Multitenancy *with Rails*

Building industrial strength Software-as-a-Service Applications

by Ryan Bigg

Multitenancy with Rails - 2nd edition

Ryan Bigg

This book is for sale at http://leanpub.com/multi-tenancy-rails-2

This version was published on 2018-04-13

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.

© 2015 - 2018 Ryan Bigg

Contents

Laying the foundations

We're now going to add accounts to the Twist application and allow users to sign up for new accounts. These accounts will be the foundations on which we'll build our multitenancy work.

An account will have a single owner and many other users associated with it. The owner will have permission to invite or remove users. The accounts' owners will also be able to add new books.

For just this chapter, we'll focus on laying the foundations for making Twist multitenanted, which will involve creating accounts and linking books to those accounts. In the next chapter we'll look at managing the users for those accounts.

As we add these foundational features, we'll be tweaking bits and pieces of Twist to adapt to those new features. This'll be very similar to how you'll tweak your own application to fit the new multitenancy features too.

But first, some background on what Twist is.

Twist: the book review application

Twist is a non-trivial book review application. By "non-trivial" I mean that it has several different elements to it, just like your application (probably) does. It's an already built application which has some tests written in RSpec that use Capybara. It uses Sidekiq (and therefore Redis[1] too) to run background jobs which fetch books from GitHub and processes them.

Twist is a legacy Rails application

Twist's original design came about 6 years ago. Since then, it has evolved dramatically, and so it's a great example of a "legacy" application. If you're new to the Rails world and you've completed some basic tutorials, then this is perfect for you. You might only have experience with building small applications from scratch from those tutorials, using just one version of Ruby and Rails. Twist has been upgraded through multiple versions of both Ruby and Rails, and it has the scars to prove it!

In the real world when you find a job, it's likely that you'll be working on an application that has existed for some time. It will have similar scars to Twist. There'll be code in there that'll leave you scratching your head, and code that you will (hopefully) marvel at. Twist will give you some practice for those kinds of scenarios.

[1]Redis is a key-value store which is used to store Sidekiq's jobs. You can learn more about Redis at http://redis.io.

(Writing this book has also given me a great excuse to tidy up the code once more too!)

By using Twist, we'll learn how to apply the multitenancy concepts in this book to a moderately complex application and ultimately be the better for it. This'll help in the long run when you go to apply these concepts to your real applications.

Twist's data model is very simple: Books have many chapters, which have many elements, which have many notes, which have many comments. This will make the work we'll undertake in this book much easier because we'll scope everything by an account's books. We'll determine if a user has permission to read a chapter by checking if the book that chapter belongs to an account that the user has access to.

