LfWYKHLFaklF2GwweFJk", interval="mo\
nth", product_id="prod_MOJFBT9TLxLs", human_name="Free", \
active=true, displayable=true>,
#<OpenStruct users=10, teams=3, integrations=true, amount\
=1900, price_id="price_1LfWZ6HLFfOoaklFCekb6k", interval=\
"month", product_id="prod_MOJBKncL46E6", human_name="Star\
ter", active=true, displayable=true>,
#<OpenStruct users=9999, teams=9999, integrations=true, a\
mount=3900, price_id="price_1LfWZLHLFfOoaklFMcgBa5", inte\
rval="month", product_id="prod_MOJBb53QT0nA", human_name=\
"Unlimited", active=true, displayable=true>]
```

Now that products and prices are available in Stripe and the application, we are ready to start creating subscriptions!

10.4 New Account; New Subscription

One of the best places to start this is by attaching subscriptions to an account, particularly a new one.

Let's take a step back and think about this at a high level:

1. A product and price need to exist within Stripe, and this application
2. A Stripe `Customer` will have a Stripe `Subscription` to a Stripe `Price`, which belongs to a Stripe `Product`
3. After Registration, but before loading the dashboard, these Stripe objects will be created.
4. Save all relevant Stripe information into the Account record's columns.

Since we will need to update an Account record, let's use the Rails CLI to update the model(migration, associations):

```
rails g migration AddStripeFieldsToAccounts stripe_custom\
er_id stripe_subscription_id subscription_status trial_en\
ds_at:datetime product_id
```

Run `rails db:migrate` and you are set with the updated database table.

Now, let's update the `Account` model:

app/models/account.rb

```
class Account < ApplicationRecord
  resourcify
  has_many :users
  validates :name, presence: true

  has_many :teams

  LIMIT_ATTRIBUTES = %i(users teams integrations)

  def product
    Product.find_by(product_id: product_id)
  end

  def limits
    OpenStruct.new(Product.find_by(product_id: product_id\
).to_h.slice(*LIMIT_ATTRIBUTES))
  end
end
```

Starting from `LIMIT_ATTRIBUTES` and lower will be the new code. `product` allows an easy way to access the product once an account has been created, and the `LIMIT_ATTRIBUTES` and `limits` method will be used to provide limit information for feature gating later.

Now, to make this available throughout the Controllers and Views, we will need to add a few small changes to the `ApplicationController`:

```
class ApplicationController < ActionController::Base

  ...

  helper_method :current_account
  helper_method :current_product

  ...

  def current_account
    @current_account ||= current_user.account
    @current_account
  end

  def current_product
    @current_product ||= current_user&.account&.product
    @current_product
  end
```

Here you will see a new method, `current_product`, allowing quick and easy access to the Account's product record. Then, to enable the method to be accessed by the view templates, we add a `helper_method`, and `current_product`.

The next step is adding products to the sign-up workflow. The workflow will now look a little more link this:

1. User hit's login page as usual.
2. The sign-up link now loads a product selector.
3. The user selects a product and is directed to the original user information sign-up page. However, now this page is tracking a `product` in the URL parameters.
4. The user signs up and is directed to the Account page(as previously) now, with the product still passed through as a URL parameter.

5. After the Account form is submitted, `NewRegistrationService` is called, with an extra service for creating Stripe resources.

With a Product table and model, let's move on to setting up the **New Account Product** portion of the Sign-Up flow. The first piece here will be the new `Account::ProductssController`:

app/controllers/accounts/products_controller.rb

```
class Accounts::ProductsController < ApplicationController
  skip_before_action :authenticate_user!
  layout 'devise'

  def index
    @products = Product.where(active: true, displayable: \
true)
  end

end
```

This controller is pretty simple, just two small actions. One, `index`, to display a list of plans to the end-user. The second, `show`, redirects to the actual Sign Up form with the `plan` URL parameter now in the URL with the `plan_id` of the plan picked.

This controller causes the need for other follow-up changes. First, add routes to the `routes.rb` file and then template for the `index` action. For the routes, just add the following line inside the `account` scope:

```
...
  scope 'account', as: 'account' do #this line exists
    resources :products, only: [:index], controller: 'acc\
ounts/products'
...
```

app/views/accounts/products/index.html.erb

```
<div class="flex p-5" aria-label="Breadcrumb">
  <ol class="inline-flex items-center space-x-1 md:space-\
x-3">
    <li class="inline-flex items-center">
      <h2 href="#" class="inline-flex items-center font-m\
edium text-gray-700 hover:text-gray-900">
        Select A Plan
      </h2>
    </li>
  </ol>
</div>

<section class="bg-white">
  <div class="py-8 px-4 mx-auto max-w-screen-xl lg:py-16 \
lg:px-6">
    <div class="space-y-8 lg:gap-6 xl:gap-10 lg:space-y-5\
">
      <% @products.each do |product| %>
        <div class="flex flex-col p-6 mx-auto max-w-lg te\
xt-center text-gray-900 bg-white rounded-lg border border\
-gray-100 shadow">
          <h3 class="mb-4 text-2xl font-semibold"><%= pro\
duct.human_name %></h3>
          <p class="font-light text-gray-500 sm:text-lg">\
Best option for personal use & for your next project.</p>
          <div class="flex justify-center items-baseline \
my-8">
              <span class="mr-2 text-5xl font-extrabold">\
<%= Money.new(product.amount, 'USD').format %></span>
              <span class="text-gray-500">/month</span>
          </div>
          <ul role="list" class="mb-8 space-y-4 text-left\
">
            <li class="flex items-center space-x-3">
```

```
                <strong><%= product.users %> users</stron\
g>
            </li>
            <li class="flex items-center space-x-3">
                <strong><%= product.teams %> teams</stron\
g>
            </li>
            <li class="flex items-center space-x-3">
                <%= product.integrations ? 'Allows' : "Do\
esn't Allow" %> Integrations
            </li>
          </ul>
          <a href="<%= new_user_registration_path(product\
: product.product_id)%>" class="text-white bg-blue-700 ho\
ver:bg-blue-800 focus:ring-4 focus:ring-primary-200 font-\
medium rounded-lg text-sm px-5 py-2.5 text-center">Get st\
arted</a>
        </div>
      <% end %>
    </div>
  </div>
</section>
```

The usual TailwindCSS markup elements and classes are in the template here. This template is important because it iterates over the plans(assuming you have three) and displays the cost, a button to select the product, and the product's features/limits. `Money.new(product.amount).format` is an interesting line. Here we are using the `Money` gem to format the product amount into formatted US dollars from cents. Using `new_user_registration_path(product: product.product_id)` as the links' `href`, we are sending the user to the new registration form but now with the products's `id` as a URL query parameter. The remainder of the template is just outputting information originating in the products's YAML file.

Once the link navigates the user to the new registration form, the end-user fills out the account name and proceeds. In Chapter 6, the `NewRegistrationService` was added to handle the needed changes when a new account is created. Now it is time to add to that with a further step in the workflow and its corresponding Service Objects that deal with subscriptions.

> A quick digression here to handle a `money` gem configuration that will prevent deprecation warnings from littering the logs and spec output. We will want to add `Money.locale_backend = nil` and `Money.rounding_mode = BigDecimal::ROUND_HALF_EVEN` right at the end of the application.rb file.

A one-line change will be needed to make sure the Sign Up based controller, `app/controllers/registrations_controller.rb`, will pass the plan parameter along:

```
def after_sign_up_path_for(resource)
  new_accounts_path
end
```

to:

```
def after_sign_up_path_for(resource)
  new_accounts_path(product: params[:product])
end
```

This change will now pass `product` through as a `GET` URL param.

There is another small one-line change to the `AccountsController` where you will now include the passed along `products` URL parameter to the service. Change:

```
result = NewRegistrationService.new(account: @account, us\
er: current_user)
```

to:

```
result = NewRegistrationService.new(
  account: @account,
  user: current_user,
  product: params[:product]
)
```

Let's go ahead and update the `NewRegistrationService`:

app/services/new_registration_service.rb

```
class NewRegistrationService

  Result = ImmutableStruct.new(:success?, :things, :error)

  def initialize(params)
    @user = params[:user]
    @account = params[:account]
    @product = params[:product]
  end

  def process_registration
    account_create
    create_stripe_objects
    send_welcome_email
    notify_slack
    Result.new(success: true, user: user, account: accoun\
t, error: nil)
  rescue ActiveRecord::RecordInvalid, Slack::Notifier, ::\
Stripe::StripeError => exception
    Result.new(
      success: false, user: user, account: account, error\
```

```
: exception.message
    )
  end

  private

  attr_reader :user, :account, :product

  def account_create
    post_account_setup if account.save!
  end

  def post_account_setup
    user.account_id = account.id
    user.save!
    user.add_role :admin, account
  end

  def create_stripe_objects
    Subscription::Create.new(product, user, account).proc\
ess
  end

  def send_welcome_email
    WelcomeEmailMailer.welcome_email(user).deliver_later
  end

  def notify_slack
    SlackNotificationJob.perform_later(user)
  end
end
```

There are three updates to this Service. First, there is a `product` passed into the method. The argument must be included in the initialize method and `attr_reader`. The next change is adding

`stripe_create` after `account_create` in the `process_registration` method. This will add the Stripe object creations to the Service workflow. Lastly, add the `create_stripe_objects` method with a call to a `Subscription::Create` Service Object. This object will also like the same parameters the `NewRegistrationService` got. Lastly, as this current object handles error catching, we will add `::Stripe::StripeError` to the list of exceptions that will be caught and used to set the `Result` object to unsuccessful.

Meaning, the next step here will be to create that Service!

app/services/subscription/create.rb

```
class Subscription::Create
  DEFAULT_PRODUCT = 'prod_MOJAWFBT9TLxLs'

  def initialize(product_id, user, account)
    @product_id = product_id
    @user = user
    @account = account
  end

  def process
    update_account
  end

  private

  attr_reader :product_id, :user, :account

  def product
    @product ||= Product.find_by(product_id: (product_id.\
blank? ? DEFAULT_PRODUCT : product))
  end

  def customer
    @customer ||= ::Stripe::Customer.create({ email: user\
```

```
.email, description: "Account: #{account.id}" }, idempote\
ncy_key: SecureRandom.uuid)
  end

  def subscription
    @subscription ||= ::Stripe::Subscription.create(
      { customer: customer.id, items: [ { price: product.\
price_id }, ], trial_period_days: 14 },
      idempotency_key: SecureRandom.uuid
    )
  end

  def update_account
    account.update(
      stripe_customer_id: customer.id,
      stripe_subscription_id: subscription.id,
      subscription_status: subscription.status,
      trial_ends_at: Time.at(subscription.trial_end),
      product_id: product.product_id
    )
  end
end
```

The first 15 lines(namespacing, initialization, and attr_reader) are all necessary object housecleaning. The `process` method is meant to kick off two main changes. First, create the `customer` object in Stripe. A customer is Stripe's primary classification of a `person`. Next, a `customer` will need to be created to attach subscriptions, charges, and events(webhooks). The other method, called `process`, is to make this application's user's subscription. Lastly, the `process` will catch any ActiveRecord and Stripe exceptions to be used in the response flow.

Two things are going on if we look at `customer` and `subscription` methods. One is that we are creating a random UUID. This will be used in the Stripe Customer creation API request as the

`idempotency_key`. What is the use of such a key? When making modifying API calls(create, update, delete) to the API, Stripe would like to ensure they only process one of your outgoing API requests once. Adding an `idempotency_key` is Stripe's way of handling this. This means if something crazy got weird in the series of tubes called the internet and the API request was somehow duplicated, it only actually manipulates the change once.

The Stripe customer creation API call only requires an `email` to be present to succeed. However, the optional `description` attribute is a great way to have small annotations in your Stripe account's customer directory.

You may also notice that the Stripe API method invocation starts with `::`, `::Stripe::Customer.create`. This is a small but important distinction. In Ruby, the method lookup will go up the current namespace path to find a method. If Ruby finds the namespace but not the method, it will return an exception stating the method was not found. If you prepend your namespaced method call with `::`, you are telling Ruby to start at the top of your namespace and follow namespaces down to the method. Think of this as more of an `absolute` path to your method/class/module. Since the Stripe gem uses a top-level `Stripe` namespace and our service use `Stripe` in the namespace path, the `::` is necessary to reach the Stripe gem's methods/classes/modules.

The Stripe API response is saved to an instance variable and memoized in a method that will be used in the following method, `subscription`.

The `subscription` method is much like the `customer` as it uses the `::Stripe` namespace to make an API call to the subscription API. Then returns the value to a memoized instance.

Lastly, `update_account`, which was initially the method called from the public `process` method, uses the `account` that was passed into the service and updated the record to save the new customer, subscription, and product data to the columns we added earlier!

To tie all of these changes back together, we will replace the original "Sign Up" link with the link to the products page:

app/views/devise/_links.html.erb

```
<%- if controller_name != 'sessions' %>
  <%= link_to "Log in", new_session_path(resource_name), \
class: "font-medium text-indigo-600 hover:text-indigo-500\
" %><br />
<% end %>

<%- if devise_mapping.registerable? && controller_name !=\
 'registrations' %>
  <%= link_to "Sign up", account_products_path, class: "f\
ont-medium text-indigo-600 hover:text-indigo-500" %><br /\
>
<% end %>

<%- if devise_mapping.recoverable? && controller_name != \
'passwords' && controller_name != 'registrations' %>
  <%= link_to "Forgot your password?", new_password_path(\
resource_name), class: "font-medium text-indigo-600 hover\
:text-indigo-500" %><br />
<% end %>

<%- if devise_mapping.confirmable? && controller_name != \
'confirmations' %>
  <%= link_to "Didn't receive confirmation instructions?"\
, new_confirmation_path(resource_name), class: "font-medi\
um text-indigo-600 hover:text-indigo-500" %><br />
<% end %>

<%- if devise_mapping.lockable? && resource_class.unlock_\
strategy_enabled?(:email) && controller_name != 'unlocks'\
 %>
  <%= link_to "Didn't receive unlock instructions?", new_\
unlock_path(resource_name), class: "font-medium text-indi\
```

```
go-600 hover:text-indigo-500" %><br />
<% end %>

<%- if devise_mapping.omniauthable? %>
  <%- resource_class.omniauth_providers.each do |provider\
| %>
    <%= link_to "Sign in with #{OmniAuth::Utils.camelize(\
provider)}", omniauth_authorize_path(resource_name, provi\
der), method: :post, class: "font-medium text-indigo-600 \
hover:text-indigo-500" %><br />
  <% end %>
<% end %>
```

Here `= link_to "Sign up", account_products_path` is the new path being used in the link.

10.4.1 Some Subscription Specs

If you were to try to run your tests right now, you might see something unexpected. They are submitting customers and subscriptions to your test data and can fail.

This is the first time your application is connected to the outside world, where its external dependency is not separated from you. In the long run, while it may be helpful to see a spec's data working in an external API, the general idea is that you want to mock that HTTP request from happening and control the response.

There are a few ways to handle this in a higher abstraction, such as Webmock or VCR, to mock HTTP requests and record HTTP requests, respectively. Some third-party APIs may even have mock servers to handle most of their API endpoints (which is the case for Stripe). However, in most cases, a simple mock stub to catch specific web requests and return the data we expect is the simplest way to go and does not introduce new dependencies to manage.

To mock web requests in Rails, we will add the Webmock gem, add some setup, add a few mock methods, and use those in specs!

First, you will want to add `gem 'webmock'` into your `Gemfile` in the `group :development, :test do` block. Next, run `bundle install` to install that new gem into your application. After that, we will want to add the Webmock `require` and configuration code:

spec/spec_helper.rb

```
### Near the top
require 'webdrivers'
require 'capybara/rspec'
require 'webmock/rspec'

...
### Near the bottom

WebMock.disable_net_connect!(
  {
    allow_localhost: true,
    allow: 'chromedriver.storage.googleapis.com' ## <- Al\
lows webdriver gem to get new versions of Chromedriver
  }
)
```

Now, if you rerun RSpec, you will not see any requests make it to Stripe, but you will see a bunch of failed specs due to Webmock blocking requests and failing specs (with an exception). So now we can add in our mock methods that will stub requests and mock the response.

spec/support/stripe_mocks.rb

```
module StripeMocks
  def stripe_mock_customer_success
    stub_request(:post, %r{https://api.stripe.com/v1/cust\
omers\w*})
      .to_return(body: { id: '1' }.to_json)
  end

  def stripe_mock_subscription_success
    stub_request(:post, %r{https://api.stripe.com/v1/subs\
criptions\w*})
      .to_return(
        body: {
          id: 'sub_H2vySBdpRqcHoi',
          start_date: 1_586_160_915,
          trial_end: 1_586_162_123,
          status: 'trialing'
        }.to_json
      )
  end
end
```

The general syntax is that you can stub a request with `stub_request(method, URL)` and chain the mocked response. In this case, we are stubbing Stripe Customer and Subscription API calls and handling the mocked JSON response. The `%r{}` syntax means we use a regular expression to use a specific URL, except at the end of the URL, we allow any alphanumeric character (Stripe Resource IDs).

To get these mocks available within the RSpec suite, we will need to take a quick stop in the `rails_helper` for RSpec:

spec/rails_helper.rb

```
### Near the top
require 'support/system_macros'
require 'support/stripe_mocks' ## New Line
include Warden::Test::Helpers

...
## Starting around line 40
RSpec.configure do |config|
  config.include Devise::Test::ControllerHelpers, type: :\
controller
  config.include Devise::Test::ControllerHelpers, type: :\
view
  config.include ActionMailbox::TestHelper, type: :mailbox

  config.extend ControllerMacros, type: :controller
  config.extend SystemMacros, :type => :system

  config.include StripeMocks ## New Line
```

The syntax requires the module from the support folder and includes it within all spec files in the test suite. After adding the configuration, we will add the method calls to specs following the code paths introduced in this chapter.

The first spot we will update is the `NewRegistrationService` spec. However, the change is simple. We will be adding two lines to the `before` block:

```
before do
  allow(Slack::Notifier).to receive_message_chain(:new, :\
ping)
  stripe_mock_customer_success ## New Line
  stripe_mock_subscription_success ## New Line
end
```

The `stripe_mock_customer_success` method and the `stripe_mock_subscription_success` method are called in the `before` block to register the mocked requests before each spec. Once it runs, the `NewRegistrationService` calls the code path using Stripe to create a Customer and Subscription.

The next spot to add a mock will be the `accounts_spec` in requests:

spec/requests/accounts_spec.rb

```
...
## Starting around line 23

describe "POST #create" do
    it "creates an account" do
      stripe_mock_customer_success
      stripe_mock_subscription_success

      post '/accounts', params: { account: FactoryBot.att\
ributes_for(:account)}
      expect(response).to redirect_to root_path
    end
  end
```

Lastly, in the "Sign Up" system specs, we will see failed tests as the user flow has changed to include the product page. The fix is to add one more "click" to each spec. After `click_on 'Sign up'`, you will want to add the following line to click on the first products link:

```
find("a.bg-blue-700[href='#{new_user_registration_path(pr\
oduct: Product.all.first.product_id)}']").click
```

If you run your whole test suite, all specs should now be passing!

10.5 Stripe's Billing Portal

In years past, we could spend 100 pages or more creating all the code necessary to manage a user's billing information in Stripe. However, as Stripe continues to improve their products and tooling, they also create features to make integrating billing into applications easier.

Previously, we would go through the steps to allow users to add or remove payment data, cancel their plan, upgrade/downgrade their plan and view a list of their charges. Most of these actions would need the whole stack of Rails components (controllers, views, service objects, jobs). However, now we're really handling this in two steps.

1. Create a controller that requests a unique Billing Portal URL from Stripe and redirects to the URL.
2. Create a webhook listener to listen to changes from the Stripe Event and update the `Account` record in the database.

That is it! First the new controller:

app/controllers/billing_controller.rb

```
class BillingController < ApplicationController
  def index
    session = Stripe::BillingPortal::Session.create(
      { customer: current_account.stripe_customer_id, ret\
urn_url: root_url }
    )
    redirect_to session.url, allow_other_host: true
  end
end
```

The one new concept you will see here is the extra parameter on the `redirect_to`, `allow_other_host`. By default, Rails only allows redirects to the same host as the application. However, to redirect to Stripe, we need to override this for this specific redirect.

Next, we can add the accompanying route:

```
...
patch 'user/update_password', to:'users#update_password',\
 as:'my_update_password'

get 'billing', to: 'billing#index' # <- NEW ROUTE

scope 'account', as: 'account' do
...
```

Lastly, for the portal to work on Stripe's end, you must save the Billing Portal settings at least once to activate the portal. You can find the Billing Portal settings in your main settings area for Stripe or by using this link[5].

Now, when you navigate to the link http://localhost:3000/billing, and you have a valid account record with Stripe resource IDs, you should be redirected to the Stripe Billing Portal for that account.

[5] https://dashboard.stripe.com/test/settings/billing/portal

10.5.1 Stripe Events

Now that the portal is set up, the Stripe Subscription status and your application will be out of sync when a change is made in the portal. Whether it is changing a plan, canceling, or changing billing data. Additionally, when a trial period is over, Stripe sends a webhook event. Fortunately, we can easily handle these changes by listening to those Webhook events.

The first step will be to add the `StripeEvent` gem. This gem makes configuring, securing, and selecting the events to listen to [and act on] very easy. You can add this gem by adding `gem 'stripe_event'` to your `Gemfile` and running `bundle install`

Next, you will want to set up an endpoint in Stripe to tell where the Webhook requests should be sent. You can access this through the "Developer" and then "Webhook" links or follow this link[6].

The endpoint you will want to use on that Webhook settings page will be the NGROK endpoint you set up earlier in the book with `events/stripe` at the end, which will match a route that will be added shortly. The URL should look like `https://subdomain.ngrok.io/events/stripe`. Next, the description does not have any bearing on the code and can be any description for you to recognize the webhook endpoint in your Stripe account. You will then choose "listen to" `events on your account`. Next, click the `add events` button and select `All Customer Events`. Before saving these settings, you can see the `signing_secret` needed in the configuration. Save that text, save the settings in Stripe and then add it to your Rails Credentials by using the terminal:

```
EDITOR="code --wait" rails credentials:edit
```

and add the signing secret nested under stripe:

[6] https://dashboard.stripe.com/test/webhooks/create

```
stripe:
  api_key: sk_test_poewihrf
  pkey: pk_test_kljsdfb
  signing_secret: whsec_kljshdbflkjsadf
```

Now we can continue the setup and configuration of the gem. The next step will be to add more configuration to the bottom of the existing `stripe.rb` initializer file.

config/initializers/stripe.rb

```
Rails.configuration.stripe = {
  publishable_key: Rails.application.credentials.stripe[:\
pkey],
  secret_key: Rails.application.credentials.stripe[:api_k\
ey]
}

Stripe.api_key = Rails.configuration.stripe[:secret_key]

## New code below this comment

StripeEvent.signing_secret = Rails.application.credential\
s.stripe[:signing_secret] # e.g. whsec_...

StripeEvent.configure do |events|
  events.subscribe 'customer.', StripeSubscriptionUpdate.\
new
end
```

You can see the setup is pretty straightforward. First, you will want to make sure `StripeEvent` has the signing_secret and then configure the events to listen to. The way `StripeEvents` listens to webhooks is using the `subscribe` method with two arguments. First is the name of the webhook event. It also allows you to listen to multiple events by omitting levels of the name. For example, `customer.card.`

would listen to all customer card change events. So to match the configuration in the Stripe dashboard, we will listen to all customer events using `customer.*`. The second argument is the object instance you will send the event to. `StripeEvent` expects these objects to respond to a `call` method. Let's set that file up now.

app/services/stripe_subscription_update.rb

```
class StripeSubscriptionUpdate

  def call(event)
    StripeSubscriptionUpdateJob.perform_later(event.id)
  end
end
```

This file is simple, as it forwards the event ID to a job that can be run asynchronously instead of the same response/request cycle the webhook received during.

Next, we can add the job that will do the updating of the account record:

app/jobs/stripe_subscription_update_job.rb

```
class StripeSubscriptionUpdateJob < ApplicationJob
  queue_as :default

  def perform(event_id)
    event = ::Stripe::Event.retrieve(event_id)
    account = Account.find_by(stripe_customer_id: event.d\
ata.object.customer)
    return unless account

    data = {
      subscription_status: event.data.object.status
    }

    data = data.merge(trial_ends_at: Time.at(event.data.o\
```

```
bject.trial_end)) unless event&.data&.object&.trial_end.n\
il?

    if !!event&.data&.object&.plan&.id
       product = Product.find_by(price_id: event&.data&.o\
bject&.plan&.id)
       data = data.merge(product_id: product.product_id)
    end

    account.update(data)
  end
end
```

In this job, we first look up some data, such as the remote event data from Stripe and the account record from the database. Once done, we set up the `data` hash that will be used for the account's `update` method. For example, if there is a `trial_end` value in the event data, we can set the trial ending, and if there is a plan ID attached, the plan can be updated (which applies when customers are switching between plans). Finally, the account record change is saved on the last non-end line.

10.5.1.1 Specs

We can pause here and ensure our test suite is passing and testing the new code we have added.

The first spec we can add will check if the `BillingController` can request a portal URL and redirect to that URL.

spec/requests/billing_spec.rb

```
require 'rails_helper'

RSpec.describe "Billing", type: :request do
  login_admin

  describe "GET #index" do
    it "redirects to Stripe Billing Portal" do
      stripe_mock_portal_success
      get billing_path
      expect(response).to redirect_to("https://billing.st\
ripe.com/session/{SESSION_SECRET}")
    end
  end
end
```

This is all straightforward request spec syntax, with a `get` and expecting the response to redirect. The one change to note here is the inclusion of a new mock, `stripe_mock_portal_success`. Let's go add another mock method to the `stripe_mocks.rb` file:

```
def stripe_mock_portal_success
  stub_request(:post, %r{https://api.stripe.com/v1/billin\
g_portal/sessions\w*})
    .to_return(body: { url: "https://billing.stripe.com/s\
ession/{SESSION_SECRET}" }.to_json)
end
```

Like the other mocks, we are stubbing a POST on a stripe endpoint, then returning some JSON to be used by the Stripe gem that mimics the attributes we need from a response.

With one spec file down, we can move on to the next, which will be the `StripeSubscriptionUpdate` service that the `StripeEvent` gem calls. This one will be simple, checking if the service responds to `call` and enqueues a `StripeSubscriptionUpdateJob` job.

spec/services/stripe_subscription_update_spec.rb

```
require 'rails_helper'

describe StripeSubscriptionUpdate do
  subject(:stripe_subscription_update) { StripeSubscripti\
onUpdate }

  it 'enqueues the StripeSubscriptionUpdateJob' do
    expect{
      stripe_subscription_update
      .new
      .call(OpenStruct.new(id: 'evt_1'))
    }.to have_enqueued_job(StripeSubscriptionUpdateJob)
  end
end
```

Finally, we can add a spec for the job itself!

spec/jobs/stripe_subscription_update_job_spec.rb

```
require 'rails_helper'

RSpec.describe StripeSubscriptionUpdateJob, type: :job do
  let(:customer_id) { 'cu_123'}
  it "matches with the enqueued job" do
    StripeSubscriptionUpdateJob.perform_later('evt1')
    expect(StripeSubscriptionUpdateJob).to have_been_enqu\
eued
  end

  it 'updates the account with Stripe Response informatio\
n' do
    account = FactoryBot.create(:account, stripe_customer\
_id: customer_id)
    stripe_mock_events_success(customer_id)
    StripeSubscriptionUpdateJob.perform_now(('evt1'))
```

```
    account.reload
    expect(account.subscription_status).to eq 'active'
    expect(account.product_id).to eq Product.all.last.pro\
duct_id
  end
end
```

The first spec is rather basic housekeeping as it checks if using `perform_later` enqueues the job as expected. The second, however, creates an account with a specific `customer_id`, uses that in a new mock, then checks the `account` record (reloaded for the latest state from the database) to see if it was updated with the mocked Stripe response. Here is the mock for this request type:

```
def stripe_mock_events_success(customer_id)
  stub_request(:get, %r{https://api.stripe.com/v1/events/\
\w*})
    .to_return(body: {
      id: 'evt_00000000000000',
      object: 'event',
      type: 'customer.updated',
      data: {
        object: {
          customer: customer_id,
          trial_end: Time.now.to_i,
          status: 'active',
          plan: {
            id: Product.all.last.price_id
          }
        }
      }
    }.to_json)
end
```

This one is a little different for two reasons. First, it takes an

argument for the `customer_id`. As it is used to find an account in the job, we will want to pass it in from a created account. Secondly, the event response has a bit more metadata that needs to be included for the Stripe gem to parse it correctly as if it were coming from the actual API request. Otherwise, we are filling the event data with valid mocked data to ensure the job tests all code paths without errors.

If you run all of the test suite at this point, you should have all test passing with plenty of code coverage:

```
rspec
.....................................................\
.....................................................\
..................................................

Finished in 26.81 seconds (files took 4.38 seconds to loa\
d)
168 examples, 0 failures

Coverage report generated for RSpec to standup_app/covera\
ge. 534 / 560 LOC (95.36%) covered.
```

10.6 Checking for an Active Subscription

In the same way, Devise checks for authentication on every request that comes through the Rails application; we will want to check the state of the subscription. When the subscription has a status that signifies an active account(active, trialing, past_due), we want the request to continue. However, if it is not in one of those states, we will want to handle the inactive subscription for both admins and users.

When the account is inactive, we want to direct the admins to the Billing Portal to update the account's plan. If a regular user tries to access a page while the account is inactive, we will immediately just log them back out.

The good news is this functionality needs one method in the `ApplicationController` to work.

```
class ApplicationController < ActionController::Base
  include StandupsHelper
  protect_from_forgery with: :exception

  before_action :authenticate_user!
  before_action :subscription_check # <-- new before_acti\
on

...

  protected

  def subscription_check
    return if layout_by_resource == 'devise' ||
              controller_name.in?(%w[products billing acc\
ounts]) ||
              current_account&.active?

    if current_user.has_role? :admin, current_account
      redirect_to(billing_path, notice: 'There is no vali\
d subscription')
    else
      sign_out current_user
    end
  end
```

A few minor things are going on here. First, we're checking if the request is Devise-based, as we don't want this check to interfere

with anything around sign-ups, logins, etc. The same conditional also checks if it has anything to do with the Billing Center. This is, so we don't get stuck in a redirect loop or prohibit a user from re-subscribing. Finally, the last condition in the if statement checks for an active subscription. If active, we don't want to go any further in the method. Now, if none of the conditions are met, we have an inactive account that needs some action. In the case of a user with an `:admin` role, the user will be redirected to the Billing Center to rectify the situation. Otherwise, we use the Devise helper, `sign_out`, to log the user out of the application.

There is one other small area we want to update, which would be to ensure any background or scheduled jobs do not send any messages/emails when an account is inactive.

This is simple by just adding logic to exit early from the iteration if the `team` belongs to an `account` with an inactive subscription. This only affects two jobs, `Reminders::FindTeamsJob` and `Recaps::FindTeamsJob`.

In `Reminders::FindTeamsJob` change:

```
def perform(*_args)
  teams.each { |team| Reminders::EmailUserOnTeamJob.perfo\
rm_later(team) }
end
```

to

```
def perform(*_args)
  teams.each do |team|
    next unless team.account&.active?

    Reminders::EmailUserOnTeamJob.perform_later(team)
  end
end
```

The added `next unless team.account.subscription&.active?` will skip to the next team in the iteration if the subscription is inactive.

The same added line will show up in the `Recaps::FindTeamsJob`:

```
def perform(*_args)
  teams.each do |team|
    next unless team.account&.active?

    Recaps::EmailUserOnTeamJob.perform_later(team)
  end
end
```

Now hold up! If you were to run your specs, you'd see quite a bit of failing specs. However, the explanation is simple as we are checking for a subscription, and there is not one that exists based on the current factories. However, updating these factories will break a few specs where we are expecting to deal with subscription and subscription state in a specific way. We'll update the factories to handle the bulk of the cases and re-adjust our subscription reliant spec files. Let's get started on the factories:

spec/factories/accounts.rb

```
FactoryBot.define do
  factory :account do
    name { "MyString" }
    addr1 { "MyString" }
    addr2 { "MyString" }
    city { "MyString" }
    state { "MyString" }
    zip { "MyString" }
    country { "MyString" }
    settings { "" }
    subscription_status { 'active' }
    product_id { 'po_123' }
  end
end
```

The new lines in this factory are the `subscription_status` and `product_id` and include those default values every time we create an Account (which happens when a User is created!).

Finally to capture the codepath around the end user we can add two specs that make sure the `ApplicationController` changes are working:

spec/requests/billing_spec.rb

```
require 'rails_helper'

RSpec.describe "Billing", type: :request do

  describe "GET #index" do
    context "as admin" do
      login_admin

      it "redirects to Stripe Billing Portal" do
        stripe_mock_portal_success
```

```
        get billing_path
        expect(response).to redirect_to("https://billing.\
stripe.com/session/{SESSION_SECRET}")
      end

      it "redirects to Stripe Billing Portal when account\
 is inactive" do
        @admin.account.update(subscription_status: 'inact\
ive')
        stripe_mock_portal_success
        get root_path
        expect(response).to redirect_to(billing_path)
      end
    end

    context 'as user' do
      login_user
      it "redirects to Stripe Billing Portal when account\
 is inactive" do
        @user.account.update(subscription_status: 'inacti\
ve')
        stripe_mock_portal_success
        get root_path
        expect(response).to redirect_to(root_path)
        follow_redirect!
        expect(response).to redirect_to(new_user_session_\
path)
      end
    end
  end
end
```

10.7 Limits

Just checking for an active account is the only time a subscription's information is needed on an application interaction. First, you will want to check the ability of a resource, such as a team or a user, against the current amount of the resource versus the subscribed plan's limits. The general workflow is that a resource type is passed into a method, the existing account's current usage is calculated, and the plan's resource limit is looked up. Then a simple equality comparison is made to see if the current subscription has sufficient resources available to add another one of the resources.

This will be done in two parts. First, we will add a module in the `app/controllers/concerns` folder to house the methods for checking the resource limit. This method will be invoked from other controller actions that create a `user` or `team` resource.

Let's start with the module:

app/controllers/concerns/limits.rb

```
module Limits
  def check_limits(resource_type)
    current_account
      .send(resource_type.to_s)
      .count < limit(resource_type)
  end

  def check_resource_against_limits(resource_type, &failu\
re_block)
    yield failure_block unless check_limits(resource_type)
  end

  def limit(resource_type)
    Plan.find(current_subscription.plan_id).send(resource\
_type)
```

```
  end
end
```

The method that will be called by the other controllers is `check_resource_against_limits`. When this is called, it will, in turn, call `check_limits` with the resource type passed into the method. This method effectually compares the current resource's count on the current_account and the limit per product in `products.yml`. While the high-level use of this module is simple, it uses a bit of metaprogramming to keep the code footprint relatively small. The argument sent into the method will be a symbol of either `:teams` or `:users`. This is handy because the `send` method takes a symbol as an argument.

If this is the first time you see `send`, it is a fundamental method that allows a dynamic symbol to be called as a method on the object that is invoking it. For example, this means if `check_resource_against_limits` passes in `:teams` as an argument, current_account will invoke `teams`, its method that will return the list of teams on the account due to the model association. Then, call a `count` to sum up the number of teams.

The same is valid with the `send` on the `limit` method. In this case, the work was done to make products from the `products.yaml` into Struct objects, allowing `users` and `teams` to be invoked to get the integer limit specified in the YAML file.

Lastly, the `check_resource_against_limits` method passes in a `&block`, triggered by `yield` if the resource is at or over its limit. In this case, the block is the controller-based code that should be initiated (a redirect with an alert message stating the limit has been reached).

In each of the controllers, you will need to include the Concern module, like so:

```
class TeamsController < ApplicationController
  include Limits

  ...
```

Here is its use in the `TeamsController#create` action:

```
...
  def create
    check_resource_against_limits(:teams) do
      return redirect_back(
        fallback_location: root_path,
        notice: "You do not have the resources to create \
this Team,\
 please consider upgrading your plan."
      )
    end
...
```

While it may not jump right out at you, there are two arguments, the `resource_type` passed in as `:teams`, and the block(what is inside the do..end block). Here, if the limit has been reached, the browser is redirected back to the page it was on, stating the quota has been met.

To test this within our test suite, we can make some additions to the `teams` and `users` request specs. First up is the `teams_spec`:

spec/requests/teams_spec.rb

```
...
describe "POST #create" do
  context "with valid params" do
    it "creates a new Team" do
      expect {
        post teams_path, params: {team: valid_attributes}
      }.to change(Team, :count).by(1)
    end

    it "assigns a newly created team as @team" do
      post teams_path, params: {team: valid_attributes}
      expect(assigns(:team)).to be_a(Team)
      expect(assigns(:team)).to be_persisted
    end

    it "redirects to the teams#index" do
      post teams_path, params: {team: valid_attributes}
      expect(response).to redirect_to(teams_path)
    end
  end

  context "with invalid params" do
    it "assigns a newly created but unsaved team as @team\
" do
      post teams_path, params: {team: invalid_attributes}
      expect(assigns(:team)).to be_a_new(Team)
    end

    it "re-renders the 'new' template" do
      post teams_path, params: {team: invalid_attributes}
      expect(response).to render_template("new")
    end
  end
```

```
  ## New spec
  context "when at the plan's limit" do
    it "redirects to the root with flash" do
      account.update(product_id: Product.all.first.produc\
t_id) # free account
      FactoryBot.create_list(:team, 3, account: account)

      post teams_path, params: {team: valid_attributes}
      expect(response).to redirect_to(root_path)
      expect(flash[:notice]).to eq('You do not have the r\
esources to create this Team, please consider upgrading y\
our plan.')
    end
  end
end
...
```

For brevity, only the `POST #create` specs are above, and the very last context and spec are what is new. The spec changes the attached product to the existing free plan that allows up to three teams. Then it creates three teams. When it goes to make the team in the spec, it asserts it will redirect to the root path and have the error message for being at the team limit.

Finally, we will add a similar change to the `UsersController` and its request spec.

The `UsersController#create` action is much of the same:

```
...
  def create
    check_resource_against_limits(:users) do
      return redirect_back(
        fallback_location: root_path,
        notice: "You do not have the resources to create \
this User, please consider upgrading your plan."
      )
    end
...
```

And the spec:

spec/requests/users_spec.rb

```
...
describe "POST #create" do
  login_admin

  it "creates an user" do
    post "/account/users", params: {
      user: FactoryBot.attributes_for(:user, {role: 'user\
'})
    }
    expect(response).to redirect_to account_users_path
  end

  it "renders :new on failure" do
    post "/account/users",  params: {
      user: FactoryBot.attributes_for(:user,{email: nil, \
role: 'user'})
    }
    expect(response).to render_template :new
  end

  it "fails on non-unique email" do
    user = FactoryBot.create(:user)
```

```
    post "/account/users",  params: {
      user: FactoryBot.attributes_for(:user,{
        email: user.email,
        role: 'user'
        }
      )}
    expect(response).to render_template :new
  end

  ## New Spec
  context "when at the plan's limit" do
    it "redirects to the root with flash" do
      account = @admin.account
      account.update(product_id: Product.all.first.produc\
t_id) # free account
      FactoryBot.create_list(:user, 2, account: account)

      post account_users_path, params: {user: FactoryBot.\
attributes_for(:user, {role: 'user'})}
      expect(response).to redirect_to(root_path)
      expect(flash[:notice]).to eq('You do not have the r\
esources to create this User, please consider upgrading y\
our plan.')
    end
  end
end
...
```

Again, this code is reduced for space and focus. Same as the team spec, the account uses the most restrictive plan, updates the resource, and then tries to create a new user. Then it asserts that the page redirects with the correct message on the screen.

Lastly(after taking a deep breath), let's merge the branch we have been running with all chapter into the main branch:

```
git add .
git commit -am "Add Stripe, Subscriptions, and Products i\
nto the application."
git checkout main
git merge stripe
git push
```

10.8 What did you learn in this chapter?

- How to add Stripe to a Rails application
- Using Stripe and YAML files to manage local and remote Stripe plans.
- How to create Stripe Customers and Stripe Subscriptions.
- How to use StripeJS and Elements to capture an end users card information(to be saved back into the application securely)
- Receiving Stripe Webhooks and acting on those.
- Testing all the above functionality.

10.9 Exercises

Exercise answers can be found with any package or bundle purchases of Build A SaaS App in Ruby on Rails 6[7]!

1. This chapter covered a lot of ground. However, there is so much you can do with Stripe, payments, and subscriptions. For example, stripe subscriptions allow customers to use `coupons`. Implement this into the application. Hint: You will probably want a new model attribute that will enable the coupon to be added as a parameter somewhere. Don't forget to push this into any service classes as well.

[7] https://buildasaasappinrails.com

11. Using Third-Party APIs in your Application

An integral part of running a SaaS application is connecting to a third-party API to either retrieve or send data as part of the experience in your application. In the previous chapter, you have already seen an API in use in this book by using Stripe to manage payments.

Most SaaS applications will have a subset of API used as integrations to the end-user. In this case, using the term integration will mean that the end-user is enhancing and integrating the other services data into this application. For example, imagine if you were running an Account SaaS, you would want your end-user to connect to their Bank's API or some bookkeeping software to pull that data into your SaaS. Thus, allowing them to get valuable data without extra work keeps them in your application more and more.

You can also connect to third-party APIs to indirectly enhance the end user's experience in your application. Later in this chapter, you will implement an Analytics API to capture the user's actions, pages, and other session data to help you understand how they use your application. If you notice that a particular application session gets little to no traffic, you may remove it or seek feedback from your customers on how to make it more useful for them.

11.1 Github Integration

In this Standup application, we will connect to the end user's Github account to retrieve repository activity into a particular team's Standups. When a commit, pull request, issue, or other activities are updated or initiated in Github, they will be received via Github Webhooks and saved into a local record in this application. The Github activity will be displayed on team pages and surfaced through Reminder and Recap emails.

11.1.1 Github Gems

Two widely used Github Ruby gems can be used within Ruby/Rails applications. However, no official gem is created by the Github organization. The first gem is called GithubAPI[1], and the second call OctoKit[2].

I prefer to use Github API and will use it for the duration of this chapter, but I want to leave you aware that there are choices when connecting to the Github API through Ruby.

Before getting started, let's create a new branch:

```
git checkout -b gitHub-integration
```

Adding this gem to your application will be as straightforward as any other gem you have added. Add `gem "github_api"` into your Gemfile and then run `bundle install`.

[1] https://github.com/piotrmurach/github
[2] https://github.com/octokit/octokit.rb

11.1.2 Implementing Account and Github API

Providing an integration to a third-party API on your customer's behalf may be a departure from how Stripe API interactions were done. For example, in the case of using Github's API in this application, the client will need to provide authorization and permission to allow your application's Github API connection to access their data.

At a high level(which will all be covered in detail), here is the process to get access on behalf of the end user:

1. Create an `OAuth App` in Github. This "App" will provide you with a `client_id` and a `client_secret` set of strings.
2. The user will authenticate and authorize Github access through OAuth. Oauth itself is a multi-step process.

- The end-user will be linked to your Oauth App's Github authorization page(provided by Github) with a few pieces of information in the URL.
- The end-user will allow access and be redirected back to your application.
- Your application will take the code from the redirect and use it to make a server-side call to the Github API to switch the returned code from the previous reply and turn it into a secure access token to access Github content on behalf of that end user.

3. The end-user will not be able to tie Github repositories to teams. Once the end-user adds the repository to any team, a webhook will be created via the API to their account. This webhook will specify where Github should send the events.
4. This application will receive webhook events and save pertinent data to the local record.

If the whole process seems both simple and complicated at the same time, don't worry, it is. Authorization for the end-user through OAuth is a standard yet somewhat complex process(you might even hear it called the OAuth dance out in the wild). However, third-party services and a set of RubyGems will obscure most of the rough edges. In the case of Github, we will use a gem called `OmniAuth Github`. This is a gem and a dependency on its parent gem `OmniAuth`. OmniAuth provides a way to have the OAuth authorization process wrapped up in a gem, exposing specific third-party configurations through third-party gems. Finally, OmniAuth will hook directly into Devise to save the user's OAuth information. This will significantly simplify the process of going back and forth to derive the end user's access token.

11.1.3 Granting Access to Github via OAuth

The logical entry point to adding this functionality into the application is to create a new Integrations Center that will house the install button and a place for the end-user to view the current status, repositories available for use, and integration removal information.

There will need to be a small amount of necessary data modeling to support storing the installed integrations and any required token information. In the case of integrations, it is best to use Single Table Inheritance to store the Account ID and the `type`. In this book, we are only using one third-party integration, but in a real-world SaaS application, you will commonly have two or more integrations in your application.

The second striking piece of this model will be using a JSONB column.
In PostgreSQL version 9.4 or above, you can store JSON data in a table column as binary data(versus the previous JSON storage as a string). This means that PostgreSQL can utilize advanced querying syntax AND indexes on this data. You can think of it as

performant unstructured data. The more structure you can add, the more performant you can make it.

We will prepare the database table in two steps here. First, we will use a Rails CLI migration command to create most of the table's structure. Then go to the migration file created to set a column default and add a few indexes.