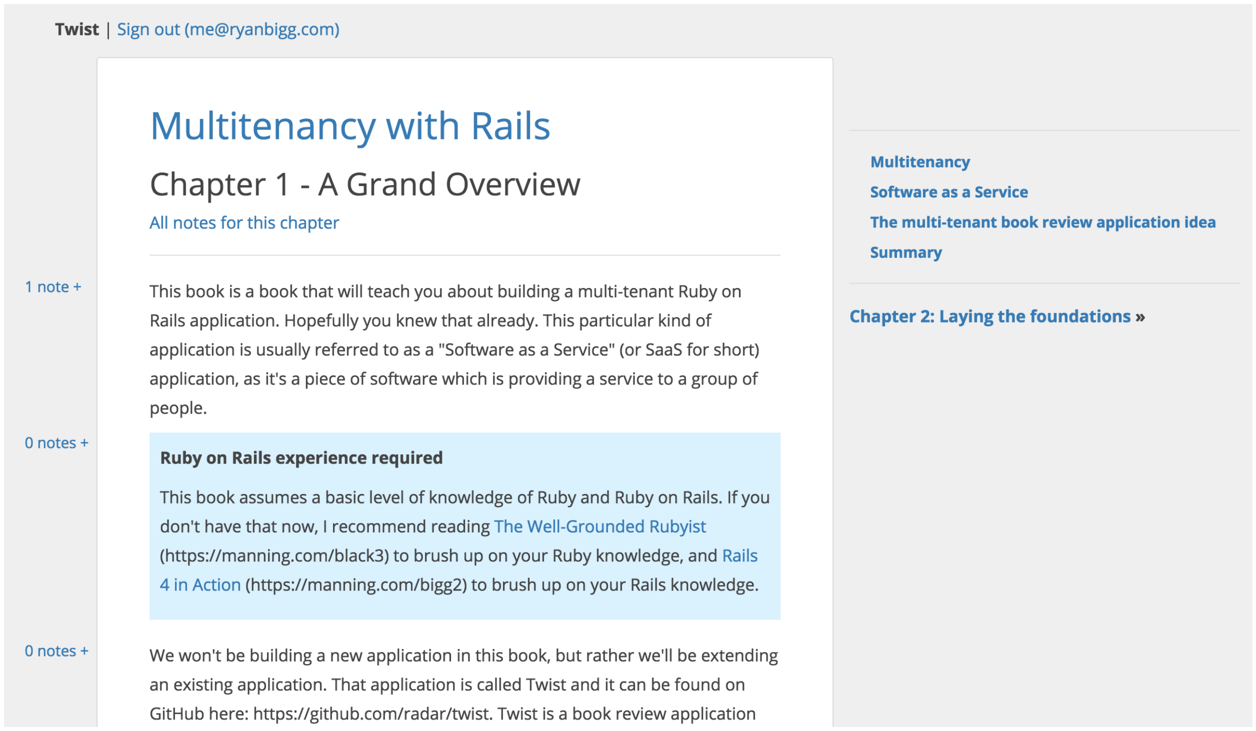
The way Twist works is this: Twist receives a post-receive hook from GitHub's webhooks feature[2], which hits the `receive` action inside `BooksController`. This action enqueues a job to fetch the book's repo from GitHub. Twist clones that repo (or checks out the latest commit if already cloned), which contains the files for a book to a local `repos` directory. Twist then reads the manifest file (called `Book.txt`) within that repo and then reads the Markdown files mentioned in that manifest.

A manifest file looks like this:

```
frontmatter:
introduction.markdown

mainmatter:
chapter_1.markdown
chapter_2.markdown
chapter_3.markdown
chapter_4.markdown
chapter_5.markdown
chapter_6.markdown
```

It then takes these Markdown files mentioned in the manifest and generates a bunch of objects from them (chapters, elements and images) and stores them in the database. It then presents these processed elements when someone views a chapter. This is what the first few paragraphs of this book look like:

[2] GitHub Webhooks: https://developer.github.com/v3/repos/hooks



Twist | Sign out (me@ryanbigg.com)

Multitenancy with Rails

Chapter 1 - A Grand Overview

All notes for this chapter

1 note +

This book is a book that will teach you about building a multi-tenant Ruby on Rails application. Hopefully you knew that already. This particular kind of application is usually referred to as a "Software as a Service" (or SaaS for short) application, as it's a piece of software which is providing a service to a group of people.

0 notes +

Ruby on Rails experience required

This book assumes a basic level of knowledge of Ruby and Ruby on Rails. If you don't have that now, I recommend reading The Well-Grounded Rubyist (https://manning.com/black3) to brush up on your Ruby knowledge, and Rails 4 in Action (https://manning.com/bigg2) to brush up on your Rails knowledge.

0 notes +

We won't be building a new application in this book, but rather we'll be extending an existing application. That application is called Twist and it can be found on GitHub here: https://github.com/radar/twist. Twist is a book review application