```
rails g model Integration account:references type setting\
s:jsonb
```

```
class CreateIntegrations < ActiveRecord::Migration[7.0]
  def change
    create_table :integrations, id: :uuid do |t|
      t.references :account, null: false, foreign_key: tr\
ue, type: :uuid
      t.string :type
      t.jsonb :settings, null: false, default: {}

      t.timestamps
    end

    add_index :integrations, :settings, using: :gin
  end
end
```

There are three changes:

- Add `index: true` to the account column to ensure the referenced column has an index for lookups.
- Add `null: false` and `default: {}` to better facilitate JSON lookups and parsing.
- Add a `gin` index to the `settings` column. This will allow PostgreSQL to have better lookup performance when looking up a value in the JSON column.

Run `rails db:migrate` to get the migration's changes reflected in the database.

Lastly, even though Rails will create the Integration model for us with the belongs_to association, we will need to create the `Integrations::Github` model to utilize the Single Table Inheritance. To better organize the integration-based models, we will namespace them inside an `integrations` folder inside the `app/models` folder.

app/models/integrations/github.rb

```
module Integrations
  class Github < Integration
  end
end
```

This new file will allow statements like `Integrations::Github.create` to create a record in the integration's table with the `type` of `github`.

The integration authorization link and callback paths will be generated through OmniAuth.

Speaking of OmniAuth. Now is a great time to get the OmniAuth setup up and running. We can start that by adding the gems to the Gemfile(I put them under the Stripe gems in my Gemfile) and run `bundle install`

```
gem 'omniauth-github'
gem 'github_api'
```

Now, adding Github OAuth is going to require a little configuration inside of the Github Settings area as well(the following is directly from Github's guide[3]):

1. In the upper-right corner of any page, click your profile photo, then click Settings.

[3] https://developer.github.com/apps/building-integrations/setting-up-and-registering-oauth-apps/registering-oauth-apps/

2. Developer settings section in the left sidebar, click Developer settings.
3. OAuth Apps section in the left sidebar, click OAuth Apps.
4. Click New OAuth App. (Note: If you haven't created an app before, this button will say, Register a new application.)
5. In "Application name", type the name of your app. Typically, for development work, I will add `-dev` to the end of the name.
6. In "Homepage URL", type the full URL to your app's website. However, this is optional to the development OAuth App.
7. Optionally, in "Application description", type a description of your app that users will see. This is also optional to the development OAuth App.
8. In The field for the authorization callback URL of your app, "Authorization callback URL", type the callback URL of your app. You can use localhost in this field(some third-party providers do not allow it). In most cases, you can use: `http://localhost:3000/users/auth/github/callback` if you follow the default OmniAuth setup.
9. Click Register application.

After the page reloads and the app is created, you will see a Key and Secret. These strings will be needed and are best added to your encrypted credentials. To open the credentials up for editing, run (where `atom` would be the command-line interface for the editor you use):

```
EDITOR="code --wait" rails credentials:edit
```

Once open in an editor, you can add the following at the end of the file:

```
github:
  client_id: '<<key>>'
  secret: '<<secret>>'
```

You can verify that the credentials were saved by using the Rails console:

```
Rails.application.credentials.github
 => {:client_id=>"client_id", :secret=>"secret"}
```

Next, we can advise Devise to use OmniAuth for github by adding the following lines to the config file, `config/initializers/devise.rb`. In this file, you will find most Devise settings commented out. Near the bottom of the file, you will see a mention of OmniAuth, `# config.omniauth :github`. You can insert the following configuration below that line:

```
config.omniauth(
  :github,
  Rails.application.credentials.github[:client_id],
  Rails.application.credentials.github[:secret],
  scope: 'user:email repo read:org admin:org_hook admin:r\
epo_hook'
)
```

The Devise configuration will take n number of Omniauth providers, with the first argument as the provider name. The following arguments will line up with the individual OmniAuth strategies configuration. In the case of omniauth-github, it is the key, secret, and then a list of scopes. In the github API, access levels are granted via `scopes`. Scopes are a subset of permissions that the end-user will agree to provide when accepting the Oauth Application. The scopes used in the Devise configuration will allow this application to read repositories and manage webhooks.

If you were to restart your Rails server right now, you would likely see an error saying the Devise has not been updated in the User's model to reflect OmniAuth. This is as simple as updating the `devise` line to `app/models/user.rb`:

```
devise :invitable, :database_authenticatable, :registerab\
le,
       :recoverable, :rememberable, :validatable,
       :omniauthable, omniauth_providers: [:github]
```

You will notice `:omniauthable, omniauth_providers: [:github]` is added. As you add omniauth providers, you will just add their names to the array.

We're a few more steps away from closing the loop on getting an end-user to log into Github and allow this application access. The next step will be to add a specific OmniAuth callback controller and the corresponding route entries:

First, change the following in the `config/route.rb` file from:

```
devise_for :users, controllers: { registrations: 'registr\
ations' }
```

to

```
devise_for :users, controllers: {
    registrations: 'registrations',
    omniauth_callbacks: 'users/omniauth_callbacks'
  }
```

Then add the following new controller:

app/controllers/users/omniauth_callbacks_controller.rb

```
class Users::OmniauthCallbacksController < Devise::Omniau\
thCallbacksController

  def github
    @auth = request.env['omniauth.auth']
    if successful_integration_creation
      redirect_to(
        integrations_path,
        notice: 'Github integration has been added'
      )
    else
      redirect_to(
        integrations_path,
        alert: "Github was unable to add integration.\
Contact support if this issue persists."
      )
    end
  end

  private

  def create_github_integration
    Integrations::Github.create!(
      account_id: current_account.id,
      settings: {
        token: @auth.credentials.token
      }
    )
  end

  def successful_integration_creation
    @auth != :invalid_credentials &&
      @auth&.credentials&.token &&
      create_github_integration
```

```
  end
end
```

Combined, we are allowing Devise to hook into the Oauth (and OmniAuth process). We are telling Devise to use the `Users::OmniauthCallbacksController` and associated paths in the routes. Once in the controller, there is one action, `github`. The callbacks controller will have an action for each OmniAuth strategy, and in this case, just for Github.

Inside the controller itself, the `github` action is solely responsible for saving a new `Integrations::Github` record and redirecting it with the right messaging. For example, `request.env['omniauth.auth']` is a request hash set by OmniAuth within the Middleware layers. It will contain any information that the Oauth provider is sending back. While Github's case will include additional information like the user's email, we are only interested in the access token sent back from the OAuth response. That token is then saved to the integrations record to connect later when reading a list of repositories or setting up webhooks.

We use the `successful_integration_creation` method to ensure all necessary response items exist and the database record saves as expected. Otherwise, its `falsey` response will cause the user to see an error message.

With that out of the way, we can create the `IntegrationsController`. First, the route addition:

```
resources :integrations, only: [:index] do
  collection do
    delete ':provider', action: 'destroy', as: 'remove'
  end
end
```

The route is a simple resources style route, then nesting the `delete`

route to add an additional route parameter. This parameter will be used for determining which third-party integration to delete.

Now, this is all followed by the controller:

app/controllers/integrations_controller.rb

```
class IntegrationsController < ApplicationController
  def index
    @github = Integrations::Github.find_by(account_id: cu\
rrent_account.id)
    return unless @github

    fetch_remote_repos
  end

  def destroy
    if delete_integration
      redirect("Removed #{params[:provider].capitalize} i\
ntegration. Please be\
sure to remove the application through your settings area\
 in\
#{params[:provider].capitalize}")
    else
      redirect("Unable to removed #{params[:provider].cap\
italize} integration")
    end
  end

  private

  def integration
    case params[:provider]
    when 'github'
      Integrations::Github
    end
  end
```

```
  def fetch_remote_repos
    client =  Github.new oauth_token: @github.settings['t\
oken']
    @grepos = client.repos.list.body.map(&:name)
  end

  def delete_integration
    integration&.find_by(account_id: current_account.id)&\
.delete
  end

  def redirect(flash_message)
    redirect_to(
      integrations_path,
      notice: flash_message
    )
  end
end
```

Right now, the controller is relatively simple. An `index` action will prepare the integration data to be used within the rendered template. Due to simplicity, the Github local data lookup and data retrieval can all be written inline. However, as soon as you add your second integration, the look will demand a refactored approach to fetching data.

The `destroy` action uses the integration lookup to return the class of the integration needed. A record lookup, followed by a delete, will occur. If no record or the deletion fails, it will return the failure path.

The next logical step to take here is to add the `index.html.erb` with the markup needed to list available, connected, or any other relevant information:

app/views/integrations/index.html.erb

```
<div class="flex p-5" aria-label="Breadcrumb">
  <ol class="inline-flex items-center space-x-1 md:space-\
x-3">
    <li class="inline-flex items-center">
      <h2 href="#" class="inline-flex items-center font-m\
edium text-gray-700 hover:text-gray-900">
        Integrations
      </h2>
    </li>
  </ol>
</div>

<div class="flex justify-center">
  <div class="p-4 w-4/5 bg-white rounded-lg border border\
-gray-200 shadow-md sm:p-6 lg:p-8">
    <h5 class="text-xl font-bold leading-none text-gray-9\
00 py-2">Github</h5>
    <div class="space-y-6">
      <p class="mt-0 mb-4">
        Associate GitHub commit and pull request events i\
nto Standup App, set up per team.
      </p>
      <% unless @github %>
        <%= button_to "Install GitHub Integration",\
          user_github_omniauth_authorize_path, class: "te\
xt-white bg-blue-700 hover:bg-blue-800 focus:ring-4 focus\
:outline-none focus:ring-blue-300 font-medium rounded-lg \
text-sm w-full sm:w-auto px-5 py-2.5 text-center", method\
: :post, data: { turbo: false } %>
      <% else %>
        <div class="row">
          <div claSS="col-lg-12">
            <p class="mt-0 mb-4">
              Github installed with access to repos:
```

```
            </p>
            <div class="p-4 rounded-lg bg-gray-100" >
              <%= @grepos.join(", ") %>
            </div>
            <p class="mt-4 mb-4">
              You can setup teams individually to follow \
repo activity
              <br />
              <%= button_to "Remove Github Integration",\
                  remove_integrations_path('github'),\
                  {class: "text-white bg-red-700 hover:bg\
-red-800 focus:ring-4 focus:outline-none focus:ring-red-3\
00 font-medium rounded-lg text-sm w-full sm:w-auto px-5 p\
y-2.5 text-center" ,method: :delete, data:\
                    { confirm: 'Are you sure? Deleting is\
 permanent'}\
                  } %>
            </p>
          </div>
        </div>
      <% end %>
    </div>
  </div>
</div>
```

The bulk of this view is set up with some basic content boxes to hold the buttons on the page. The main `div` will be for Github integration. The key lines will be:

```
<% unless @github %>
  <%= button_to "Install GitHub Integration",\
    user_github_omniauth_authorize_path, class: "text-whi\
te bg-blue-700 hover:bg-blue-800 focus:ring-4 focus:outli\
ne-none focus:ring-blue-300 font-medium rounded-lg text-s\
m w-full sm:w-auto px-5 py-2.5 text-center", method: :pos\
t, data: { turbo: false } %>
```

Back in the controller, `@github` was set if there was an integration. Thus, if not, there will be a link(in the style of a button) to the omniauth-github "authorize" action. This action is internal to the `omniauth-github` gem and begins the OAuth process. Once clicked, the end-user will be directed to a Github-hosted page to accept the OAuth app authorization. Assuming they clicked yes, they will be redirected to your application(the URL you set in the OAuth app's settings). This is where the `OmniauthCallbacksController` get's called and processed, adding the integration record. When the `IntegrationsController` `index` page gets loaded, it will list the associated repositories to the user who authorized the application.

Let's take a minute and think about how much more work would need to be done without OmniAuth. The authorization URL would need to be manually crafted, the actual code and token transformations would need to be manually generated(and adding additional HTTP requests), and the callback would need to be handled manually.

Now, if a user has already authorized the integration, instead of seeing an "Install" link, they will see a "Remove" link:

```
<%= button_to "Remove Github Integration",\
    remove_integrations_path('github'),\
    {class: "text-white bg-red-700 hover:bg-red-800 focus\
:ring-4 focus:outline-none focus:ring-red-300 font-medium\
 rounded-lg text-sm w-full sm:w-auto px-5 py-2.5 text-cen\
ter" ,method: :delete, data:\
      { confirm: 'Are you sure? Deleting is permanent'}\
    } %>
```

This will direct the user back to the `destroy` action in the `IntegrationsController`, which was mentioned in greater detail earlier. Once an integration is removed, the `index` action is once again reloaded, and now the button will be back to an "Install" button.

11.1.4 Fetching Data through Github

Now that we have an access token from Github for the end-user, we can query github API endpoints through the `github_api` gem and get information about the repositories. If you go back to the `index` action on the `IntegrationsController`, you will see we are doing just that after retrieving the Integrations::Github record(which contains the token):

```
def fetch_remote_repos
  client =  Github.new oauth_token: @github.settings['tok\
en']
  @grepos = client.repos.list.body.map(&:name)
end
```

We connect to the Github API client(gem) and set it to the `client` variable. That is then used to get a list of repositories. `.repos` is a core classification, while the `.list` is the endpoint for getting a list of all repositories. `.list` also takes an optional argument for

automatic pagination, `.list(auto_pagination: true)`. The default limit of repositories returned is 30, and this argument will continue to retrieve repositories from Github until GitHub says there are no more to list. As this will make a request each time, turning this on could slow down the rendering of the page. I included this argument option in this paragraph to reduce confusion.

Finally, we map the repository names into a new array to be iterated within the `index` template.

```
<%= @grepos.join(", ") %>
```

Now, onto a slightly more complex area of retrieving repositories, the `team` form. This form will show up on new and editing team pages. When a Team is created or updated, it can list zero to many repositories to listen to events. The events will be collected through webhooks from Github, stored in this application, and listed on the `team#show` page and `reminder` emails.

There will be a few changes to facilitate this new functionality above, just adding in the repository selector. With all that is happening after clicking the submit button on forms, we should now use a Service Object for the process of creating or updating. Next, a new JSONB column will be on the team table to store integration settings. Here is the migration:

This will now add the column defaults and an index. You can finish this up by running `rails db:migrate`.

Also, managing the Github webhooks will have a few methods that will be best contained within another service class. After that is set up, there will be two new ActiveJob classes to handle adding and removing webhooks through the Github API.

The best place to start here is the changes to the `TeamsController`:

app/controllers/teams_controller.rb

```
class TeamsController < ApplicationController
  include Limits

  load_and_authorize_resource except: [:create]
  around_action :use_time_zone, only: [:edit]
  before_action :setup_integration_data, only: [:new, :ed\
it]

  RepoInfo = ImmutableStruct.new(:name, :owner)

  ...

  def create
    check_resource_against_limits(:teams) do
      return redirect_back(
        fallback_location: root_path,
        notice: "You do not have the resources to create \
this Team, please consider upgrading your plan."
      )
    end

    @team = Team.new(team_params.except("days"))
    response = Teams::Create.new(
      @team,
      current_account,
      team_params,
      -> { authorize!(:create, @team) }
    ).create

    if response.success?
      redirect_to teams_path, notice: 'Team was successfu\
lly created.'
    else
      set_users
```

```
      setup_integration_data
      render :new, status: :unprocessable_entity
    end
  end

  ...

  def update
    @team = Team.find(params[:id])
    response = Teams::Update.new(@team, team_params).upda\
te
    if response.success?
      redirect_to teams_url, notice: 'Team was successful\
ly updated.'
    else
      set_users
      setup_integration_data
      render :edit, status: :unprocessable_entity
    end
  end

  def destroy
    @team = Team.find(params[:id])
    @team.integration_settings['github_repos'] = nil
    Webhooks::Github::Manage.new(@team, {}).manage
    @team.destroy
    redirect_to teams_url, notice: 'Team was successfully\
 destroyed.'
  end

  ...

  private

  def team_params
```

```
    params.require(:team).permit(:name, :description, :ti\
mezone, :has_reminder,
    :has_recap, :reminder_time, :recap_time, days: [], us\
er_ids: [],
    integration_settings: [:github_collect_events, { gith\
ub_repos: [] }])
  end

  def setup_integration_data
    setup_github
  end

  def setup_github
    @github = Integrations::Github.find_by(account_id: cu\
rrent_account.id)
    return unless @github

    client =  Github.new oauth_token: @github.settings['t\
oken']
    @grepos = client.repos.list.body.map do |r|
      RepoInfo.new(name: r.name, owner: r.owner.login)
    end
  end

  def use_time_zone(&block)
    Time.use_zone(@team.timezone, &block)
  end

  def set_users
    @account_users ||=
      (
        current_account.users.where.not(invitation_accept\
ed_at: nil) +
        User.with_role(:admin, current_account)
      ).uniq
```

```
  end
end
```

Starting with the change with the least amount of other files, it needs to change with zero, which is the addition of `setup_integration_data`. This method will then call `setup_github`. This change is done to allow more integrations to be easily added later. The method to retrieve a list of repositories is much the same as earlier. Still, instead of returning an array of repository names, we will return an OpenStruct with the name and repository owner. The additional repository owner string will be needed when setting up webhooks. Adding `before_action :setup_integration_data, only: [:new, :edit]` will ensure the Github repository data will be available to the form markup generated in those two actions.

Also, we need to allow the `integration_settings` parameter to be accessible through the Strong Parameters hash by adding it to the `permit` list.

As the `app/views/teams/_form.html.erb` template file is already rather long, we will look at the end of the file, as we are rendering out a partial for the integration form elements anyway:

```
<div data-controller="visibility">
  <div>
    <%= label_tag '', "Send Recap Notification?",\
      class: 'block mb-2 text-sm font-medium text-gray-90\
0' %>
      <div class="">
        <%= f.check_box :has_recap, data: {action: "visib\
ility#toggleTargets"}, class: 'w-4 h-4 text-blue-600 bg-g\
ray-100 border-gray-300 rounded focus:ring-blue-500' %>
        <%= label_tag 'team_has_recap', " ".html_safe\
, class: 'inline' %>
      </div>
    <p id="helper-text-explanation" class="mt-2 text-sm t\
```

```
ext-gray-500">Send a recap notification on days the team \
has reports.</p>
  </div>
  <div class="pt-4" data-visibility-target="hideable" id=\
"recap-time-box" <%= f.object.has_recap ? '' : 'hidden="h\
idden"' %>
    <%= f.label :recap_time, class: 'block mb-2 text-sm f\
ont-medium text-gray-900' %>
    <%= f.time_select :recap_time, {ampm: true, minute_st\
ep: 15},\
      {style:'    display: inline;width: 10%;',class: 'bg\
-gray-50 border border-gray-300 text-gray-900 text-sm rou\
nded-lg focus:ring-blue-500 focus:border-blue-500 block w\
-full p-2.5'} %>
    <p id="helper-text-explanation" class="mt-2 text-sm t\
ext-gray-500">Time of the day to send report recap in the\
 selected time zone.</p>
  </div>
</div>
<%= render partial: 'form_integrations', locals: {f: f} %\
> <-- this
```

Here we are calling out to the `_form_integrations` and passing along the form object `f` as a local to the partial. Here is the partial:

app/views/teams/_form_integrations.html.erb

```
<% if current_account.github %>
  <div class="hr-line-dashed" />
  <div class="col-lg-11 offset-lg-1">
    <h3>Event Integration Settings</h3>
  </div>
  <div class="hr-line-dashed" />
  <div class="col-lg-11 offset-lg-1">
    <h3>Github Settings</h3>
  </div>
  <%= f.fields_for(:integration_settings) do |o| %>
```

```
    <div data-controller="visibility">
    <%= label_tag '', "Collect events on a GitHub repo?",\
        class: 'block mb-2 text-sm font-medium text-gray-\
900' %>
        <div class="">
          <%= o.check_box :github_collect_events, checked\
: !!f.object.integration_settings.dig("github_collect_eve\
nts"), data: {action: "visibility#toggleTargets"}, class:\
 'w-4 h-4 text-blue-600 bg-gray-100 border-gray-300 round\
ed focus:ring-blue-500' %>
          <%= label_tag 'github_collect_events', " ".\
html_safe, class: 'inline' %>
        </div>
      <p id="helper-text-explanation" class="mt-2 text-sm\
 text-gray-500">Listen for events from Github Repos.</p>

      <div class="pt-4" data-visibility-target="hideable"\
 id="github_collect_events-box" <%= f.object.integration_\
settings.dig("github_collect_events") ? '' : 'hidden="hid\
den"' %>>
        <%= o.label :github_repos, "Repos", class: 'contr\
ol-label col-lg-2' %>
        <div class="col-lg-10">
          <div class="fieldset">
            <% @grepos.each do |repo| %>
              <div class="checkbox checkbox-inline checkb\
ox-primary">
                <%= check_box_tag "team[integration_setti\
ngs][github_repos][]",\
                  "#{repo.name}|#{repo.owner}",\
                  (f.object.integration_settings["github_\
repos"]&.include?\
                  ("#{repo.name}|#{repo.owner}")),\
                  {id: "team_integration_settings_github_\
repos_#{repo.name}"} %>
```

```
                <%= f.label "integration_settings_github_\
repos_#{repo.name}",
                    repo.name,
                    {style: "padding-left: 0px"} %>
              </div>
            <% end %>
          </div>
        </div>
      </div>
    </div>
    </div>
  <% end %>
<% end %>
```

While this may be an intimidating template, it's actually relatively simple. First, we will be toggling a boolean value for `github_collect_events` within the `integration_settings` JSONB column. This will be used for a few things, such as controlling whether to show the list of repositories or when the service object to save the `team` record should act on any webhook changes. However, the bulk of the form's use will come from the list of checkboxes corresponding to the repositories. We use a select form element with an On/Off display to achieve this, passing a boolean result back to the JSON data.

The other form element is a list of checkboxes. The tag's name value(or the first argument in the form helper `check_box_tag`) is a nested representation of the record structure. The team, `integration_settings`, `github_repos`, and an empty set of brackets. This tells the form POST that zero to many values will be passed in. The second argument is a string we are making with the repo name and repo owner. This is the easiest way to pass it through and have it parsed later. When parsing later, we can `.split` on the pipe character(`|`). This is all much like the user checkboxes in the same form. The difference is that the values are saved to a JSONB column instead of through an association table.

Finally, for this partial to render, we will need to update the `Account` model to include a relationship to the Github Integration model:

app/models/account.rb

```
class Account < ApplicationRecord
  resourcify
  has_many :users
  validates :name, presence: true

  has_many :teams
  has_one :github, class_name: 'Integrations::Github' ## \
<-- new line

  LIMIT_ATTRIBUTES = %i(users teams integrations)

  def product
    @product ||= Product.find_by(product_id: product_id)
  end

  def limits
    OpenStruct.new(Product.find_by(product_id: product_id\
).to_h.slice(*LIMIT_ATTRIBUTES))
  end

  def active?
    %w(trialing active past_due).include?(subscription_st\
atus) && product_id.present?
  end
end
```

This brings us back to the `update` action in the `TeamsController`:

```
def update
  @team = Team.find(params[:id])
  response = Teams::Update.new(@team, team_params).update
  if response.success?
    redirect_to teams_url, notice: 'Team was successfully\
 updated.'
  else
    setup_integration_data
    set_users
    render :edit, status: :unprocessable_entity
  end
end
```

The change here, which will also encompass the new webhook updating functionality, is the call to a new `Teams::Update` object. The eventual `save` on the team record will provide the same behavior as it did earlier, only now through a `true` value in the `.success?` method in the response Struct. Here is the `Teams::Update` class:

app/services/teams/update.rb

```
module Teams
  class Update
    include Teams::Shared

    def initialize(team, team_params)
      @team = team
      @team_params = team_params
    end

    def update
      team.attributes = team_params.except('days', 'integ\
ration_settings')
      days_of_the_week
      convert_zone_times_to_utc
```

```
      Webhooks::Github::Manage.new(
        team,
        team_params['integration_settings']
      ).manage
      team.save!
      Result.new(success: true, team: team)
    rescue ActiveRecord::RecordInvalid => e
      Result.new(success: false, team: team, error: e.mes\
sage)
    end

    private

    attr_reader :team, :team_params

  end
end
```

The beauty of this class is that there is no new functionality sans the webhook stuff. As the `update` method was getting large and handling too many responsibilities in the controller, it was best to move into a Service Object with another module to `include` the shared instance methods. The shared module will be found in the same folder with the same namespace. The methods it contains are the exact same as the controller before this change:

app/services/teams/shared.rb

```
module Teams
  module Shared
    Result = ImmutableStruct.new(:success?, :team, :error)

    def days_of_the_week
      team.days = team_params[:days]
      &.map do |day|
        DaysOfTheWeekMembership.new(
          team_id: team.id,
          day: day
        )
      end || []
    end

    def convert_zone_times_to_utc
      convert_reminder
      convert_recap
    end

    def convert_reminder
      return nil unless team.reminder_time && team.has_re\
minder

      team.reminder_time =
        ActiveSupport::TimeZone[team.timezone]
        .parse(team.reminder_time.to_s[11..18])
        .utc
    end

    def convert_recap
      return nil unless team.recap_time && team.has_recap

      team.recap_time =
        ActiveSupport::TimeZone[team.timezone]
```

```
        .parse(team.recap_time.to_s[11..18])
        .utc
    end
  end
end
```

Now, going back to the `Teams::Update` class, the only *real* new line is the invocation of `Webhooks::Github::Manage.new(team).manage`. This class handles orchestrating the webhook additions and removals:

app/services/webhooks/github/manage.rb

```
module Webhooks
  module Github
    class Manage

      def initialize(team, incoming_integration_settings)
        @team = team
        @incoming_integration_settings = incoming_integra\
tion_settings
      end

      def manage
        return if
                  (
                    team.integration_settings.dig('github\
_repos') ==
                    incoming_integration_settings&.dig('g\
ithub_repos')
                  )

        check_webhooks_to_remove
        check_webhooks_to_add
      end
```

```
      private

      attr_reader :team, :incoming_integration_settings

      def check_webhooks_to_remove
        repos = (
          team.integration_settings.dig('github_repos') |\
| [] -
          incoming_integration_settings.dig('github_repos\
') || []
        ) || []
        remove_webhook(repos)
      end

      def check_webhooks_to_add
        repos = (
          incoming_integration_settings.dig('github_repos\
') || [] -
          team.integration_settings.dig('github_repos') |\
| []
        ) || []
        add_webhook(repos)
      end

      def add_webhook(repos)
        return if repos.empty?

        Webhooks::Github::AddJob.perform_later(repos, tea\
m)
      end

      def remove_webhook(repos)
        return if repos.empty?

        Webhooks::Github::RemoveJob.perform_later(repos, \
```

```
team)
      end
    end
  end
end
```

This Service class has a small surface area with a guard clause that will bail out of the process if there are no changes in the `github_repos` array(meaning there were no repos added or removed in the form). However, if there are changes, it will allow the `check_webhooks_to_remove` and `check_webhooks_to_add` methods to collect those changes. This is achieved by comparing the current array of repos on the database-backed team object and the array of repos from the form submission. Thus we can use the Array#difference functionality(or the - character) to determine the changes. As this all works best as arrays, there are default arrays assigned everywhere an array would exist if the data exists.

In `check_webhooks_to_remove`, we are basically asking what is extra in the database record, as that is now not in the form's input. So, based on the form, the repo has been removed from watching events. The following method, `check_webhooks_to_add`, is doing the same but in reverse.

Two ActiveJob classes handle the API addition and removal of Github webhooks, `Webhooks::Github::AddJob` and `Webhooks::Github::RemoveJob`. Both classes expect to be initialized with an array of repositories and the team object.

Let's start with `Webhooks::Github::AddJob`:

app/jobs/webhooks/github/add_job.rb

```
module Webhooks
  module Github
    class AddJob < ActiveJob::Base
      queue_as :default

      def perform(repos, team)
        client =
          ::Github.new oauth_token: team.account.github.s\
ettings['token']
        webhooks = team.integration_settings['github_webh\
ooks'] || []
        repos.each do |repo|
          repo_name, repo_owner = repo.split('|')
          response = client.repos.hooks.create(
            repo_owner,
            repo_name,
            webhook_attributes('https://basair.ngrok.io',\
 team)
          )
          response = response.to_h
          webhooks.push(
            webhook_record_attributes(repo_name, repo_own\
er, response)
          )
        end
        team.integration_settings['github_webhooks'] = we\
bhooks
        team.integration_settings['github_repos'] = repos
        team.save
      end

      private

      def webhook_attributes(domain, team)
```

```
        {
          name: 'web',
          config: {
            url: "#{domain}/integrations/github/webhook/#\
{team.id}",
            content_type: 'json',
            secret: team.id
          },
          active: true,
          events: %w[push pull_request]
        }
      end

      def webhook_record_attributes(repo_name, repo_owner\
, response)
        {
          repo_name: repo_name,
          repo_owner: repo_owner,
          id: response['id'],
          internal_url: response['config']['url'],
          external_url: response['url'],
          ping_url: response['ping_url'],
          test_url: response['test_url']
        }
      end
    end
  end
end
```

The job starts by initializing a new Github API(gem) client and then grabbing the array of existing webhooks(or using an empty array if there are no webhooks already tracked by the team). From there, the job iterates over each repository, using the passed-in repo argument to add the webhook. Inside the `.each`, it creates a new webhook and pushes that into the webhooks array(which will later be saved).

The attributes needed to save a webhook in the Github API are represented in the `webhook_attributes` method. In short, it will need `name: 'web'` to signify it is a request coming for a general web application. Next is the configuration, the settings required for the Github webhook creation. The main one to touch on here is the URL attribute. Github will need to know where to send the webhook. For the development and this book, we are just using an Ngrok domain name to set the domain Github will use. It is an exercise for the reader to implement a way to use a different domain name per environment. The remainder of the path will be used when routing the webhook event within this application. The response data is ready to save JSON to the local `team.integration_settings` column on API success. Here we will want the repo information, webhook ID(from Github), the URLs returned from Github, and the URL sent in as a configuration setting.

The `Webhooks::Github::RemoveJob` job is somewhat similar, with the significant change the webhook deletion Github API is utilized instead of `create.`:

app/jobs/webhooks/github/remove_job.rb

```
module Webhooks
  module Github
    class RemoveJob < ActiveJob::Base
      queue_as :default

      rescue_from('Github::Error::UnprocessableEntity') {}

      def perform(repos, team)
        client =
          ::Github.new oauth_token: team.account.github.s\
ettings['token']
        webhooks = team.integration_settings['github_webh\
ooks'] || []
        repos.each do |repo|
          webhook = delete_remote_webhook(client, team, r\
```

```
epo)
          webhooks.delete_if { |wh| wh['id'] == webhook['\
id'] }
          repos.delete(repo)
        end
        team.integration_settings['github_webhooks'] = we\
bhooks
        team.integration_settings['github_repos'] = repos
        team.save
      end

      private

      def delete_remote_webhook(client, team, repo)
        repo_name, repo_owner = repo.split('|')
        webhook = team.integration_settings['github_webho\
oks'].find do |gwh|
          gwh['repo_name'] == repo_name && gwh['repo_owne\
r'] == repo_owner
        end

        client.repos.hooks.delete repo_owner, repo_name, \
webhook['id']
        webhook
      end
    end
  end
end
```

At a high level, each repository in the iteration of repositories will need to get its webhook ID from the existing array of `github_webhooks` in the `integration_settings` JSON. That is what the `.find` is doing. It is looking for the first instance where the `repo_name` and `repo_owner` match in the list, then uses those three pieces of information to run the `delete` API call via

`client.repos.hooks.delete`.

Before heading off to handle the actual webhook HTTP requests the Github API will be serving the application, let's backtrack a bit and go over the `create` action on the `TeamsController` and the `Teams::Create` Service:

```
def create
  check_resource_against_limits(:teams) do
    return redirect_back(
      fallback_location: root_path,
      notice: "You do not have the resources to create th\
is Team, please consider upgrading your plan."
    )
  end

  @team = Team.new(team_params.except("days"))
  response = Teams::Create.new(
    @team,
    current_account,
    team_params,
    -> { authorize!(:create, @team) }
  ).create

  if response.success?
    redirect_to teams_path, notice: 'Team was successfull\
y created.'
  else
    set_users
    render :new, status: :unprocessable_entity
  end
end
```

This method seems rather large but is mostly large in appearance due to a few multi-line argument lists. Now regarding the Service being called, you may notice one of the arguments is a lambda(the

-> symbol). You can use the syntax of a lambda, much like a block. You would be able to preserve the execution of the item, by calling `.call` on it. This is handy here, as `authorize` will only be available in the context of a controller. However, it would need to be called in the service class.

app/services/teams/create.rb

```
module Teams
  class Create
    include Teams::Shared

    def initialize(team, account, team_params, auth_lambd\
a)
      @team = team
      @account = account
      @team_params = team_params
      @auth_lambda = auth_lambda
    end

    def create
      team.account_id = account.id
      team.days = days_of_the_week
      convert_zone_times_to_utc
      auth_lambda.call
      Webhooks::Github::Manage.new(
        team,
        team_params['integration_settings']
      ).manage
      team.save!
      Result.new(success: true, team: team)
    rescue ActiveRecord::RecordInvalid => e
      Result.new(success: false, team: team, error: e.mes\
sage)
    end

    private
```

```
    attr_reader :team, :account, :team_params, :auth_lamb\
da
  end
end
```

This class matches much of the `Teams::Update` class regarding setup, updates to the record, updates to Github webhooks, calls the incoming authorization lambda, and finally, saving the record. Additionally, the webhook management class will do the same as it did from the `Teams::Update` method.

The new piece here is the `auth_lambda`. A lambda is very much like a `block` or a `proc` because it is a *piece of code* that will be run later. This matters in the context of the authorization call from CanCanCan. We would not be able to call this method outside of a controller normally. However, since we are passing it in through a lambda, it will be called in the proper context when `.call` is invoked. As a bit more of an aside, if this were just a simple method invoked from a module or concern, a `block` could be passed instead. Here I feel that passing the authorization code `authorize!(:create, @team)` in a lambda is a little clearer on the intent.

Ok, now that we got all of the angles adding repository webhooks to the Github API, we can now collect events.

11.1.5 Accessing and Handling Github Webhooks

In the previous Chapter about Payments and Stripe, a specific gem could handle the receiving and routing of webhook events. Unfortunately, at this time, I have not seen anything similar for Github events. This means we will need to construct our own Controller and Action to receive the webhook HTTP requests and route the information accordingly.

At a high level, the approach we will take here is to receive the webhook HTTP request at the URL used when adding the webhook to Github. Once the request is received, a `github` action will first verify the authenticity of the event HTTP request. If that succeeds, our controller will route to different ActiveJob classes based on the type of event received.

The events will be stored in a local model(table) to the application and will be later queried to be displayed on the `team#show` page and `reminder` email.

Let's jump right in by adding a route and the new controller!

```
mount StripeEvent::Engine, at: '/events/stripe'

# Webhooks
post 'integrations/github/webhook/:team_id', to: 'events/\
github#create'

scope 'account', as: 'account' do
```

As this will likely be a group of routes as you grow the integrations you are using in your application, I suggest putting the route near the bottom, as noted above. The route will match the URL created in the webhook, accepts only an HTTP POST(which is what Github sends) requests, and will use the new `Events::GithubController`:

app/controllers/events/github_controller.rb

```
class Events::GithubController < ApplicationController
  skip_before_action :authenticate_user!
  skip_before_action :verify_authenticity_token
  skip_before_action :subscription_check

  rescue_from ActionController::BadRequest do
    head :unauthorized
  end
```

```
  def create
    verify_signature(request.body.read)

    handle_request

    head :ok
  end

  private

  def verify_signature(payload_body)
    signature =
      'sha1=' + OpenSSL::HMAC.hexdigest(
        OpenSSL::Digest.new('sha1'),
        params[:team_id],
        payload_body
      )
    unless Rack::Utils.secure_compare(
      signature,
      request.env['HTTP_X_HUB_SIGNATURE']
    )
      raise ActionController::BadRequest
    end
  end

  def handle_request
    case request.env['HTTP_X_GITHUB_EVENT']
    when 'push'
      Webhooks::Github::PushEvent::RecordJob.perform_late\
r(
        JSON.parse(request.body.read),
        params[:team_id]
      )
    when 'pull_request'
```

```
      Webhooks::Github::PullRequestEvent::RecordJob.perfo\
rm_later(
        JSON.parse(request.body.read),
        params[:team_id]
      )
    end
  end
end
```

In this file, there are three methods. The `create` method that responds to the incoming webhook HTTP request, a `verify_signature` method, and a method to handle what events are accepted and the background job that will process them. The signature verification method comes from Github's support documentation regarding webhooks. It essentially verifies that a proper Digest was generated with the supplied `secret` sent as a configuration when creating the webhook. In the case here, it is the `id`.

The `handle_request` method in the controller uses a `case` statement to conditionally switch the job being called based on the event type, which is sent in the `request.env['HTTP_X_GITHUB_EVENT']` HTTP header. There are two types of events we care about from Github right now. First is the `push` events. Github generates events for a `git push` to the repository. This means pushing up a commit, a list of commits, a tag, or more. The second type of event that is conditionally switched on is the `pull_request` event, which is sent when a pull request is assigned, unassigned, labeled, unlabeled, opened, edited, closed, reopened, or synchronized (updated due to a new push in the branch that the pull request is tracking). Also, any time a pull request review is requested or a review request is removed.

Before digging into the jobs, we should set up the `events` table and model(s) through the Rails CLI and a few extra file changes. First, we can create the base model and the table through the `rails g model` CLI:

```
rails g model Event type team:references user_name user:r\
eferences event_name event_body:text event_id event_data:\
jsonb event_time:datetime
```

Let's unpack this. The Event model will `belong to` a team and user. This makes sense as the webhooks are created on a team-by-team basis, and we will try and tie an event back to a user if we can. Event `_name`, `_body`, `_id`, and `_time` will all be event attributes we want to collect and somewhat normalize. Finally, JSONB shows up yet again. With all of the differences between type-specific event data and even event differences between different third-party integrations.

As with all of the other migrations that used JSONB, we will want to update the migration file manually to add some extra settings and index:

```
class CreateEvents < ActiveRecord::Migration[7.0]
  def change
    create_table :events, id: :uuid do |t|
      t.string :type
      t.references :team, null: false, foreign_key: true,\
 type: :uuid
      t.string :user_name
      t.references :user, type: :uuid
      t.string :event_name
      t.text :event_body
      t.string :event_id
      t.jsonb :event_data, default: {}
      t.datetime :event_time

      t.timestamps
    end

    add_index :events, :event_data, using: :gin
    add_index :events, :type
```

```
    add_index :events, :user_name
    add_index :events, :event_id
  end
end
```

`team_id` and `user_id` will automatically get indexed through reference modeling. Then we add `type` as it will be another Single Table Inheritance model, `event_data` as a `:gin` index type to allow quick lookups through the JSON data, `user_name` for quick looks up for users, and lastly, `event_id` which will be used in existential lookups. Finally, run a short `rails db:migrate` to get this database all synced up.

Next, as the `Event` model will be set up automatically from the Rails CLI(including the `belong_to` associations), we will want to add the `Github` model for the `type` based child model:

app/models/events/github.rb

```
module Events
  class Github < Event
  end
end
```

The above is typical STI modeling and not anything different than other STI based models we have been using. This means we can finally jump to the event collection ActionJob classes invoked through the `Events::GithubController` above.

Going in order, the first up is `Webhooks::Github::PushEvent::Record`:

app/models/events/github.rb

```
module Events
  class Github < Event
  end
end
```

The above is typical STI modeling and not anything different than other STI based models we have been using. This means we can finally jump to the event collection ActionJob classes invoked through the `Events::GithubController` above.

Going in order, the first up is `Webhooks::Github::PushEvent::Record`:

app/jobs/webhooks/github/push_event/record_job.rb

```
module Webhooks
  module Github
    module PushEvent
      class RecordJob < ApplicationJob
        rescue_from('ActiveRecord::RecordNotFound') {}

        def perform(json, team_id)
          team = Team.find(team_id)
          json['commits'].each do |commit|
            next if Event.exists?(event_id: commit['id'])

            event = Events::Github.new(
              event_attributes(team, commit, json)
            )
            set_user!(commit, event)
            event.save!
          end
        end

        private
```

```
        def event_attributes(team, commit, json)
          {
            team_id: team.id,
            event_name: 'commit',
            event_body: commit['message'],
            event_id: commit['id'],
            event_time: Time.parse(commit['timestamp']),
            event_data: event_custom_attributes(commit, j\
son)
          }
        end

        def event_custom_attributes(commit, json)
          {
            commit_time: commit['timestamp'],
            commit_url: commit['url'],
            branch: json['ref'].split('/')[-1],
            repo_name: json['repository']['name'],
            repo_url: json['repository']['html_url'],
            changed: {
              added: commit['added'],
              removed: commit['removed'],
              modified: commit['modified']
            }
          }
        end

        def set_user!(commit, event)
          if (user = User.find_by(github_username: commit\
['author']['username']))
            event.user_id = user.id
          elsif (user = User.find_by(email: commit['autho\
r']['email']))
            event.user_id = user.id
          else
```

```
            event.user_name = commit['author']['username']
          end
        end
      end
    end
  end
end
```

Let's dig into what is going on in this job. By its line count, the perform method is relatively short. The first line looks up the team from the team_id passed into the method. While ActiveJob would have allowed the team object itself to be passed in, it is best to push the lookup into the job to make the webhook controller's response as fast as possible.

Once we have the team, we can iterate over one or more commit objects from the event payload. If you think about normal git usage, this makes sense. Many small commits are often made before they are pushed up to Github. Once inside the iteration, we check to see if the Event has already been recorded in the local database. Just like Stripe, it can not be guaranteed the Event was sent just once. Thus, we must do a check ourselves to make sure duplicates are not being stored. If the Event exists, we tell the iteration to move on to the next item.

After that, a `new` Events::Github object is created with the hash values from the `event_attributes` method. In that method, we use the `team.id` and values from the event payload to set the top-level event information, such as `event_name`, the `event_time`, and the rest. The last value is another method call that sets the `event_data` JSONB structure. From here, we can add custom data such as URLs to the specific commit, diff data, and a list of commit files(added, removed, or changed).

Finally, after that, data is collected and set on the new Event object, we invoke a method to find and set a user or user_name. We'll stop here as we need to add another field to the User model:

```
rails g migration AddGithubUsernameToUsers github_username
rails db:migrate
```

Note: Rails CLI migrations generators have a shortcut naming syntax to add new columns. If you use `Add<<ColumnName>>To<<pluralModel>> column_name:type`, it will create a migration with everything needed to add the column.

Back to the job class, we will use a conditional if/elsif/else statement to find a local User record by first the github_username(or `commit['author']['username']`) from the payload). If that does not find a user, the email address from the commit is attempted to be matched with the email address locally. Last, if both fail, we simply store the GitHub username from the payload to the `event.user_name` field.

On the very last line of the iteration, the event is saved. This process will be repeated for as many commits as there are in the event. That concludes the job class for the `push` event. Let's move on to the class for the `pull_request` event, `Webhooks::Github::PullRequestEvent::Record`:

app/jobs/webhooks/github/pull_request_event/record_job.rb

```
module Webhooks
  module Github
    module PullRequestEvent
      class RecordJob < ApplicationJob
        rescue_from('ActiveRecord::RecordNotFound') {}

        def perform(json, team_id)
          pull_request = json['pull_request']
          team = Team.find(team_id)
          data_from_json = determine_event_name_and_time(
            json['action'],
            pull_request
          )
```

```
          return if Event.exists?(
            event_id: pull_request['id'],
            event_name: data_from_json[:event]
          )

          event = Events::Github.new(
            event_attributes(team, pull_request, json, da\
ta_from_json)
          )
          set_user!(pull_request, event)
          event.save!
        end

        private

        def determine_event_name_and_time(action, pull_re\
quest)
          case action
          when 'opened'
            {
              event: 'pull_request_opened',
              event_time: Time.parse(pull_request['create\
d_at'])
            }
          when 'closed'
            if pull_request['merged']
              {
                event: 'pull_request_merged',
                event_time: Time.parse(pull_request['merg\
ed_at'])
              }
            else
              {
                event: 'pull_request_closed',
                event_time: Time.parse(pull_request['clos\
```