Multitenancy

Software as a Service

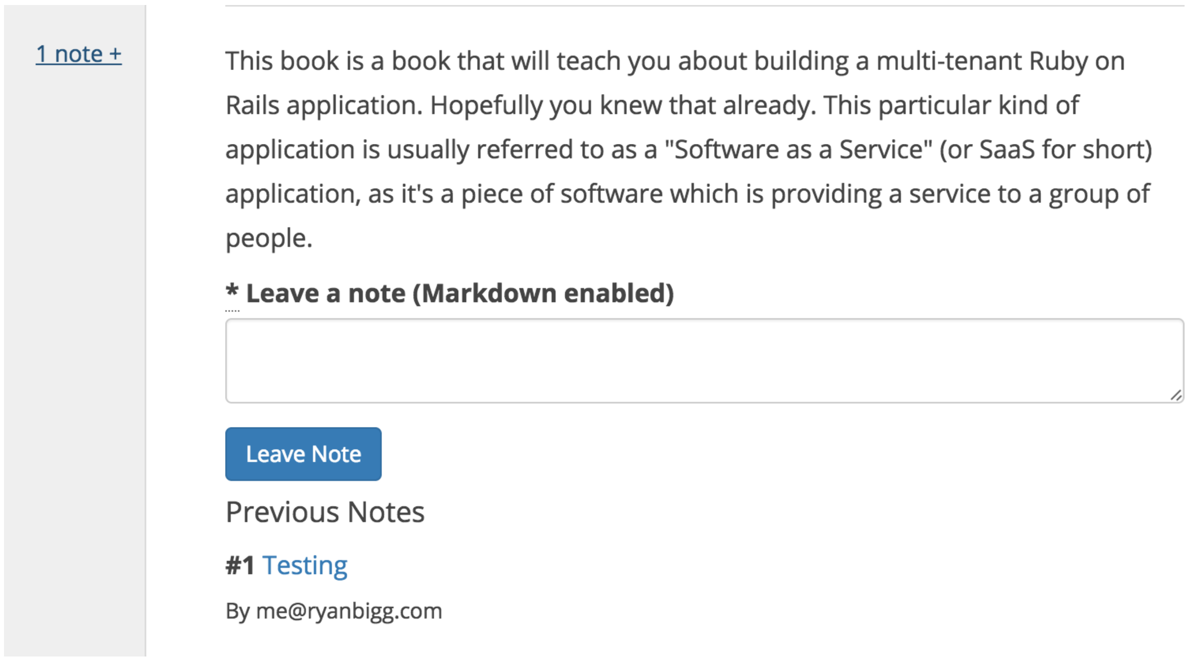
The multi-tenant book review application idea

Summary

Chapter 2: Laying the foundations »

Twist screenshot

If a reader of the book spots an error in the book, they can click the note count next to an element and then a form will pop up:



1 note +

This book is a book that will teach you about building a multi-tenant Ruby on Rails application. Hopefully you knew that already. This particular kind of application is usually referred to as a "Software as a Service" (or SaaS for short) application, as it's a piece of software which is providing a service to a group of people.