```
ed_at'])
              }
            end
          end
        end

        def event_attributes(team, pull_request, json, da\
ta_from_json)
          {
            team_id: team.id,
            event_name: data_from_json[:event],
            event_body: pull_request['title'],
            event_id: pull_request['id'],
            event_time: data_from_json[:event_time],
            event_data: event_custom_attributes(pull_requ\
est, json)
          }
        end

        def event_custom_attributes(pull_request, json)
          {
            pull_request_url: pull_request['html_url'],
            diff_url: pull_request['diff_url'],
            pull_head: pull_request['head']['label'],
            pull_base: pull_request['base']['label'],
            repo_url: json['repository']['html_url'],
            repo_name: json['repository']['name']
          }
        end

        def set_user!(pull_request, event)
          if (user = User.find_by(github_username: pull_r\
equest['user']['login']))
            event.user_id = user.id
          else
```

```
            event.user_name = pull_request['user']['login\
']
          end
        end
      end
    end
  end
end
```

The `PullRequestEvent` class is very similar but will utilize more information to address the variations the event payload might contain.
Since a `pull_request` event might be sent when a pull request is opened, merged, or closed (and not merged), we must adjust the `event_name` and pull the time from a different payload structure for each type. That logic is contained in the `determine_event_name_and_time` method, which will determine the `action` of the event to set the name and time.

The existential check is slightly different now. It would need to have the same ID *and* the same action to be an actual duplicate event. Now that we are past those differences, a new Event is created similarly with the `event_name` and `event_time` provided from a hash returned on the earlier method. The main `event_attributes` method will also call the method `event_custom_attributes` to capture event type-specific unstructured data to be stored in the JSONB column. Next, the user is attempted to be set again through `set_user`, but instead, there is no User email lookup this time. In the pull_request-based payloads, there is no email provided. Thus, the lookup could not even be attempted. Lastly, the event is finally saved to the database.

With receiving and storing the event data from Github webhooks out of the way, we will now want to display the data to the end users.

11.1.6 Displaying Event Data

There will be two main areas where the event data will be displayed. The `team#show` and `teams#standups` will show the data through the web application. The event data will also be sent out to the standup reminder email.

The first part will be to expose the events needed through a query on the team's events:

To use an association to retrieve events from a specific team, we will need to add the inverse relationship of `belongs_to` on the `Team` model:

```
  has_many :events
```

From there, we can update the `TeamsController` with a private method to load up the events and its invocation on the `show` action:

app/controllers/teams_controller.rb

```
class TeamsController < ApplicationController
  around_action :use_time_zone, only: [:edit, :show, :sta\
ndups]
  include TeamEvents ## new Lines
  ...

  def show
    set_teams_and_standups(Date.today.iso8601)
    set_events ## new line
  end

  ...

end
```

Then, add it to the `Teams::StandupsController`:

app/controllers/teams/standups_controller.rb

```
class Teams::StandupsController < ApplicationController
  include TeamEvents ## new line
  def index
    set_teams_and_standups(current_date)
    set_events ## new line
  end
end
```

Finally, we can add the method to a new concern `TeamEvents` to allow it to be available to multiple controllers:

app/controllers/concerns/team_events.rb

```
module TeamEvents
  def set_events
    @events =
      @team
      .events
      .where(
        event_time: (
          current_date.to_time.beginning_of_day..current_\
date.to_time.end_of_day
        )
      )
      .includes(%i[user team])
      .order('event_time DESC')
  end
end
```

The critical part here is the `set_events` method, which will query the events by the current `@team` and the `current_date` at the beginning and end of the day. We use `includes` to make sure relationship records will be loaded efficiently and finally order the most recent results first.

Let's start with the template for `teams#show`:

app/views/teams/show.html.erb

```
<div class="flex p-5" aria-label="Breadcrumb">
  <ol class="inline-flex items-center space-x-1 md:space-x-3">
    <li class="inline-flex items-center">
      <h2 href="#" class="inline-flex items-center font-medium text-gray-700 hover:text-gray-900">
        Teams
      </h2>
    </li>
    <li class="inline-flex items-center">
      <h2 href="#" class="inline-flex items-center font-medium text-gray-700 hover:text-gray-900">
        <svg class="w-6 h-6 text-gray-400" fill="currentColor" viewBox="0 0 20 20" xmlns="http://www.w3.org/2000/svg"><path fill-rule="evenodd" d="M7.293 14.707a1 1 0 010-1.414L10.586 10 7.293 6.707a1 1 0 011.414-1.414l4 4a1 1 0 010 1.414l-4 4a1 1 0 01-1.414 0z" clip-rule="evenodd"></path></svg>
        <% @team.name %>
      </h2>
    </li>
  </ol>
</div>

<div class="flex justify-center">
  <div class="p-4 w-4/5 bg-white rounded-lg border border-gray-200 shadow-md sm:p-6 lg:p-8">
    <div class="float-right">
      <% if current_user.has_role? :admin, current_account %>
        <%= link_to edit_team_path(@team), class: 'inline' do %>
          <%= heroicon "pencil-alt", class: 'inline' %>
        <% end %>
```

```
        <%= button_to team_path(@team), {form_class: 'inl\
ine', method: :delete, form: {data: { turbo_confirm: "Are\
 you sure? Deleting this team is permanent"}}} do %>
          <%= heroicon "trash", class: 'inline' %>
        <% end %>
      <% end %>
    </div>
    <h3 id="team-settings"><%= @team.name %> Team Setting\
s</h3>
    <p>
      <span>Users:</span>
      <%= @team.users.map(&:email).join(", ") %>
      <% if @team.has_reminder %>
        <span>| Reminder at: </span>
        <%= "#{@team.reminder_time.in_time_zone(current_u\
ser.time_zone)\
          .strftime("%I:%M %p %Z")}\
          (#{@team.reminder_time.in_time_zone(current_use\
r.time_zone)\
          .strftime("%H:%M")})" %>
      <% end %>
      <% if @team.has_recap %>
        <span>| Recap at: </span>
        <%= "#{@team.recap_time.in_time_zone(current_user\
.time_zone)\
        .strftime("%I:%M %p %Z")}\
        (#{@team.recap_time.in_time_zone(current_user.tim\
e_zone)\
        .strftime("%H:%M")}) " %>
      <% end %>
    </p>
  </div>
</div>

<div class="flex justify-center mt-6">
```

```
  <div class="px-4 py-1 w-4/5 bg-white rounded-lg border \
border-gray-200 shadow-md">
    <div class="float-right">
      <%= link_to "See More", team_standups_path(@team) %>
    </div>
    <h3 class="card-title"><%= "\"#{@team.name}\"'s Stand\
ups Today" %></h3>
  </div>
</div>

<%= render partial: 'teams/events', locals:\
      {events: @events, title: "\"#{@team.name}\" Events"\
} %>

<div class="flex justify-center">
  <div class="p-4 w-4/5 sm:p-6 lg:p-8">
  <%= turbo_stream_from @team, Date.today.iso8601, "stand\
ups" %>
    <div class="grid grid-cols-3 auto-cols-auto gap-4" id\
="<%= "#{@team.id}_#{Date.today.iso8601}_standups" %>">
      <% unless current_user.standups.where(standup_date:\
 Date.today).size > 0 %>
        <% if @standups.empty? %>
          <div class="py-8">
            <p>No standups are available to display</p>
          </div>
        <% end %>
        <%= turbo_frame_tag Standup.new, class:"w-96 bloc\
k p-6 bg-white rounded-lg border border-gray-200 shadow-m\
d" do %>


          <%= link_to "Add a Standup",
                 new_standup_path(date: Date.today.iso86\
01),
```

```
                  class: "font-medium rounded-lg text-sm \
px-5 py-2.5 text-center mr-2 mb-2 border border-gray-400 \
text-gray-400 hover:text-white hover:bg-gray-600 focus:ri\
ng-gray-800",
                  data: { turbo_frame: dom_id(Standup.new\
) } %>
        <% end %>
      <% end %>
      <%= render partial: 'standups/standup', collection:\
 @standups %>
    </div>
  </div>
</div>
```

The whole view template is included to show the location and indentation of the new markup, but particularly, the following was added:

```
<%= render partial: 'events', locals:\
  {events: @events, title: "Today's \"#{@team.name}\" Eve\
nts"} %>
```

Here we are calling out a partial to handle displaying the markup for the events. We're passing in `@events` and a title to be displayed as `locals`. Here is the `events` partial's markup:

app/views/teams/_events.html.erb

```
<div class="flex justify-center mt-6">
  <div class="px-4 py-1 w-4/5 bg-white rounded-lg border \
border-gray-200 shadow-md">
    <h3 class="font-bold text-lg mb-1"><%= title %></h3>
    <% if events.blank? %>
        <p> No Events have been recorded </p>
      <% else %>
        <% events.each do |event| %>
          <p>
            <%= "#{event.event_name.humanize} '#{event.ev\
ent_body}' by \
              #{User.find_by(id: event.user_id)&.name || \
event.user_name} " %>
            <%== local_time_ago(event.event_time) %>
            - Repo:
            <a
              href="<%= event.event_data.dig("repo_url")%\
>"
              target="_blank"
              style="margin:0 5px"> <%= event.event_data.\
dig("repo_name") %></a>
          </p>
        <% end %>
      <% end %>
  </div>
</div>
```

In this case, we are iterating over each event from the controller, displaying a string about the `event_name`, who it was done by, and the time. It also includes a link to the Github repository. We are using only some of the captured data here. You could display stats from the file changes, a link to a diff, and more!

Since we are using a partial, we can just as easily update the

`teams#standups` view template to make the same display available on the "standups" page:

app/views/teams/standups/index.html.erb

```
<div class="flex p-5" aria-label="Breadcrumb">
  <ol class="inline-flex items-center space-x-1 md:space-\
x-3">
    <li class="inline-flex items-center">
      <h2 href="#" class="inline-flex items-center font-m\
edium text-gray-700 hover:text-gray-900">
        Teams
      </h2>
    </li>
    <li class="inline-flex items-center">
      <h2 href="#" class="inline-flex items-center font-m\
edium text-gray-700 hover:text-gray-900">
        <svg class="w-6 h-6 text-gray-400" fill="currentC\
olor" viewBox="0 0 20 20" xmlns="http://www.w3.org/2000/s\
vg"><path fill-rule="evenodd" d="M7.293 14.707a1 1 0 010-\
1.414L10.586 10 7.293 6.707a1 1 0 011.414-1.414l4 4a1 1 0\
 010 1.414l-4 4a1 1 0 01-1.414 0z" clip-rule="evenodd"></\
path></svg>
        <%= @team.name %>
      </h2>
    </li>
  </ol>
</div>

<%= render partial: 'layouts/components/date_change' %>

<%= render partial: 'teams/events', locals:\
      {events: @events, title: "\"#{@team.name}\" Events \
for this date"} %>

<div class="flex justify-center">
  <div class="p-4 w-4/5 sm:p-6 lg:p-8">
```

```
    <%= turbo_stream_from @team, current_date, "standups"\
 %>
    <div class="grid grid-cols-3 auto-cols-auto gap-4" id\
="<%= "#{@team.id}_#{current_date}_standups" %>">
      <% unless current_user.standups.where(standup_date:\
 current_date).size > 0 %>
        <% if @standups.empty? %>
          <div class="row">
            <div class="col">
              <p>No standups are available to display</p>
            </div>
          </div>
        <% end %>
        <%= turbo_frame_tag Standup.new, class:"w-96 bloc\
k p-6 bg-white rounded-lg border border-gray-200 shadow-m\
d" do %>


          <%= link_to "Submit a Standup",
                  new_standup_path(date: current_date),
                  class: "font-medium rounded-lg text-sm \
px-5 py-2.5 text-center mr-2 mb-2 border border-gray-400 \
text-gray-400 hover:text-white hover:bg-gray-600 focus:ri\
ng-gray-800",
                  data: { turbo_frame: dom_id(Standup.new\
) } %>
        <% end %>
      <% end %>
      <%= render partial: 'standups/standup', collection:\
 @standups %>
    </div>
  </div>
</div>
```

We insert the partial right below the date changer in the' standups'

template. Again, to ensure the page displays the event information correctly, the `@events` and `title` are passed into partial.

If you were to visit your team page and you had events collected from the current day, you would see them right away. On the `standups` page, the date is changed through the `current_date`, and both the list of standups and events are looked up through the output of the `current_date`.

Lastly, there is one more spot to make the webhook events show up: the reminder email. This change will occur over two files: the mailer and the mail template.

Let's get started with the mailer class, as that will query and load in the events:

app/mailers/email_reminder_mailer.rb

```
class EmailReminderMailer < ApplicationMailer

  def reminder_email(user,team)
    @user = user
    @team = team
    reply_to = "'Standups App' <standups.#{@user.id}@app.\
yourdomain.com>"
    make_bootstrap_mail(
      to: @user.email,
      subject: "#{team.name} Standup Reminder!",
      reply_to: reply_to
    )
  end
end
```

The mailer adds an assignment to `@events`, much like the `TeamsController` method you saw earlier. The data and event query is ultimately up to you to decide the length of time to go back(in this code, it's 24 hours).

app/views/email_reminder_mailer/reminder_email.html.erb

```
<div class="container">
  <span class="text-secondary text-center" style="font-si\
ze:8px">##- Please type your reply above this line -##</s\
pan>
  <h1 class="text-center mb-4">
    Standup App
  </h1>
  <div class="card mb-4">
    <div class="card-body">
      <h3 class="text-center mb-4"><%= @team.name%> Remin\
der! </h3>
      <p class="mb-4">
        Just wanted to remind you to add your standup for
        the team: <%= @team.name %>
      </p>
      <p>
        <span>Events from the last 24 hours:<span>
        <% @events.each do |event| %>
          <p>
            <%= "#{event.event_name.humanize} \
              '#{event.event_body}' by \
              #{User.find_by(id: event.user_id)&.name\
               || event.user_name} " %>
            <%= ActiveSupport::TimeZone[@team.timezone]\
              .parse(event.event_time.to_s).strftime("%D \
%r") %>
           </p>
          <% end %>
        </p>
      <%= link_to "Add Your Standup", new_standup_url(), \
{class: "btn btn-primary btn-lg mx-auto mt-2", style: "wi\
dth:95%"} %>
    </div>
  </div>
```

```
</div>
```

Inside the markup, you will see a loop set to iterate over the events within the `div` above the button. This change is about it as there is not much different than the team-based event information, aside from the mailer's markup version being just a little bit more slimmed down to make it more digestible:

You can test this out by invoking the following inside Rails Console(or aka `rails c`) and running the following:

```
EmailReminderMailer.reminder_email(User.last, Team.last).\
deliver_now
```

With all of that new code added, let's go back and ensure our tests will safeguard against bugs, help refractor safely, and generally make you more confident in this codebase.

11.1.7 Testing the Github Integration

As we have been building this application for quite a while, the approaches taken in the following spec files will not be foreign. We will test controllers to ensure they return the correct responses and flash messages. We will test services to ensure they are invoked correctly and process data as expected. We will create specs for ActiveJob classes to ensure they are queued and process incoming arguments and data as expected. Lastly, we will use feature specs to ensure browser behavior exhibits the same information as the more unit-level specs written.

11.1.7.1 Model Specs

With all of that out of the way, we can now add some specs that will take advantage of none of the updates we just used by adding

some model specs. We're going to add four model spec files. One for `Event`, `Events::Github`, `Integration`, and `Integrations::Github`:

While the actual models will be correct, the factories must be adjusted to pass these specs. The model relationships are not automatically inferred in each case when auto-generating the factories. This issue is a simple fix by changing `relationship { nil }` to `relationship`:

spec/factories/events.rb

```
FactoryBot.define do
  factory :event do
    type { '' }
    team
    user_name { 'MyString' }
    user
    event_name { 'MyString' }
    event_body { 'MyText' }
    event_id { 'MyString' }
    event_data { '' }
    event_time { '2020-05-05 23:59:35' }
  end
end
```

and

spec/factories/integrations.rb

```
FactoryBot.define do
  factory :integration do
    account
    type { '' }
    settings { '' }
  end
end
```

Now, onto the spec files:

spec/models/event_spec.rb

```
require 'rails_helper'

RSpec.describe Event, type: :model do
  it "has a valid factory" do
    expect(FactoryBot.build(:event)).to be_valid
  end

  it { is_expected.to belong_to(:user).optional }
  it { is_expected.to belong_to(:team) }
end
```

spec/models/integration_spec.rb

```
require 'rails_helper'

RSpec.describe Integration, type: :model do
  it "has a valid factory" do
    expect(FactoryBot.build(:integration)).to be_valid
  end

  it { is_expected.to belong_to(:account) }
end
```

spec/models/integrations/github_spec.rb

```
require 'rails_helper'

RSpec.describe Integrations::Github, type: :model do
  it "has a valid factory" do
    expect(FactoryBot.build(
      :integration,
      type: 'Integrations::Github'
    )).to be_valid
  end
```

```
  it { is_expected.to belong_to(:account) }
end
```

This syntax is all standard stuff for model specs in this application. We ensure the factories and models are built correctly, then test the `belongs_to` relationships on the models.

11.1.7.2 Controller specs

Next up in line to test will be the two new controllers, `IntegrationsController`, `Users::OmniauthCallbacksController`, and `Events::GithubController`. Plus, we will adjust some of our `TeamsController` specs for the changes made there.

First up, we will want to make a `GithubMocks` module as we did for Stripe to house all of our WebMock stub methods. Let's create the module and the method for the first URL we will need to mock:

spec/support/github_mocks.rb

```
module GithubMocks
  def github_mock_repo_list
    stub = stub_request(:get, %r{https://api.github.com/u\
ser/repos\?access_token=\w*})

    stub.to_return(body: [{ id: '1' }].to_json)
  end
end
```

We will follow that up by adding the necessary lines to the `rails_helper.rb` file:

spec/rails_helper.rb

```
...
require 'support/stripe_mocks'
require 'support/github_mocks' ## new line

...

config.include StripeMocks
config.include GithubMocks ## new line
```

Finally, now that we are back to the specs we'll start with the `Integrations` request spec file:

spec/request/integrations_spec.rb

```
require 'rails_helper'

RSpec.describe 'Integrations', type: :request do
  login_admin

  describe 'GET #index' do
    it 'returns HTTP success' do
      get integrations_path
      expect(response).to have_http_status(:success)
      expect(assigns(:grepos)).to eq(nil)
    end

    it 'returns HTTP success when Github Integration exis\
ts' do
      FactoryBot.create(
        :integration,
        type: 'Integrations::Github',
        account_id: @admin.account.id,
        settings: { token: 'tok123' }
      )
```

```
      github_mock_repo_list
      get integrations_path
      expect(response).to have_http_status(:success)
      expect(assigns(:github)).to be_truthy
      expect(assigns(:grepos)).to be_truthy
    end
  end

  describe 'Delete #destroy' do
    before do
      FactoryBot.create(
        :integration,
        type: 'Integrations::Github',
        account_id: @admin.account.id
      )
    end
    it 'returns HTTP success' do
      delete remove_integrations_path(provider: 'github')
      expect(response).to have_http_status(:redirect)
      expect(flash[:notice]).to eq 'Removed Github integr\
ation. Please besure to remove the application through yo\
ur settings area in Github'
    end

    it 'does not delete integration' do
      delete remove_integrations_path(provider: '!github')
      expect(response).to have_http_status(:redirect)
      expect(flash[:notice]).to eq 'Unable to removed !gi\
thub integration'
    end
  end
end
```

Overall, this spec falls in line with other request specs, where it

sets up a few specific records for the test and then uses those to ensure the controller actions behave as they should through each conditional path. The `index` action will check two logical branches—one without integration. Thus the `@grepos` instance variable will be `nil`. This test is the simpler of the two `index` action specs. The `truthy` based branch will do a few special things. First, it creates an `Integration` on the Team, allowing the Github conditional code to run. Then, it uses the new mock `github_mock_repo_list` to ensure the HTTP API call is mocked and returns the expected data.

The following two specs ensure the `delete` action works as intended. Both specs add an integration. However, the first one requests a delete on an existing integration, and the seconded spec ensures it does not happen when provided an incorrect provider name.

The second controller spec we will add will be for the `Events::GithubController`:

spec/requests/events/github_spec.rb

```
require 'rails_helper'

RSpec.describe "Events::Github", type: :request do
  let!(:team) { FactoryBot.create(:team) }

  describe 'POST #create' do
    it 'signature passes' do
      token = 'sha1=' + OpenSSL::HMAC.hexdigest(
        OpenSSL::Digest.new('sha1'),
        team.id,
        '{}'
      )
      post github_events_path(team), params: {}, headers:\
 { 'HTTP_X_HUB_SIGNATURE': token }, as: :json
      expect(response).to have_http_status(:success)
    end
```

```
    it 'signature fails' do
      token = 'sha1=' + OpenSSL::HMAC.hexdigest(
        OpenSSL::Digest.new('sha1'),
        team.id,
        '23rtegf'
      )
      post github_events_path(team), params: {}, headers:\
 { 'HTTP_X_HUB_SIGNATURE': token }, as: :json
      expect(response).to have_http_status(:unauthorized)
    end

    it 'triggers push event' do
      token = 'sha1=' + OpenSSL::HMAC.hexdigest(
        OpenSSL::Digest.new('sha1'),
        team.id,
        '{}'
      )
      post github_events_path(team), params: {}, headers:\
 { 'HTTP_X_HUB_SIGNATURE': token, 'HTTP_X_GITHUB_EVENT': \
'push' }, as: :json
      expect(response).to have_http_status(:success)
    end

    it 'triggers pull request event' do
      token = 'sha1=' + OpenSSL::HMAC.hexdigest(
        OpenSSL::Digest.new('sha1'),
        team.id,
        '{}'
      )
      post github_events_path(team), params: {}, headers:\
 { 'HTTP_X_HUB_SIGNATURE': token, 'HTTP_X_GITHUB_EVENT': \
'pull_request' }, as: :json
      expect(response).to have_http_status(:success)
    end
  end
```

```
end
```

This request spec has much less going on, even if it may seem like a larger file. The main thing is that we are manipulating a few request headers to ensure we are basing our specs on real-world-like tokens. These same `secret` tokens are used in verifying the signature from the HTTP request matches to verify the input was sent by Github. In this case, we are merely building our token through a method the same way the token is usually created in the Github API. There are four specs within this controller spec. The two specs will test verifying the signature's success and failure through the hashing tools and comparisons. The next two make sure the verification exists, that the signature works out, and that both the `push` and `pull_request`.

The final new request spec will be for the `Users::OmniauthCallbacksController` controller:

spec/requests/users/omniauth_callback_spec.rb

```
require 'rails_helper'

RSpec.describe "/users/omniauth_callback", type: :request\
 do
   describe 'GET #github' do
    login_admin

    before(:each) do
      OmniAuth.config.test_mode = true
    end

    it 'passes auth and returns HTTP redirect' do
      OmniAuth.config.add_mock(:github, { credentials: { \
token: '12345' } })

      get user_github_omniauth_callback_path, headers: { \
'devise.mapping': Devise.mappings[:user], 'omniauth.auth'\
```

```
: OmniAuth.config.mock_auth[:github]}
      expect(response).to have_http_status(:redirect)
      expect(flash[:notice]).to eq 'Github integration ha\
s been added'
    end

    it 'fails auth and returns HTTP redirect' do
      OmniAuth.config.mock_auth[:github] = :invalid_crede\
ntials

      get user_github_omniauth_callback_path, headers: { \
'devise.mapping': Devise.mappings[:user], 'omniauth.auth'\
: OmniAuth.config.mock_auth[:github]}
      expect(response).to have_http_status(:redirect)
      expect(flash[:alert]).to eq "Could not authenticate\
 you from GitHub because \"Invalid credentials\"."
    end
  end
end
```

This spec is here to do two things: test the controller correctly behaves when authentication is valid and correctly fails when it does not. OmniAuth has a test mode you can turn on before a spec with `OmniAuth.config.test_mode = true`. This setting means it will bypass most of the redirection and handle the data coming back with mocked data you set in individual tests.

```
OmniAuth.config.add_mock(:github, {credentials: {token: '\
12345'}})
headers: { 'devise.mapping': Devise.mappings[:user], 'omn\
iauth.auth': OmniAuth.config.mock_auth[:github]}
```

And

```
OmniAuth.config.mock_auth[:github] = :invalid_credentials
headers: { 'devise.mapping': Devise.mappings[:user], 'omn\
iauth.auth': OmniAuth.config.mock_auth[:github]}
```

These two ways to mock the data OmniAuth returns to ensure it matches how we would react to a typical workflow.

The last request spec file we will update is for the `TeamsController` changes:

spec/controllers/teams_controller_spec.rb

```
require 'rails_helper'

  let(:team){ FactoryBot.create(:team) }

...

describe "GET #new" do
  it "assigns a new team as @team" do
    FactoryBot.create(
      :integration,
      type: 'Integrations::Github',
      account_id: @admin.account.id,
      settings: { token: '<<your token>>' }
    )
    github_mock_repo_list
    get new_team_path, params: {}
    expect(response).to have_http_status(:success)
    expect(assigns(:team)).to be_a_new(Team)
    expect(assigns(:github)).to be_truthy
    expect(assigns(:grepos)).to be_truthy
  end
end

describe "GET #edit" do
  it "assigns the requested team as @team" do
```

```
    team = FactoryBot.create(:team, account_id: account.i\
d)
    FactoryBot.create(
      :integration,
      type: 'Integrations::Github',
      account_id: @admin.account.id,
      settings: { token: 'tok123' }
    )
    github_mock_repo_list
    get edit_team_path(team), params: {id: team.to_param}
    expect(assigns(:team)).to eq(team)
    expect(response).to have_http_status(:success)
    expect(assigns(:team)).to eq(team)
    expect(assigns(:github)).to be_truthy
    expect(assigns(:grepos)).to be_truthy
  end
end

...
```

For this spec, I have just included the changes above. If you compare it to the previous version, the two specs would only contain two simple expectations about the status type and `@team` equality. Now, we're making sure integrations exist, using the same mock stub from the `IntegrationsController` and finally ensuring the respective `@github` and `@grepos` instance variables return the correct values.

11.1.7.3 Service specs

Earlier in the chapter, we created a handful of Service Objects to help contain the logic for new team creation, team updates, and managing webhook changes. The first two will be relatively simple. Those specs will use concepts and syntax similar to when it was part of the controller specs. First up is the `Teams::Update` spec:

spec/services/team/update_spec.rb

```
require 'rails_helper'

describe Teams::Update do
  subject(:teams_update) { Teams::Update }

  describe 'create an account' do
    context 'from form input' do
      let(:team) { FactoryBot.create(:team) }
      let(:user) { FactoryBot.create(:user) }
      let(:account) { FactoryBot.create(:account) }
      let(:valid_attributes) do
        {
          name: Faker::Team.name,
          description: Faker::Company.catch_phrase,
          account_id: account.id,
          timezone: 'Arizona',
          has_reminder: true,
          reminder_time: Time.now,
          has_recap: true,
          recap_time: Time.now
        }
      end

      before(:each) do
        user.add_role :admin
      end

      it 'updates a team' do
        expect do
          teams_update.new(team, valid_attributes).update
        end.to change(Team, :count).by(1)
      end

      it 'does not update a team' do
```

```
          account.id = nil
          result = teams_update.new(team, valid_attribute\
s).update
          expect(result.success).to eq false
          expect(result.error).to eq "Validation failed: \
Account must exist, Account can't be blank"
      end
    end
  end
end
```

With the `lets` and the `before` block, we set up data to be saved in the Service Class' `.call` method. Based on the inputs, we are testing if the update method updates a record as expected with good data and does not update with failing data.

The `Teams::Create` spec file will be similar, except that we will include some methods from the CanCanCan gem to make sure there is access to the CanCanCan authorization method inside the Service class(as it would typically be included as a lambda in the real Service Class).

spec/services/team/create_spec.rb

```
require 'rails_helper'
include CanCan::ControllerAdditions

describe Teams::Create do
  subject(:teams_create) { Teams::Create }

  describe 'create an account' do
    context 'from form input' do
      let(:team) { FactoryBot.create(:team) }
      let(:user) { FactoryBot.create(:user) }
      let(:current_user) { user }
      let(:account) { FactoryBot.create(:account) }
      let(:valid_attributes) do
```

```
    {
      name: Faker::Team.name,
      description: Faker::Company.catch_phrase,
      account_id: account.id,
      timezone: 'Arizona',
      has_reminder: true,
      reminder_time: Time.now,
      has_recap: true,
      recap_time: Time.now
    }
  end

  before(:each) do
    user.add_role :admin
  end

  it 'creates a team' do
    expect do
      teams_create.new(
        team, account,
        valid_attributes,
        -> { authorize!(:create, @team) }
      ).create
    end.to change(Team, :count).by(1)
  end

  it 'does not create a team' do
    account.id = nil
    result = teams_create.new(
      team, account,
      valid_attributes,
      -> { authorize!(:create, @team) }
    ).create
    expect(result.success).to eq false
    expect(result.error).to eq "Validation failed: Ac\
```

```
count must exist, Account can't be blank"
      end
    end
  end
end
```

The one significant difference in this spec file versus the one for `Teams::Update` is that the line `include CanCan::ControllerAdditions` gives access to the `authorize` method. Otherwise, the specs are just passing in the team object and its attributes to use in creating the record. The two specs here also test if the attributes allow for the record to be saved or not.

The last Service Class being tested will be the `Webhooks::Github::Manage` class:

spec/services/webhooks/github/manage_spec.rb

```
require 'rails_helper'

describe Webhooks::Github::Manage do
  subject(:manage) { Webhooks::Github::Manage }

  describe 'manage webhooks' do
    context 'with a team' do
      let(:team) do
        FactoryBot.create(
          :team,
          integration_settings: {
            github_repos: ['test|one']
          }
        )
      end

      it 'adds a webhook' do
        integration_settings = {
          'github_repos' => ['test|one', 'test|two']
```

```
        }
        expect(
          Webhooks::Github::AddJob
        ).to receive(:perform_later).once
        manage.new(team, integration_settings).manage
      end

      it 'removes a webhook' do
        integration_settings = {
          github_repos: []
        }
        expect(
          Webhooks::Github::RemoveJob
        ).to receive(:perform_later).once
        manage.new(team, integration_settings).manage
      end
    end
  end
end
```

Due to the simplistic nature of the input and expected behavior, we can run two specs on this class to expect that when the input indicates a webhook was added or removed in saving the team record, the jobs for adding and removing a webhook will be run, respectively. This means that for each spec, we're checking if the job's classes receive a `perform_later` invocation.

11.1.7.4 Job specs

Next up on our list of specs to add will be the spec for the new ActiveJob classes created in this chapter. The main objective here is to ensure the jobs respond to inputs correctly and that we have the correct results. Plus, assure the proper class names and inputs that the jobs get queued to the proper ActiveJob/Sidekiq queues.

The first spec file will be for the `Webhooks::Github::Add` job:

spec/jobs/webhooks/github/add_spec.rb

```
require 'rails_helper'
include ActiveJob::TestHelper

RSpec.describe Webhooks::Github::AddJob do
  let(:account) { FactoryBot.create(:account) }
  let(:team) do
    FactoryBot.create(
      :team,
      account_id: account.id,
      integration_settings: {
        github_repos: ['test|one']
      }
    )
  end

  before(:each) do
    FactoryBot.create(
      :integration,
      type: 'Integrations::Github',
      account_id: account.id,
      settings: { token: '<< your token >>' }
    )
  end

  it 'matches with an enqueued job' do
    Webhooks::Github::AddJob.perform_later(
      ['math_probability|rob-race'],
      team
    )
    expect(Webhooks::Github::AddJob).to have_been_enqueued
  end

  it 'enqueues a default based job' do
    expect do
```

```
      Webhooks::Github::AddJob.perform_later(
        ['math_probability|rob-race'],
        team
      )
    end.to have_enqueued_job.on_queue('default')
  end

  it 'adds a webhook' do
    github_mock_webhook_add('rob-race', 'math_probability\
')
    expect_any_instance_of(Github::Client::Repos::Hooks)
      .to receive(:create).and_call_original
    Webhooks::Github::AddJob.perform_now(
      ['math_probability|rob-race'],
      team
    )
  end
end
```

To get this last spec to work, we will need to add the following mock to the `GithubMocks` file:

```
def github_mock_webhook_add(user, repo)
  stub = stub_request(:post, %r{https://api.github.com/re\
pos/#{user}/#{repo}/hooks\?access_token=\w*})

  stub.to_return(
    body: { config: { url: 'https://awes.ome' } }.to_json
  )
end
```

This file is a typical ActiveJob-based spec file. Check the job gets queued, check the job gets enqueued in the correct queue, and check the result in some fashion after the job gets run. In this case, the

job's results are examined by ensuring the Github API gem calls a `Hook` create call to the API.

Another interesting new concept is the `.and_call_original` on the expectation method chain. Since `expect_any_instance_of` mocks a method call, stopping it from actually being called can sometimes lead to issues or weird states. When you need the `method` on a class that is being mocked actually to run, you add `.and_call_original`.

The spec for `Webhooks::Github::Remove` will be very similar, in fact, using much of the same concepts and setup:

spec/jobs/webhooks/github/remove_spec.rb

```
require 'rails_helper'
include ActiveJob::TestHelper

RSpec.describe Webhooks::Github::RemoveJob do
  let(:account) { FactoryBot.create(:account) }
  let(:team) do
    FactoryBot.create(
      :team,
      account_id: account.id,
      integration_settings: {
        github_repos: ['test|one'],
        github_webhooks: [{
          'id': '123',
          'repo_name': 'math_probability',
          'repo_owner': 'rob-race'
        }]
      }
    )
  end

  before(:each) do
    FactoryBot.create(
      :integration,
      type: 'Integrations::Github',
```

```
      account_id: account.id,
      settings: { token: '<< your token >>' }
    )
  end

  it 'matches with enqueued job' do
    Webhooks::Github::RemoveJob.perform_later(
      ['math_probability|rob-race'],
      team
    )
    expect(Webhooks::Github::RemoveJob).to have_been_enqu\
eued
  end

  it 'enqueues a default based job' do
    expect do
      Webhooks::Github::RemoveJob.perform_later(
        ['math_probability|rob-race'],
        team
      )
    end.to have_enqueued_job.on_queue('default')
  end

  it 'updates a subscription' do
    github_mock_webhook_remove('rob-race', 'math_probabil\
ity', '123')
    expect_any_instance_of(Github::Client::Repos::Hooks)
      .to receive(:delete).and_call_original
    Webhooks::Github::RemoveJob.perform_now(
      ['math_probability|rob-race'],
      team
    )
  end
end
```

Again, to get this final spec to work, we will need to add the following mock to the `GithubMocks` file:

```
def github_mock_webhook_remove(user, repo, hook_id)
  stub = stub_request(:delete, %r{https://api.github.com/\
repos/#{user}/#{repo}/hooks/#{hook_id}\?access_token=\w*}\
)

  stub.to_return(
    body: { config: { url: 'https://awes.ome' } }.to_json
  )
end
```

The main difference between the two webhook job specs is the Team's integration settings information. In the create spec file, we are making sure the `github_webhooks` JSONB node has information within it:

```
let(:team) do
  FactoryBot.create(
    :team,
    account_id: account.id,
    integration_settings: {
      github_repos: ['test|one'],
      github_webhooks: [{
        'id': '<<id from add_webhook cassette>>',
        'repo_name': 'math_probability',
        'repo_owner': 'rob-race'
      }]
    }
  )
end
```

Within the Github API(and the gem we are using), deleting a webhook requires the ID that Github had assigned to the webhook

configuration. In this case, you can use any string you want as we mock the request ourselves. I chose `123` to keep it simple.

We need to do a bit more setup to test the jobs that get run when Github sends a webhook event to our event controllers. To have a proper event payload, we must create a fixture JSON file for each event we listen for(in this case, push events and pull_request events). The JSON payload will be fetched right from the sample payload in Github's documentation, saved as a JSON file on the spec directory structure, and parsed at the beginning of each spec file.

The best way to achieve this is to head over to the Github API documentation for each event payload, copy the JSON and save it as `fixtures/github/<<event>>.json`. I am specifically not adding the JSON to this chapter as it may change, as well as being a large amount of data(which is also tough to format into the book's width.).

Let's start with the `Push` event payload and then the spec file. First, you can visit the event's documentation(at the time of writing) here[4]. You should scroll down a little bit and see the example payload. Copy that JSON and save it into a file, `spec/fixtures/github_events/push_event.json`. Now that we have an example payload, we can add the job's spec file:

spec/jobs/webhooks/github/push_event/record_spec.rb

```
require 'rails_helper'
include ActiveJob::TestHelper

RSpec.describe Webhooks::Github::PushEvent::RecordJob do
  let(:account) { FactoryBot.create(:account) }
  let(:team) do
    FactoryBot.create(
      :team,
      account_id: account.id
```

[4] https://developer.github.com/v3/activity/events/types/#pushevent

```
    )
  end

  let(:json) { JSON.parse(File.read('spec/fixtures/github\
_events/push_event.json')) }

  it 'matches with enqueued job' do
    Webhooks::Github::PushEvent::RecordJob.perform_later(
      json,
      team.id
    )
    expect(Webhooks::Github::PushEvent::RecordJob).to hav\
e_been_enqueued
  end

  it 'enqueues a default based job' do
    expect do
      Webhooks::Github::PushEvent::RecordJob.perform_late\
r(
        json,
        team.id
      )
    end.to have_enqueued_job.on_queue('default')
  end

  it 'creates an event without user in account' do
    expect_any_instance_of(Events::Github).to receive(:sa\
ve!)
    Webhooks::Github::PushEvent::RecordJob.perform_now(
      json,
      team.id
    )
  end

  it 'creates an event without user in account by github \
```

```
username' do
    FactoryBot.create(
      :user,
      account_id: account.id,
      github_username: 'baxterthehacker'
    )
    expect_any_instance_of(Events::Github).to receive(:sa\
ve!)
    Webhooks::Github::PushEvent::RecordJob.perform_now(
      json,
      team.id
    )
  end

  it 'creates an event without user in account by github \
user email' do
    FactoryBot.create(
      :user,
      account_id: account.id,
      email: 'baxterthehacker@users.noreply.github.com'
    )
    expect_any_instance_of(Events::Github).to receive(:sa\
ve!)
    Webhooks::Github::PushEvent::RecordJob.perform_now(
      json,
      team.id
    )
  end
end
```

The data setup on this spec file is minimal compared to other spec files in this chapter or recent chapters. We need to ensure an account is in a team and turn the JSON into a hash(as the job expects as its argument).

As usual, for job spec files, the first two specs test the job is queued

and enqueued to the correct queue name. Then, instead of one spec to test the outcome based on input, three specs test the three different conditional paths. As the user could be in the system through a `github_username` column, the event's user email attribute could match a user in the system, or neither will be met. In the two cases where we are expecting matches against users in the system, we set up users within the spec, which will match the data found in the JSON fixture that was recently added.

The last job spec file to create is the `pull_request` event's job. Here, you can visit the event's documentation(at the time of writing) here[5]. You will scroll down just a little bit and see the example payload. Copy that JSON and save it into a file, `spec/fixtures/github_events/pull_request_event.json`. You will also want to jump into the JSON file and update:

"json "closed_at": null, "merged_at": null,

to

```json
"closed_at": "2015-05-05T23:40:27Z",
"merged_at": "2015-05-05T23:40:27Z",
```

The dates will be used when the Pull Request event is recorded as a closed or merged pull request.

With that out of the way, onto the spec:

[5] https://developer.github.com/v3/activity/events/types/#pullrequestevent

spec/jobs/webhooks/github/pull_request_event/record_spec.rb

```
require 'rails_helper'
include ActiveJob::TestHelper

RSpec.describe Webhooks::Github::PushEvent::RecordJob do
  let(:account) { FactoryBot.create(:account) }
  let(:team) do
    FactoryBot.create(
      :team,
      account_id: account.id
    )
  end

  let(:json) { JSON.parse(File.read('spec/fixtures/github\
_events/push_event.json')) }

  it 'matches with enqueued job' do
    Webhooks::Github::PushEvent::RecordJob.perform_later(
      json,
      team.id
    )
    expect(Webhooks::Github::PushEvent::RecordJob).to hav\
e_been_enqueued
  end

  it 'enqueues a default based job' do
    expect do
      Webhooks::Github::PushEvent::RecordJob.perform_late\
r(
        json,
        team.id
      )
    end.to have_enqueued_job.on_queue('default')
  end
```

```
  it 'creates an event without user in account' do
    expect_any_instance_of(Events::Github).to receive(:sa\
ve!)
    Webhooks::Github::PushEvent::RecordJob.perform_now(
      json,
      team.id
    )
  end

  it 'creates an event without user in account by github \
username' do
    FactoryBot.create(
      :user,
      account_id: account.id,
      github_username: 'baxterthehacker'
    )
    expect_any_instance_of(Events::Github).to receive(:sa\
ve!)
    Webhooks::Github::PushEvent::RecordJob.perform_now(
      json,
      team.id
    )
  end

  it 'creates an event without user in account by github \
user email' do
    FactoryBot.create(
      :user,
      account_id: account.id,
      email: 'baxterthehacker@users.noreply.github.com'
    )
    expect_any_instance_of(Events::Github).to receive(:sa\
ve!)
    Webhooks::Github::PushEvent::RecordJob.perform_now(
      json,
```

```
      team.id
    )
  end
end
```

We want to handle most differences in the different conditional branches for the Pull Request event into the let area to load the JSON event, then change the critical pieces used for determining or saving data. Once again, the first two specs are standard job specs in our application's spec files and are followed by four specs. Two specs check the different conditional paths for whether or not we can match a user from the pull request event payload. The following two specs will test against the sub actions of a closed or merged pull request. This also means the default testing path and default action from the JSON fixture is an opened pull request.

11.1.7.5 System specs

Last but not least is a system spec to ensure that events imported into the application through Github's webhook controller and jobs will display correctly in the browser. We can achieve this within a system spec by adding data before the spec runs. Here is the system spec(which we'll go over):

spec/system/teams/events_spec.rb

```
require 'rails_helper'

RSpec.feature 'Teams', type: :system do
  login_admin

  let(:json) { JSON.parse(File.read('spec/fixtures/github\
_events/push_event.json')) }
  let(:team) { FactoryBot.create(:team, account_id: @admi\
n.account.id) }
```

```
  it "should display a team's events" do
    json['commits'].first['timestamp'] = Time.now.iso8601
    Webhooks::Github::PushEvent::RecordJob.perform_now(js\
on, team.id)

    FactoryBot.create(
      :integration,
      type: 'Integrations::Github',
      account_id: @admin.account.id,
      settings: { token: 'tok123' }
    )

    visit team_path(team)

    expect(page).to have_content('Events')
    expect(page).to have_content(json['commits'].first['m\
essage'].to_s)
  end
end
```

This new file is a relatively minor system spec file, even with all the data setup needed. To review this spec, first, the JSON is loaded in from an event JSON fixture. In this case, we will use the push event as the data is smaller(and faster to lead) and more accessible. From there, we ensure a team is set up with the correct account(and thus the right user). We update the timestamp to ensure the event will be listed on the `show` page, which displays the current day's events and standups. Lastly, the setup will use the `PushEvents` job class to save the event data to a record within the spec's database.

Once all the data is set up, we view the `teams#show` page to see that the text `Events` is there with an event with matching text from the JSON fixture's event payload.

This spec will wrap up our testing on the Github integration and the changes it brought. A quick `rspec` command should show passing

specs and near 100% code coverage.

Now is also a great time to commit, merge, and push your changes to your codebase.