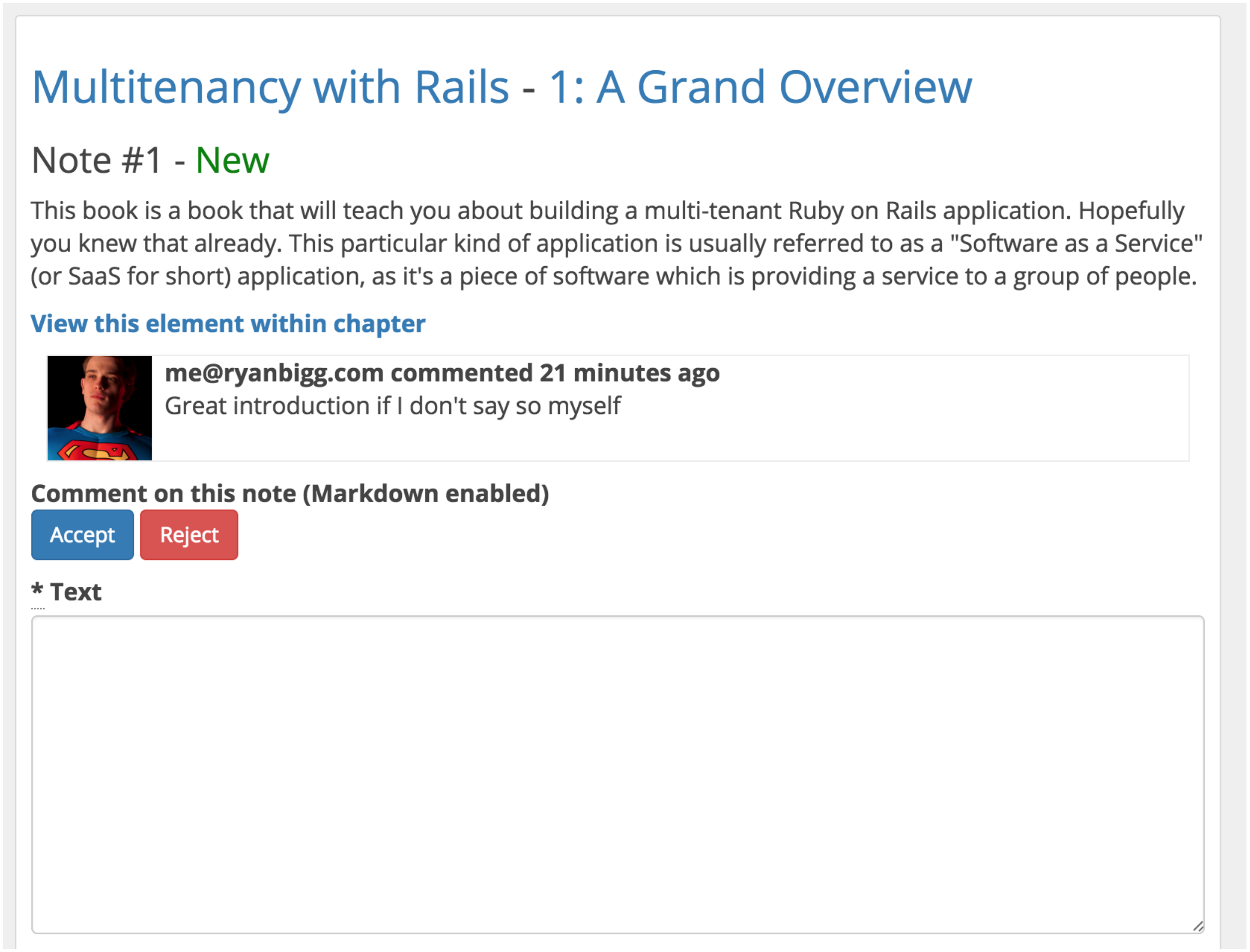
Previous Notes

#1 Testing

By me@ryanbigg.com

New note form

After a note has been created, it can be viewed by anyone with access to the book:


Multitenancy with Rails - 1: A Grand Overview

Note #1 - New

This book is a book that will teach you about building a multi-tenant Ruby on Rails application. Hopefully you knew that already. This particular kind of application is usually referred to as a "Software as a Service" (or SaaS for short) application, as it's a piece of software which is providing a service to a group of people.

me@ryanbigg.com commented 21 minutes ago

Great introduction if I don't say so myself

View this element within chapter
Comment on this note (Markdown enabled)
Accept
Reject
* Text

Viewing a note

The author of the book can then review all the notes that readers leave, and then make their books the best books that they could possibly be.[3] Each note can transition through some states: "New", "Accepted", "Rejected" and "Reopened". If a note is "Accepted" or "Rejected", then it's considered to be reviewed by the author and the correction mentioned in the note is either applied or not depending on if it's "Accepted" or "Rejected". "New" and "Reopened" are basically the same state, but "Reopened" can give a clearer idea if a note has a past history of being accepted or rejected.

Twist is not perfect by any means [4], but it does the job well enough. An example of this failure-to-be-perfect is that books need to be added to the app through the console.

Before we can begin with this application, we should clone it down from GitHub and run the tests to ensure everything is in working order. You will need to start `redis-server` first because Twist depends on Sidekiq, which in turn depends on Redis.

[3] All without the interference and incompetence of a publishing company!

[4] Find me an application that is! See earlier aside about the legacy-ness of Twist.

You will also need to have a PostgreSQL database setup.

```
git clone git@github.com:radar/twist
cd twist
git submodule update --init
bundle install
rake db:setup
bundle exec rspec spec
```

Ruby 2.2.2 or above required

Twist uses Rails 5 and that version of Rails requires Ruby 2.2.2 or higher. This book has been written with Ruby 2.3.3 and all the code examples have been tested using that version. It's for this reason that I would recommend that you use Ruby 2.3.3 as well.

You need to run the `git submodule update --init` command here to download the submodules contained in this repo, mainly the `radar/markdown_book_test` which is used for the tests and stored at `spec/fixtures/repos/radar/markdown_book_test`.

Here's what you will see after running the tests:

```
48 examples, 0 failures
```

If the tests are all green, we can now begin the work that we'll be doing in this book.

If you get stuck…

If you get stuck at any point in this book, you can look at the code from the example application and compare it with your own. The example code is at https://github.com/radar/twist/tree/mtwr-walkthrough, and each commit begins with the section title that it's from.

For example, the next section is called "Account Sign Up", and so if you wanted to see the code from that you'd find a commit which has a message beginning with "Account Sign Up".

Account Sign Up

The first new thing that we want our users to be able to do in Twist is to sign up for an account. After they've created their account the next step will be to let them add books from GitHub to their account. An account will be the bedrock on which we will build our multitenancy foundations. Anything that is tenant-specific in our application will be linked to an account. The words “tenant” and “account” for the remainder of this book are interchangeable.

Each account will have their own collection of books that are completely separate from any other account’s list of books. Only users which have access to the account will have access to the books within that account. We won’t be setting that much up yet; we’ll just focus on the account sign up feature for now.

The process of signing up for an account will be fairly bare-bones for now:

- The user should be able to click on a “Account Sign Up” link
- Enter their account’s name
- Click “Create Account” and;
- See a message like “Your account has been created.”

Super basic, and quite easy to set up.

Let’s write the test for this now in `spec/features/accounts/sign_up_spec.rb`:

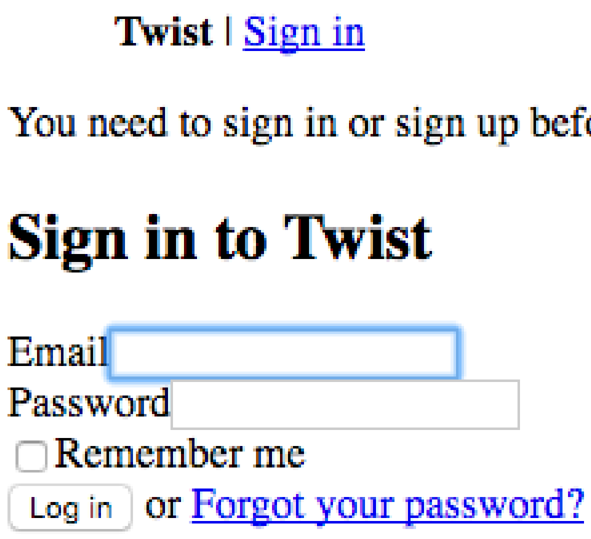
spec/features/accounts/sign_up_spec.rb

```ruby
require "rails_helper"

feature "Accounts" do
  scenario "creating an account" do
    visit root_path
    click_link "Create a new account"
    fill_in "Name", with: "Test"
    click_button "Create Account"

    within(".flash_notice") do
      success_message = "Your account has been created."
      expect(page).to have_content(success_message)
    end
  end
end
```

This spec is quite simple: visit the root path, click on a link, fill in a field, click a button, see a message. Run this spec now with `rspec spec/features/accounts/sign_up_spec.rb` to see what the first step is in making it pass. We should see this output:

```
Failure/Error: click_link "Create a new account"
Capybara::ElementNotFound:
  Unable to find link "Create a new account"
```

This test cannot find the "Account Sign Up" link, but why is that? We have two ways of finding out what it can see: we can put `save_and_open_page` in the test before the clicking of the "Create a new account" link, run it again and see this:



Twist | Sign in

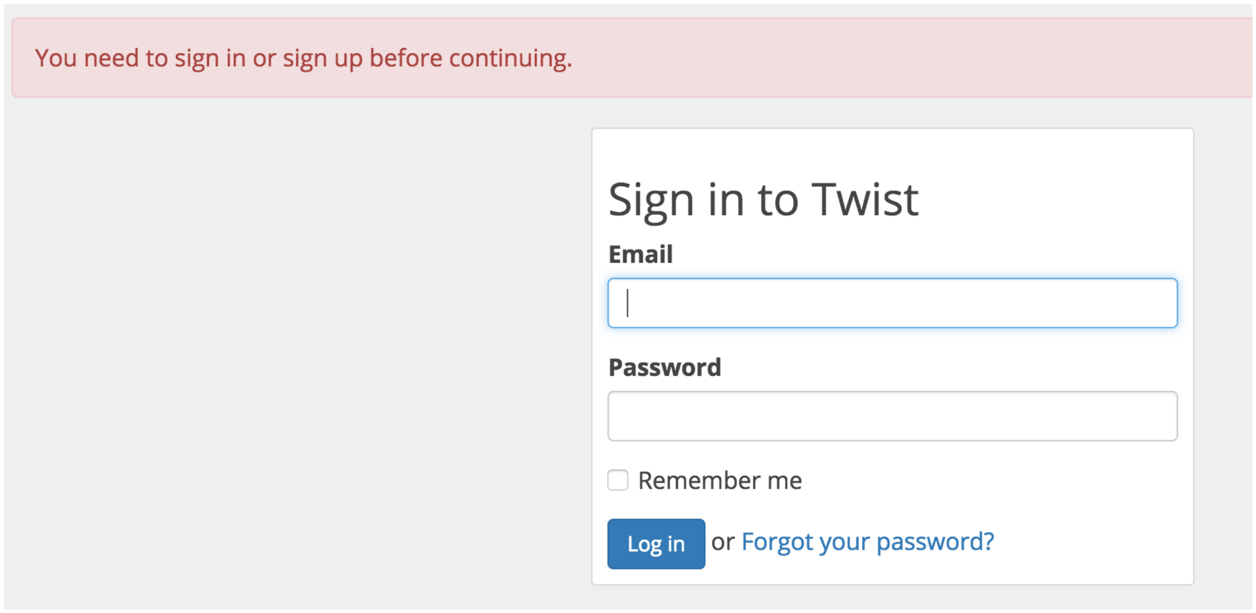
You need to sign in or sign up before continuing.

Sign in to Twist

Email
Password
Remember me
Log in or Forgot your password?

Save and open page

Alternatively, we could start `rails server` and visit `http://localhost:3000` and see the same page, but with styles applied:


You need to sign in or sign up before continuing.

Sign in to Twist

Twist root path

Either way, we'll see the same page.

> **`save_and_open_page` vs `rails server`**
>
> I show both ways here because different people prefer different ways of evaluating what their test sees. Personally, I prefer `save_and_open_page` because it reflects the real state of the test and shows exactly what the test is seeing.
>
> Imagine if we had a more complex test which setup a book with some chapters and notes. `save_and_open_page` would work better there as it would show the state of the system during the test. If we used `rails server` instead, we would need to setup the state of the test in the development environment which is just duplicating a lot of the work.

We can see from our tests here that its seeing a page which prompts it to sign in, rather than to sign up for an account. We can find out why this is happening by inspecting the `log/test.log` file to see what requests its making to arrive at this point.

The first request we'll see is this one:

```
Started GET "/" for 127.0.0.1 at [timestamp]
Processing by BooksController#index as HTML
Completed 401 Unauthorized in 14ms (ActiveRecord: 0.0ms)
```

The request is headed for the `index` action in `BooksController`, but it's stopped in its tracks. The response given is a 401 Unauthorized response, rather than a 200 response and this is because of this line in `app/controllers/books_controller.rb`:

```
before_action :authenticate_user!, except: [:receive]
```

This `authenticate_user!` method comes from Devise, and it's used to ensure that a user is authenticated before a request can be made to any actions in this controller *except* the `receive` action. Therefore we can tell that we're seeing this "You need to sign in or sign up before continuing" message because we haven't, indeed, signed in or signed up yet!