```
git add .
git commit -am "Add Github to the application to allow ca\
pturing events through\
 webhooks"
git checkout main
git merge github-integrations
git push
```

11.2 Analytics: What are they good for?

Analytics can be one of the best tools in a young application's life as it will help paint a picture regarding workflows, high and low-traffic features, and what an end-user might be doing in the application.

The main idea of using an analytics platform is that you will want to track two primary nouns: People and Events. You can be as high-level or granular with both as you would like. For people, you may want to track anonymous user sessions to correlate events to one person. On the other hand, you may follow the account information, user's location, or other bits to try and segment the user behavior through events.

Likewise, you may track the page title or controller action name with the event. However, you can also track other information, such as load time, resource counts, or other application-specific data, to correlate and segment data.

The choice will often depend on your approach and need from analytics. As we go through the chapter, we will add user email

addresses, account names, and plan types on the people side while sending the controller action for the event information. This is usually an excellent place to start inspecting and grouping end-user behavior.

11.2.1 Choices

There are a few choices out there when it comes to picking an Analytics platform. A few you find are Heap Analytics[6], MixPanel[7], or Segment[8].

While most on that list have different strengths and weaknesses, they are similar. They allow you to track events and people in some fashion.

Heap Analytics and MixPanel are very similar in tracking, features, and pricing. The main difference between both is that Mixpanel has been around a bit longer and has a richer set of APKs, specifically, a ruby gem.

Segment is the most interesting on the list as it is the intermediary for the analytical payloads. It allows ingesting data from APKs, gems/packages, and even third parties(such as Stripe). Inside Segment, you can quickly peek at the data coming into your chosen destination, such as email platforms, other analytic tools, and more. This tool can be helpful in a company where analytical data may be used for customer retention by one team, marketing by another, and feature usefulness by yet another group.

For this Chapter, we will be implementing and using MixPanel. MixPanel, at this time, allows 20 million events and 1000 people to be added to the free tier and has a rubygem to track events through the server side.

[6] https://heapanalytics.com
[7] https://mixpanel.com
[8] https://segment.com/

You can start by signing up through here[9]. This process should set you up with their free plan, asking only for basic login and identifiable information.

Once signed up, you can access a 'snippet' of javascript to add. We can safely skip that and instead go towards using their Ruby API.

11.2.2 Implementing and Sending events

Let's quickly add a new branch to handle the following code updates:

```
git checkout -b analytics
```

To add the RubyGem for Mixpanel, add `gem 'mixpanel-ruby'` to the Gemfile, under the `github_api` line, and run `bundle install`.

With the gem installed, we must grab the token from Mixpanel. This can be done by clicking on the icon in the right corner that looks like a gear. Then, click "Project Settings". You should see a line item "Project Token".

Just like we added Stripe and Github tokens, we will add another entry for Mixpanel in the Encrypted Credentials. It should look something like:

```
EDITOR="atom --wait" rails credentials:edit
```

Once open in an editor, you can add at the end of the file:

```
mixpanel:
  token: <<your token>>
```

You can verify that the credentials were saved by using the Rails console:

[9] https://mixpanel.com/register

```
Rails.application.credentials.mixpanel
 => {:token=>"token"}
```

To make use of the token, we will initialize a MixPanel tracker with the `token`:

config/initializer/mixpanel.rb

```
TRACKER = Mixpanel::Tracker.new(Rails.application.credent\
ials.mixpanel[:token])
```

We use a constant as the most efficient way to store a static variable within a Ruby program. The constant tells Ruby, "I am going to set this and never change it. Store it efficiently in memory".

With the gem, token, and tracker all setup, we can begin to create the ActionJob classes that will handle `person` identification and `event` tracking. The `person` identification class will take a user as input and identify the user to Mixpanel:

app/jobs/analytics/mixpanel/identify_person_job.rb

```
module Analytics
  module Mixpanel
    class IdentifyPersonJob < ApplicationJob
      def perform(user)
        TRACKER.people.set(user.id,
                           {
                             '$name': user.name,
                             '$email': user.email,
                             '$timezone': user.time_zone,
                             '$account_name': user.accoun\
t.name,
                             '$account_plan': user.accoun\
t.product_id
                           })
      end
```

```
    end
  end
end
```

You may notice that this is a simple job class. It uses a `User` object passed in to pass along the User's name, email address, timezone, account name, and account's `product_id`. The `Tracker.people.set` method chain is provided by MixPanel to automatically request their `People` endpoints to save the data.

Next up will be the job class that will save the events to MixPanel:

app/jobs/analytics/mixpanel/send_event_job.rb

```
module Analytics
  module Mixpanel
    class SendEventJob < ApplicationJob
      def perform(user, event_name, event_data)
        TRACKER.track(user.id, event_name, event_data)
      end
    end
  end
end
```

Here is an even simpler job class that uses the user passed in for an identifier(which at this time would have its `people` profile set), `event_name`, and a hash of data. `TRACKER.track` is provided by the MixPanel gem to send an event tracking request to the MixPanel API.

Now that our background jobs are created, we can add their use to the application to be triggered when needed. To identify a user, we can send the `identify` request as many times as needed and will use the most recent request. Thus, the most logical place to add identifiable requests would be right after a successful login. Luckily, Devise provides a model-level method that gets called whenever a successful login occurs. To use this to trigger a MixPanel Person identification, add the following to the `app/models/user.rb` file:

```
class User < ApplicationRecord

  ...

  def after_database_authentication
    Analytics::Mixpanel::IdentifyPersonJob.perform_later(\
self)
  end
end
```

Thus, every time a login happens, `after_database_authentication` is invoked, and then the `Analytics::Mixpanel::IdentifyPersonJob.perform_later` is defined within. In this case, `self` refers to the `User` that has just signed in successfully.

Lastly, with the identification handled, we will just need to send events after every controller action is invoked. We can achieve this by adding an `after_action` to the `ApplicationController`. This means that after any action is finished, the method specified will run:

app/controllers/application_controller.rb

```
class ApplicationController < ActionController::Base

  ...

  after_action :send_to_analytics

  ...

  protected

  ...

  def send_to_analytics
    return if devise_controller? || controller_name.in?(%\
```

```
w[github webhook])

    Analytics::Mixpanel::SendEventJob.perform_later(
      current_user,
      'Page View',
      controller: controller_name,
      action: action_name
    )
  end

  ...

end
```

When we add `after_action :send_to_analytics`, it will call that method after every request. Inside that method, we will use a return to exit the method without sending an event if it is a `devise_controller?`. These conditionals help us exclude all unauthenticated requests. We additionally want to skip sending events if the controller is used for Stripe and Github events, as they are also unauthenticated.

The job classes combined with the User model and ApplicationController additions will allow the MixPanel requests to make it to your MixPanel project to be recorded. This technique will enable you to view anything from a chronological list of events to segmenting based on `person` or `event` data.

We will want to add a few specs to finish adding Analytical tracking to the application. By interacting with a model and controller that gets used by nearly every other spec, the existence of the added code to those files is tested throughout the test suite. Thus, we will want to add a spec for each new job file.

Let's start with the `SendEventJob` class:

spec/jobs/analytics/mixpanel/send_event_job_spec.rb

```
require 'rails_helper'
include ActiveJob::TestHelper

RSpec.describe Analytics::Mixpanel::SendEventJob do
  let(:user) { FactoryBot.create(:user) }

  it 'matches with enqueued job' do
    Analytics::Mixpanel::SendEventJob.perform_later
    expect(Analytics::Mixpanel::SendEventJob).to have_bee\
n_enqueued
  end

  it 'enqueues a default based job' do
    expect { Analytics::Mixpanel::SendEventJob.perform_la\
ter(user, '', {}) }
      .to have_enqueued_job.on_queue('default')
  end

  it 'receives a track invocation onto the Tracker' do
    expect(TRACKER).to receive(:track)
    Analytics::Mixpanel::SendEventJob.perform_now(user, '\
', {})
  end
end
```

This spec file is very nearly like most other job spec files. We're testing for the same things:
A job is queued.
A job is queued to the correct ActiveJob queue.
The intended Class method is invoked.
That is no different here, as the queues have a spec each, along with the last spec, making sure `track` is called on the `Tracker` class.

Finally, the `IdentifyPersonJob` class is nearly identical:

spec/jobs/analytics/mixpanel/identify_person_job_spec.rb

```
require 'rails_helper'
include ActiveJob::TestHelper

RSpec.describe Analytics::Mixpanel::IdentifyPersonJob do
  before(:each) do
    ActiveJob::Base.queue_adapter = :test
    @user = FactoryBot.create(:user)
  end

  it 'matches with enqueued job' do
    ActiveJob::Base.queue_adapter = :test
    Analytics::Mixpanel::IdentifyPersonJob.perform_later
    expect(Analytics::Mixpanel::IdentifyPersonJob).to hav\
e_been_enqueued
  end

  it 'enqueues a default based job' do
    ActiveJob::Base.queue_adapter = :test
    expect { Analytics::Mixpanel::IdentifyPersonJob.perfo\
rm_later(@user) }
      .to have_enqueued_job.on_queue('default')
  end

  it 'receives a people.set invocation onto the Tracker' \
do
    expect(TRACKER).to receive_message_chain([:people, :s\
et])
    Analytics::Mixpanel::IdentifyPersonJob.perform_now(@u\
ser)
  end
end
```

The main difference in this spec file is the job class it is testing and the method chain expected to be invoked on the `TRACKER` class,

`.people.set`.

Let's go ahead and merge the changes into the main branch to push up to the remote repository:

```
git add .
git commit -am "Add MixPanel to track analytics"
git checkout main
git merge analytics
git push
```

11.3 What did you learn in this chapter?

- Using Third-Party APIs to collect end-user interactions or data for application use.
- Allowing users to authorize a Rails application to access their data through OAuth via the OmniAuth gem and strategies.
- Store unstructured data in a JSONB column
- Mock external HTTP requests through Webmock and VCR
- Collect analytical information through a third-party analytics platform.

11.4 Exercises

Exercise answers can be found with any package or bundle purchases of Build A SaaS App in Ruby on Rails 7[10]!

1. The `PushEvent` job allows a github_username column to match the push events payload information. However, there

[10] https://buildasaasappinrails.com

is no interface for the user to add the information. Add the information to the user settings form and an alert/flash message to advise when it is missing.

2. Analytics such as MixPanel is not only used for the same event with different information. The other business logic process may send their events as well. Add a new event when a new admin user registers.

12. Speeding up Rails

If you have been around the programming or web development world, you may have heard Ruby or Rails are slow. This commentary is partially correct but does need some context.

Firstly, let's get it out of the way that Ruby, a dynamic language with duck typing, will be slower than other compiled and statically typed programming languages. Compilers can turn the developer's code into machine-level code to provide top performance. On a pure performance level, a language like Golang will blow Ruby out of the water nearly every time on every benchmark test.

However, there are other parts of the puzzle when building applications. The development experience between building your application from scratch in Golang versus having Ruby on Rails means you can get to a releasable state many times faster.

This preface brings us to what is "fast enough"? Can you achieve that with Rails? The answer to this is "yes". To build a SaaS application, Rails will be quick enough. Assuming you are not trying to find the 1000th number in the Fibonacci Sequence on a web page. However, this only leaves a little margin for error when making a quick application.

This chapter may not be huge concerning the number of pages, but it will pack some of this book's most "bang for your buck" advice.

12.1 What Slows Rails Down?

Several areas can either slow a Rails application down by mistake or speed a Rails application up through configuration or usage. Going

through the following in most Rails applications will yield results that speed up your Rails application in some manner.

These slowdowns and speedups can fall under one of two categories, "getting data" or "displaying data". You may be slowing your application down with slow queries to the database or slowing your application down by computing markup more often than you need to do.

One of the better things you can do at the start is to add a profiling mechanism to your application. One of the most popular ways to do so now is to use `RackMiniProfiler`.

`RackMiniProfiler` is a gem that is a piece of Rack Middleware(aka request classes that sit between the web server and Rails-specific code) that profiles the web request. It also nicely adds a badge to your development session to allow you to introspect the speed and timing of your application.

To install, add the following to your Gemfile and `bundle install`

```
gem 'rack-mini-profiler'
```

From their README:

> NOTE: Be sure to require rack_mini_profiler below the pg and MySQL gems in your Gemfile. rack_mini_profiler will identify these gems if they are loaded to insert instrumentation. If included too early no SQL will show up.

This gem install should be all the setup and configuration you need in Rails. `rack_mini_profiler` will automatically work in the development environment.

12.1.1 N + 1 Queries

One of the first places you want to look to find quick performance gains would be to seek and eliminate N+1 queries.

What is an N+1 query? When using an ORM such as ActiveRecord, the associations are lazily loaded, meaning the ORM does not grab the data upfront but when it is later needed(i.e., called by a template). While this can help data usage and efficiency, there are some cases when calling out to associated attributes will make the ORM go back to the database for each iteration. Hence the N+1 name.

How do we fix it? The quick answer is eager_loading. When you eager load, you are telling the ORM (ActiveRecord) to fetch the data beforehand, in one query to then fetch the data from memory (the ActiveRecord association Array).

How is this done in Rails/ActiveRecord? When you want to eager load data, you chain the `.includes` method into the query. For example, let's say you wanted to display all of the user's names on a particular page. If you iterated on the user's association on the account (`account.users`), you would experience the N+1 query. However, if instead, you used `.includes` on the account users query (`account.includes(:users).users`), it would perform one `User` lookup for all of the associated users.

To help locate N+1 queries in your application, we will add the Bullet[1] gem. Let's add `gem 'bullet'` in the `group :development, :test do`, which will make the gem's functionality available to the development and testing environments. However, Bullet is set up only to begin working once configured in the environment configuration files.

To enable bullet and have it log warnings to the Rails logger(and its logfile), run the following command that will automatically add the following to the `config/environments/development.rb` file:

[1] https://github.com/flyerhzm/bullet

```
bundle exec rails g bullet:install
```

There are actually around 15 different settings you can configure Bullet with, and you can review them to choose the settings that best fit your development style here: Bullet Doc[2]

Now, assuming there is an N+1 somewhere within the application, you will see a log entry like:

```
2009-08-25 20:40:17[INFO] N+1 Query: PATH_INFO: /;    mod\
el: Standup =>
associations: [tasks]·
Add to your finder: :include => [:tasks]
```

Note: We had already done some work setting the queries up earlier in the book to eager-load all task types on the Standup queries.

Additionally, you can have Bullet raise errors in your tests by adding the following to the `config/environments/test.rb` file:

```
config.after_initialize do
  Bullet.enable = true
  Bullet.bullet_logger = true
  Bullet.raise = true
end
```

And enable Bullet per each spec in `spec/spec_helper.rb`:

[2] https://github.com/flyerhzm/bullet#configuration

```
if Bullet.enable?
  config.before(:each) do
    Bullet.start_request
  end

  config.after(:each) do
    Bullet.perform_out_of_channel_notifications if Bullet\
.notification?
    Bullet.end_request
  end
end
```

However, having this run in the test suite is a personal preference and not my preference.

12.1.2 Queries without indices

Another database-related area of performance improvement will be the database indices. What is a database index? In its most simplistic explanation, a database index is a copy of a subset of the table's columns to allow faster lookups. An example would be when you often select a `standup` by its `standup_date`, and there is no index. It would need to scan the whole table sequentially to find any relevant records. However, when there is an index, the database can find records optimally (without explaining what a `btree` index is in computer science), thus limiting the time it takes to return records and the strain on the database.

When adding tables through Rails migrations, columns with the type `references` are automatically set up with an index. Additionally, right after creating a new model, you can also jump into the migration file and add:

```
add_index :table_name, :column_name
```

The migration file also allows us to use different index types(such as `:gin`, which is used in JSONB indices) or multiple column indices(`[:col1, :col2]`) when a lookup is often done with a few different `.where` constraints.

Again, there is a tool to help seek and fix the issue of missing indices called lol_dba[3].

Here's how to use it:

1. Add `gem "lol_dba"` to your `group :development do` block.
2. `bundle install`
3. `bundle exec rake db:find_indexes`
4. Read output on what a migration would look like to add missing indices:

> Note: I ran into an issue where the gem would not run its checks due to a missing relation from "ActionText" (Rich Text fields). To resolve this, I needed to run the ActionText installer, `bin/rails action_text:install`, skip modifying spec helper files, run the migration `bin/rails db:migrate`, and was finally able to run the LOLDBA rake command.

Finally, you can add and run the migration with `rake db:migrate_sql`.

Adding or adjusting indices is another quick win for your application's performance.

12.1.3 Pagination

An often overlooked way to speed up an application is limiting the number of records returned to a page. Why would this help?

[3] https://github.com/plentz/lol_dba

For starters, an object is stored in memory when querying through ActiveRecord. While this doesn't make much of a problem for a handful of records, if you were returning 1000s of records in a query, it would get slow and cause memory issues on the server running the application. Additionally, you need to render all of the records you have returned from the query, which will then slow the server rendering the HTML and the browser parsing the HTML.

Pagination(splitting information into multiple pages) helps combat this by using a `page` and `limit` on any page where you expect to render a collection of items. In this case, think of a page listing standups per person, which is not currently limited by the date. When the end-user starts using the app, they will have a handful of standups, but over time, that number will grow from tens to hundreds of standups. If you are still trying to picture pagination, think of an e-commerce website with hundreds of products showing only ten per page, with numbered links to continue perusing the products.

So, how will we implement pagination? There are a few solutions, but the `pagy` gem is one of the best for handling pagination. The idea here is that Pagy will help you use a page number and an optional per-page value to traverse a collection of data with the database's implementation of `limit`(number of records returned) and `offset`(where to start the counting from).

Here is how we can add Pagy to the application. On pages where there may be a large number of standups:

First, add `gem 'pagy'` if it was not already added earlier in the book, and run `bundle install`.

Following that, there will be three major areas to update: The `routes.rb` file, controller actions that will paginate will add a method that takes the database records as an argument, and the views will be slightly updated.

The addition of the routes will be to allow optional URL paths and parameters to be used on paginated pages. We will be updating the user's standups route and the "my activity" route:

```
scope 'account', as: 'account' do
  resources :plans, only: [:index, :show], controller: 'a\
ccounts/plans'
  resources :users do
    member do
      get 's', to: 'users#standups', as: 'standups'
      get 's/(p/:page)', action: :standups ## <-- New rou\
te
    end
  end
end


get 'activity/mine'
get 'activity/mine/(p/:page)', to: 'activity#mine' ## <--\
 New route
```

Following the route changes will be the new initializer. Per Pagy's documentation, the best way to grab this file will be to get a version-specific link from their documentation here[4]. As stated in their doc, place this file in the `config/initializers/pagy.rb` file. Finally, change a single line to "6" for our use case as the default page size: `Pagy::DEFAULT[:items] = 6`.

Another piece of setup will be to add the following to the `ApplicationHelper` to supply the helper methods for the views:

app/helpers/application_helper.rb

```
module ApplicationHelper
  include Pagy::Frontend


...
```

Next, is updating the `index` action on the `Users::StandupsController` and the `mine` action on the `ActivityController` to include the paging methods:

[4] https://ddnexus.github.io/pagy/quick-start/#configure

```
class Users::StandupsController < ApplicationController
  include Pagy::Backend ## <-new line

  def index
    @standups = user
      .standups
      .includes(:dids, :todos, :blockers)
      .references(:tasks)
      .order('standup_date DESC')
    @pagy, @records = pagy(@standups) ## <-new line
  end


class ActivityController < ApplicationController
  authorize_resource :class => "ActivityController"
  include Pagy::Backend ## <-new line

  def mine
    @standups =
      current_user
      .standups
      .includes(:dids, :todos, :blockers)
      .references(:tasks)
      .order('standup_date DESC')
    @pagy, @records = pagy(@standups) ## <-new line
  end
end
```

Then, we can add a few lines from the Pagy docs for TailwindCSS to style their pagination markup within TailwindCSS design.

app/assets/stylesheets/application.tailwind.css

```
@tailwind base;
@tailwind components;
@tailwind utilities;

/*

@layer components {
  .btn-primary {
    @apply py-2 px-4 bg-blue-200;
  }
}

*/

/* New lines lower */

/*
   Add the following markup AFTER your import statements
   Notice: this style contains only the rules for pagy-nav
*/

.pagy-nav.pagination {
  @apply isolate inline-flex -space-x-px rounded-md shado\
w-sm;
}

.page.next a {
  @apply relative inline-flex items-center rounded-r-md b\
order border-gray-300 bg-white px-2 py-2 text-sm font-med\
ium text-gray-500 hover:bg-gray-50 focus:z-20;
}

.page.prev a {
  @apply relative inline-flex items-center rounded-l-md b\
```

```
order border-gray-300 bg-white px-2 py-2 text-sm font-med\
ium text-gray-500 hover:bg-gray-50 focus:z-20;
}

.page.next.disabled {
  @apply relative inline-flex items-center rounded-r-md b\
order border-gray-300 bg-slate-100 px-2 py-2 text-sm font\
-medium text-gray-500 hover:bg-gray-50 focus:z-20;
}

.page.prev.disabled {
  @apply relative inline-flex items-center rounded-l-md b\
order border-gray-300 bg-slate-100 px-2 py-2 text-sm font\
-medium text-gray-500 hover:bg-gray-50 focus:z-20;
}

.page a, .page.gap {
  @apply bg-white border-gray-300 text-gray-500 hover:bg-\
gray-50 relative inline-flex items-center border px-4 py-\
2 text-sm font-medium focus:z-20;
}

.page.active {
  @apply z-10 border-indigo-500 bg-indigo-50 text-indigo-\
600 relative inline-flex items-center border px-4 py-2 te\
xt-sm font-medium focus:z-20;
}
```

Finally, we will update the markup that displays a list of these standups. This markup will be the view helper that shows the number of pages and the links they can use to navigate between them.

In the file `app/views/users/standups/index.html.erb` change:

```
<%= render partial: 'standups/standup', collection: @stan\
dups, locals: {user: current_user} %>
```

to

```
<%= render partial: 'standups/standup', collection: @reco\
rds, locals: {user: current_user} %>
<div class="my-3 float-right">
  <%== pagy_nav(@pagy) %>
</div>
```

Then, in the file `app/views/activity/mine.html.erb` change:

```
<div class="grid grid-cols-3 auto-cols-auto gap-4" id="<%\
= dom_id(current_user, 'standups') %>">
  <%= render @standups, user: current_user %>
</div>
```

to

```
<div class="grid grid-cols-3 auto-cols-auto gap-4" id="<%\
= dom_id(current_user, 'standups') %>">
  <%= render @records, user: current_user %>
</div>
<div class="my-3 float-right">
  <%== pagy_nav(@pagy) %>
</div>
```

It's the same deal here; the last three lines include the pagination markup. As an aside, the pagination markup will only display if there are enough items to page upon. So, one of the page results would need to be seven or more standups(which you may have already added by this point while playing around with your application).

Lastly, we will adjust the respective Request specs to ensure the results are paged. First, we will add another spec that makes sure after seven standups are created, only six are returned to the `@records` variable:

spec/requests/users/standups_spec.rb

```
require 'rails_helper'

RSpec.describe "/users/standups", type: :request do
  login_admin

  describe "GET #index" do
    it 'assigns all standups as @standups' do
      standup = FactoryBot.create(:standup, user_id: @adm\
in.id)
      get standups_account_user_path(@admin.id)
      expect(assigns(:user)).to eq(@admin)
      expect(assigns(:standups)).to eq([standup])
    end

    ## New Spec
    it 'pages the standups' do
      7.times.each do |i|
        Standup.create(
          user_id: @admin.id,
          standup_date: (1+i).days.ago.iso8601
        )
      end
      get standups_account_user_path(@admin.id)
      expect(assigns(:records).size).to eq 6
    end
  end
end
```

As you can see, seven standups are needed to trigger pagination, and the query will only return six standups per the pagination

methods. This change is tested by accessing the `assigns(:records)` array size. Now we'll do the same with the `Activity` request specs:

spec/requests/activity_spec.rb

```
require 'rails_helper'

RSpec.describe "Activities", type: :request do
  login_user

  describe "GET /mine" do
    it "returns http success" do
      get "/activity/mine"
      expect(response).to have_http_status(:success)
    end

    it 'pages the standups' do
      7.times.each do |i|
        Standup.create(
          user_id: @user.id,
          standup_date: (1+i).days.ago.iso8601
        )
      end
      get "/activity/mine"
      expect(assigns(:records).size).to eq 6
    end
  end

...

end
```

Lastly, if you were to run all of your specs, you would see the view spec for the `activity#mine` view fails due to not having access to the pagination helper methods. This problem can be solved with the following update to the `before` block in the spec, but it does

raise the question of how effective a view spec can be in the suite of specs. Here is the new `before` block:

```
before do
  allow(view).to receive(:current_date)
    .and_return(Date.today.strftime('%a %d %b %Y'))
    allow(view).to receive(:current_user)
    .and_return(FactoryBot.create(:user))
  assign(:records, [FactoryBot.create(:standup)])
  assign(:pagy, Pagy.new(count: 1))
end
```

Here, instead of how it was previously mocking FactoryBot-created standups into the `@standups` variable, we are now mocking the `records` and `pagy` variables that the view uses.

Breaking up large collections by groups of data helps speed up the user experience by reducing the size of the data returned by the database, creating fewer Ruby ActiveRecord objects, and rendering less markup on a page. While it may not make a massive difference with collections of 5, 10, or 20 items, the difference will undoubtedly be felt on selections of 50, 100, or more, especially with complex objects with associations such as the Standup objects in this application.

12.1.4 Gzip

This tool may be one of the quickest performance wins in your Rails application. You can use a Rack Middleware that is built into every Rails codebase to compress string responses sent to the browsers, which are almost every type(HTML, JSON, etc.) of content. The compressed strings will have smaller file sizes and, thus, a quicker loading time from the server to the browser.

Here's how to add it!

1. In `config/application.rb` add the following inside the configuration block, `config.middleware.use Rack::Deflater`.
2. That's it.

From this point forward, any browser environment will receive compressed content, saving end users bandwidth and time!

12.1.5 Caching

Adding caching is one of the most performance-gaining and more involved methods for speeding up a Rails app. Caching is storing data or HTML resulting from a Request/Response cycle to be used again later. In more direct terms, you can store some of the response, such as a portion of the markup in another system(Redis, file system, etc.), to be quickly retrieved instead of performing the entire cycle again. Most times, this will cut out re-rendering or an SQL query.

There are a few different Cache stores that are available to use out of the box from Rails:

- There is the file system, which means Rails will write temporary files to store HTML or other marshaled content. The advantage here is that it is the default and simple to use. However, file-based Rails caching is slow when coupled with the often-used key-based cache expiration.
- A different store type is MemoryStore. MemoryStore directly utilizes storing the data in memory, AKA RAM. This storage type is FAST. However, it doesn't necessarily scale well with most application hosting for two main reasons. It won't be shared across server processes and adds to the RAM usage of an application, which is a common constraint on hosting options.

- Another cache-store option is Redis. This option is usually almost as quick as memory storage and can help guard against the shortcomings of the above stores. First, Redis is a distributed key-value data store, making it available to all processes. It also allows different cache eviction policies, but the LRU method should work for the expiration used in Rails applications(it would come in handy if you are a cache/store expert to utilize the best cache policies needed for your specific data needs). If there was one disadvantage, it is that when using Redis in production, the Redis server will be in another process or server, leaving some minor susceptibility to network or process problems and latency.
- Finally, the last method is MemCache(a distributed cache store), which is nearly identical to the advantages/disadvantages of Redis. However, most applications already run Redis for other uses(ActiveJob or ActionCable). Redis is also currently under more development compared to MemCache.

While the store choice can be more up to the developer or application use cases, it is often best to use the Redis store approach.

There are two main things that can be cached: objects and markup. If you were to look through the Rails documentation, you might see it referred to as "Low-Level Caching". The idea here is that you will write a cache fetch block, where if there is a hit in the store for the cache's key, it will return the data from the store. However, if there is a miss, it will execute the code inside the block and store that data. The downfall of low-level caching is the method by which it expires the data. When using the fetch method, you will include an `expires:` argument that will accept a form of time such as `5.minutes` or `1.day`. Depending on the data, this may not be very flexible. This approach can be used when data changes infrequently, and you can safely assume an expiration period slightly shorter than expected. Another time it may be helpful is when up-to-date data is less critical than speedy or performant responses.

The second thing that can be cached is markup. You will commonly see this referred to as fragment caching or Russian doll caching when the fragments are nested within each other(like Russian dolls). The idea here is that after all other speed enhancements are taken care of, the rendering of markup is usually the slowest portion of the request. If we could just cut to the chase and retrieve the markup from a cache store instead of generating it on every request, it would make responding to the end user much quicker. By default, fragment caching uses the records ID and update_at timestamp to generate a cache key. Thus, once the record is updated, the cache key misses, creating a new cache. Also, the markup's contents are considered by using the hash of the markup content in the cache key. This consideration means that if the markup itself changes(such as an update to the template), all existing fragments using that template will be invalid and expired.

Another fragment caching feature is caching with `collection` partial rendering. For example, if you were rendering todos in the following way:

```
<%= render partial: 'todo', collection: standup.todos %>

// In the _todo.html.erb
<% cache todo %>
  <%= todo.name %>
<% end %>
```

All of the todos would have their markup cached and fetched individually. However, adding the `cached: true` argument to the render line will fetch all cached items at once, handling cache misses automatically and is faster.

12.1.6 How to Implement Caching

To use caching with Redis and Rails, you will need to configure Rails to use Redis as a cache store:

In the development configuration file `config/environments/development.rb`, you will replace the `config.cache_store` with:

```
config.cache_store = :redis_cache_store, { namespace: 'ca\
che' }
```

The settings are pretty simple; it uses the Redis store, pointing to a local version of Redis with a Redis namespace.

Lastly, Rails allows you to toggle caching locally with a command easily:

```
bin/rails dev:cache
```

Note: This command will typically trigger the Rails server to restart by itself, but if you do not notice that happening, a server restart is needed to utilize the above changes.

Now for the fun part, actually implementing caching. First, let's take a stab at Rails object caching. An excellent place to use this can be the Github Repo requests. The list of repositories is usually not updating very much and is an external request that can benefit from being retrieved from a cache instead of GitHub's API each time.

Inside the `IntegrationsController`, we can update the `@grepos` assignment to `Rails.cache.fetch`:

```
@grepos = Rails.cache.fetch(
  "github.integration.repos.#{current_account.id}",
  expires_in: 15.minutes
) do
  client.repos.list.body.map(&:name)
end
```

The main component of this is the first argument, aka the cache key. Here, we are using a period-delimited string, which is not required

but is somewhat of a convention I have seen in most code bases, with the last section being the most critical. We are interpolating the current_user's account hash_id. This key means anyone accessing this action under the same account will benefit from a possible cache hit. The second argument states that the cache will expire in 15 minutes. This time frame is an arbitrary choice that the developer can make. Fifteen minutes seems like a good middle ground, but depending on user behavior, something much higher, like hours or days, may make sense.

You can add a similar call in the `TeamsController`, however as the output is slightly different as a Struct instead of an array, you will need to either use a different cache key for a new cache entry or modify the Struct mapping to read for a variable created by the result of a `Rails.cache.fetch`:

```
@grepos = Rails.cache.fetch(
  "github.teams.repos.#{current_account.id}",
  expires_in: 15.minutes
) do
  client.repos.list.body.map do |r|
    RepoInfo.new(name: r.name, owner: r.owner.login)
  end
end
```

Or using the shared repo approach:

```
repos = Rails.cache.fetch(
  "github.repos.#{current_account.id}",
  expires_in: 15.minutes
) do
  client.repos.list.body
end

@grepos = repos.map do |r|
  RepoInfo.new(name: r.name, owner: r.owner.login)
end
```

With this method you can go back to the `IntegrationsController` and utilize the same cache key, moving the map out of the `cache.fetch`:

```
repos = Rails.cache.fetch(
  "github.repos.#{current_account.id}",
  expires_in: 15.minutes
) do
  client.repos.list.body
end
@grepos = repos.map(&:name)
```

The other cache type you can use is fragment caching. A great place to implement a Russian Doll cache approach would be a list of Standups. You can cache each `Task` type's markup and roll that up into a cache for each standup. When any of the items are updated, you can `touch` the parent object to invalidate the cache.

To make sure that creations, changes, and deletions of Task types trigger an update to a Standup, which will invalidate the top-level standup fragment cache, we will want to add an `after_commit` hook to our Task model:

```
after_commit { standups.find_each(&:touch) }
```

This approach is different from what most tutorials or documentation guides might show. You will usually see `belongs_to :standups, touch: true`. The issue is that we are using a `has_many through:` association here that adds a level of association that Rails can not automatically infer in a `touch`. Thus, after the database commits the changes, we manually look up any standup that Task might be associated with and call the `touch` manually.

With that in place, we can update the markup in the `_standup` template partial:

app/views/standups/_standup.html.erb

```
<%= turbo_frame_tag dom_id(standup), class:"w-96 block p-\
6 bg-white rounded-lg border border-gray-200 shadow-md" d\
o %>
  <div>
    <div class="float-right">
      <% if standup.user_id == local_assigns[:user]&.id %>
        <%= link_to edit_standup_path(standup) do %>
          <%= heroicon "pencil-alt", class: 'inline' %>
        <% end %>
        <%= button_to standup_path(standup), {form_class:\
 'inline', method: :delete, form: {data: { turbo_confirm:\
 "Are you sure? Deleting this standup is permanent"}}} do\
 %>
          <%= heroicon "trash", class: 'inline' %>
        <% end %>
      <% end %>
    </div>
    <h3 class="font-bold leading-tight text-gray-900 text\
-xl"><%== standup_header standup, local_assigns[:caller]%\
></h3>
  </div>
  <h4 class="mt-4 font-bold leading-tight text-gray-900 t\
```

```
ext-lg">Dids</h4>
  <ul>
    <% standup.dids.each do |did| %>
      <% cache did do %>
        <li class="bg-gray-300 border border-gray-100 p-2\
 rounded shadow-sm"> <%= did.title %> </li>
      <% end %>
    <% end %>
  </ul>

  <h4 class="mt-4 font-bold leading-tight text-gray-900 t\
ext-lg">Todos</h4>
  <ul>
    <% standup.todos.each do |todo| %>
      <% cache todo do %>
        <li class="bg-gray-300 border border-gray-100 p-2\
 rounded shadow-sm"> <%= todo.title %> </li>
      <% end %>
    <% end %>
  </ul>

  <h4 class="mt-4 font-bold leading-tight text-gray-900 t\
ext-lg">Blockers</h4>
  <ul>
    <% standup.blockers.each do |blocker| %>
      <% cache blocker do %>
        <li class="bg-gray-300 border border-gray-100 p-2\
 rounded shadow-sm"> <%= blocker.title %> </li>
      <% end %>
    <% end %>
  </ul>
<% end %>
```

As you can see, the small additions are the `cache *object* do` lines that take the markup as a block. It utilizes Rails' built-in key

generation to use the object's record ID and update_at timestamps to handle key expiration. This means the first visit to the page will cache all of the fragments: all Task-type markup and the standups above them. The next visit will read from the cache for each fragment. Then, let's say you add a Blocker for one of the standups, returning to the `_standup` template will only rewrite the fragment for the Standup the Blocker was in and write a new fragment for the new Blocker.

A great thing about this cache layer is that it will work for the template being rendered anywhere. This means that, let's say, ActionCable is using an `ActionController.render` in a background job. It, too, will use the cache fragments, speeding up the jobs and lessening the resources needed to generate the markup passed through the cable's broadcast.

You use the same fragment caching techniques in other templates to cache the standups for maximum speed benefits. However, you should be careful if there are slight differences when displaying the data, such as a standup's box may display different header information. In that case, you can use either `cache` after that data or manually set specific keys for specific types of content.

12.2 What did you learn in this chapter?

- What can slow Rails down, and what conventional tools can be used to diagnose the speed at which your application is working?
- How to look for N+1 queries and how to fix them.
- How to locate queries that may not exist and how to add those queries after the model was created.
- How to add pagination to allow paged data in your application to chunk up the data for a better and quicker user

experience.

- Understand caching, the different cache stores in Rails, and how to implement low-level and fragment caching.

13. Deployments

13.1 There are many choices

Building a Rails application is just one part of a SaaS application. It may have taken a while to get here, but we're in the home stretch.

There are many choices to deploy an application. They range from easy to Do It Yourself. With that range, there is also a reasonably correlated cost associated.

For instance, while Heroku is one of the easiest ways to deploy a Rails application, it is one of the more expensive options. On the opposite side of the spectrum, you could get a VPS from services such as Digital Ocean or Linode and then wholly manage the server administration and deployments yourself. This option, however, is a reasonable approach where the only cost is around the VPS(s) you use. In the middle is an option that uses the low cost of VPS/servers and handles most of the service and deployment orchestration for you, and it is called Hatchbox[1]

In this chapter, we will go over the Heroku deployment strategies. While "Do It Yourself" is cheap, the time it will take to set up, manage, and administrate usually outweighs the low price.

13.2 Prepare application

When preparing to deploy, there are a few files and configuration items you may need to update and add some settings to ensure the Rails application is ready to run on a domain of your choice.

[1] https://hatchbox.io

First is making sure any setting that relies on a domain name to construct paths, URLs, etc., will need to be added or adjusted. In this case, make sure the following is in your `config/environments/production.rb`:

```
...
config.action_cable.url = 'wss://example.com/cable'
config.action_cable.allowed_request_origins = [
  'http://example.com',
  /http:\/\/example.*/
]

...

config.action_mailer.default_url_options = { :host => 'ex\
ample.com' }
routes.default_url_options = { :host => "example.com" }
config.action_mailer.asset_host = "https://example.com"

...
```

You are replacing your domain wherever you see `example.com`. For the most part, this makes sure Mailers and ActionCable work on your production domain.

Though `Rack::Deflator` was mentioned in the previous chapter. If you only want to apply GZIP on production instances, you may want to move this out of `application.rb` and into your `config/environments/production.rb` file. Personally, I don't mind it existing in any environment, but it is my preference!

You should have set up your mailer sending settings in Chapter 8 for production. If not, refer back to the settings in the chapter and add them now. Your deployed application cannot send any outgoing mail without those settings!

Lastly, there is a setting in `config/environments/production.rb`

for forcing SSL on your production environment. The setting is commented out by default, and you will want to uncomment it shortly. As deployment systems will need some settings to be modified and DNS changes to be made on your domain, you should wait to make that change!

13.3 Deploy with Heroku

Deploying on Heroku is reasonably easy! There will be a little setup at the beginning, but then it will be mostly smooth sailing!

- You will want to install the Heroku Toolbelt. You can do so with instructions from Heroku[2].
- You will want to create an app in Heroku. You can do so with the following command `heroku create:app *app_name_here*`
- You can now begin to add a free level Postgresql and Redis add-on through their user interface.
- Heroku merges in their needed database connection settings, thus allowing you to use a minimal configuration in your database file(`config/database.yml`):

```
production:
  encoding: utf8
  pool: 15
```

5. To connect to Redis(for ActionCable) in production you will need to add the `ENV["REDIS_URL"]` environment variable to the `config/cable.yml` file:

[2] https://devcenter.heroku.com/articles/heroku-cli

```
development:
  adapter: redis
  url: redis://127.0.0.1:6379/1

test:
  adapter: async

production:
  adapter: redis
  url: <%= ENV["REDIS_URL"] %>
```

- You are actually in the home stretch here. Next, you will want to make sure you add Heroku to your git configuration; if you run the `heroku create:app` command in your application's directory, you should see that Heroku-based remote git was added via this command, `git config --list | grep heroku`.
- We will need to make sure environment variables are loaded to Heroku that will be used by the application and or its gems. First, add a variable to match the REDIS_URL, `heroku config:add REDIS_PROVIDER=${REDIS_URL}`. By using Rails Encrypted Credentials, you only need to add the `RAILS_MASTER_KEY` in the Heroku Environment Variables.
- Adjust the concurrency numbers in the production Procfile to:

```
web: bundle exec puma -t 3:5 -p ${PORT:-3000} -e ${RACK_E\
NV:-development}
worker: bundle exec sidekiq -q default -q mailers -c 3
```

- You can now run your first deployment with `git push heroku master`.
- While the initial deployment sets up quite a bit for you, the migrations will still need to be run with `heroku run rake db:migrate`.

- You should now add your domain name through the `Settings` tab and `Domain` section of that page. You will be instructed to update your DNS to point to Heroku.
- If upgrading to a paid dyno, you will have SSL. This change will be done through the `Settings` tab. Once added, you can now update the `config.force_ssl = true` line of your production configuration. Finally, push your change to Heroku with `git push heroku master`.
- DONE! YAY!!!

There you have it. There are other add-ons you can add through Heroku, ranging from log parsing to bug catching and too many others to name! Each will be set up through the dashboard and provide you with any additional application-level changes if needed.

14. A Few Other Rails Tools

Here are two more features new to this version of Rails that can be used to optimize your application and make it more secure. These tools are in their own chapter as they are not vital to building the app, but they can be helpful to know and use.

14.1 Async Queries

Another source of slowdowns in an app is running multiple queries, which by default get run sequentially as the application code runs into them. However, a new tool has been introduced to allow queries to be run asynchronously (or simultaneously) to speed up the time it takes your application to get data from the database.

This feature is baked right into ActiveRecord and is available by default, meaning you must chain the `.load_async` method to the queries you want to run asynchronously.

For example, loading the Team's "show" page will retrieve team members, their standups, the `current_user`'s notifications, and events! By introducing that method to these queries, the application can be sped up even further:

app/controllers/application_controller.rb

```
class ApplicationController < ActionController::Base

  ...

  def set_teams_and_standups(date)
    @team = Team.includes(:users).find(params[:id])
    @standups = @team.users.load_async.flat_map do |u|
      u.standups.where(standup_date: date)
       .includes(:dids, :todos, :blockers)
       .references(:tasks)
    end
  end

  def notification_standups
    Standup.notification_standups(current_user).load_async
  end

  ...
```

As you can see, the standups and notifications are now loaded in parallel. Finally, we can head over to the `TeamEvents` file and add the method there as well:

app/controllers/concerns/team_events.rb

```
module TeamEvents
  def set_events
    @events =
      @team
      .events
      .where(
        event_time: (
          current_date.to_time.beginning_of_day..current_\
date.to_time.end_of_day
        )
```

```
      )
      .includes(%i[user team])
      .order('event_time DESC')
      .load_async
  end
end
```

That is all that is needed to run queries in parallel in Rails 7!

14.2 Active Record Encryption

Securing your data at rest is the name of the game with this Rails tool. When using this encryption, the columns you specify within the particular model files will be encrypted in the database when saved, meaning that if someone were to get database access, they would see a random string instead of the stored raw value. This tool would be great for encrypting personally identifiable information (sometimes abbreviated to PII), sensitive data, or just anything else you would want to be protected.

The first step is to use a Rails command to initialize the encryption tooling.

```
bin/rails db:encryption:init
```

Take the output of the command and add it to your credentials:

```
EDITOR="code --wait" rails credentials:edit
```

Now, let's say, for the sake of security, we want to add the application-level encryption to the Task's `title`, which is what is later displayed to the end user. You will first start by adding a configuration line to your `config/application.rb` to make sure the application does not get hung up on `null` or empty database values:

```
...
config.active_record.encryption.support_unencrypted_data \
= true
...
```

With these changes made, an application restart is needed. End your development session and then run `bin/dev`

Next, we can add the change to the `Task` model:

```
class Task < ApplicationRecord
  validates_presence_of :title
  validates_presence_of :type

  has_many :task_memberships
  has_many :standups, through: :task_memberships

  after_commit { standups.find_each(&:touch) }

  encrypts :title ## new encryption line
end
```

You can now jump to the console to play around with the encrypted objects (`bin/rails c`):

To encrypt the last object:

```
Task.last.encrypt
```

Or to encrypt all of the records for the model:

```
Task.all.each(&:encrypt)
```

This command will use the keys, salts, and other configurations from the `init` command and encrypt the data in the database. If

you look at the last record, `Task.last`, you will see your record still reads back correctly in the console, or if you head to your application, you will see the Tasks are still displaying to the end user.

That process handles read-only fields, but what if you needed to encrypt a field that the application would query? The application requires a way to deterministically match the query's plain text input to the encrypted database value. Luckily for us and all Rails developers, the Rails maintainers had thought about this and offered a simple option to the `encrypts` model line, `deterministic: true`. Let's update the email field on the user:

app/models/user.rb

```
class User < ApplicationRecord
  rolify
  # Include default devise modules. Others available are:
  # :confirmable, :lockable, :timeoutable, :trackable and\
 :omniauthable
  devise :invitable, :database_authenticatable, :register\
able,
         :recoverable, :rememberable, :validatable,
         :omniauthable, omniauth_providers: [:github]

  belongs_to :account, optional: true
  attr_accessor :role

  has_many :team_memberships
  has_many :teams, through: :team_memberships

  has_many :standups

  encrypts :email, deterministic: true ## new encrypts li\
ne

  def after_database_authentication
```

```
    Analytics::Mixpanel::IdentifyPersonJob.perform_later(\
self)
  end
end
```

Again, you can now jump to the console to play around with the encrypted objects (`bin/rails c`):

To encrypt the last object:

```
User.last.encrypt
```

Or to encrypt all of the records for the model:

```
User.all.each(&:encrypt)
```

Now, if you check the records in the console or use the app, it still works as intended, but now the user's email is encrypted in the database.

There we go! That is it! You are now using both asynchronous queries and encrypted fields in your app, with only a few lines of code for each tool!