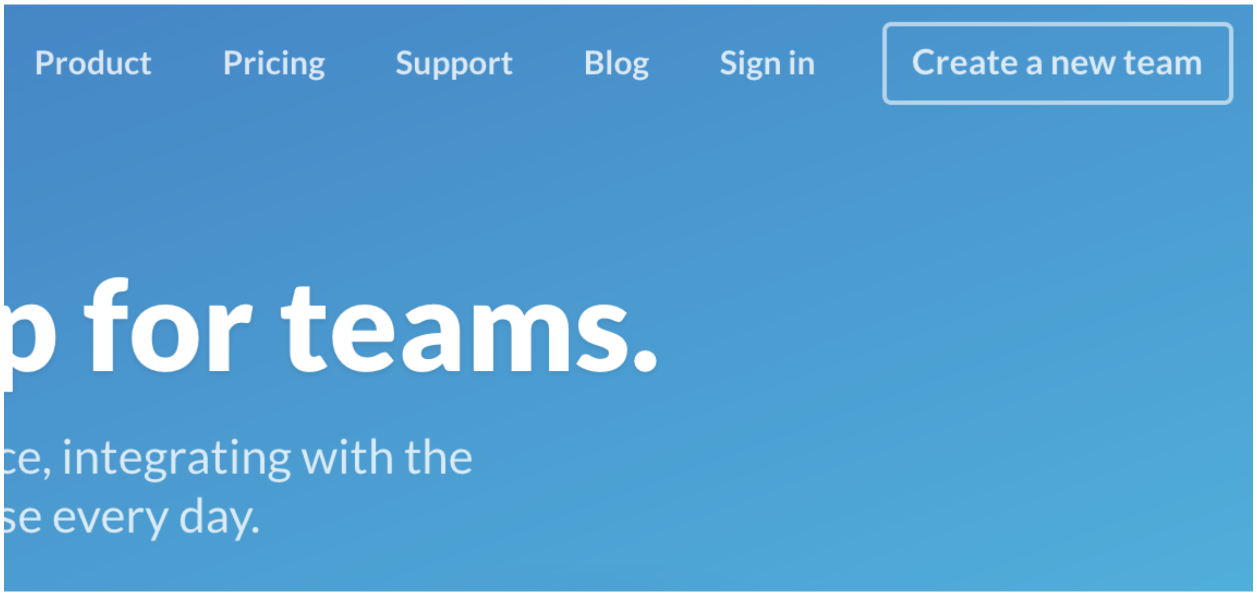
This isn't what we want to be seeing though: we want to be seeing a link to sign up for account. Therefore, rather going to the `index` action in `BooksController`, we should be sending users who first visit our application to somewhere else; a landing page of sorts.

Implementing account sign up

Let's take a look at how another Software as a Service application sets up their account sign up feature.

Slack (https://slack.com) is another great example of a multitenanted application. Slack is a real-time communication tool built for teams and each team has their own set of data which is completely segregated from any other team's. A user can belong to multiple teams on Slack. We'll be implementing something very similar for Twist: accounts have their own books which are completely segregated from any other account's books.

For implementing the account sign up, we're going to take a leaf out of Slack's playbook. When you go to http://slack.com, this is what you see at the top right:



Slack

At the top right, there's a link to create a new slack team. This is what we're going to be replicating in this section, but rather than creating a 'team' we'll be creating an account.

The first step to do here is to create a new landing page where we can present our users with a link to create a new account. We can start by creating a new controller.

We want this controller to be fairly bare-bones for now. We don't need any helpers, assets or controller specs. Therefore we'll generate this new controller using this command:

```
rails g controller home --no-helper --no-assets --no-controller-specs
```

Rather than routing to `BooksController`'s `index` action for the root of the Twist application, we'll route instead to the `index` action in this new `HomeController`. Let's change the `config/routes.rb` file line where it has this:

config/routes.rb

```
root to: "books#index"
```

To this:

config/routes.rb

```
root to: "home#index"
```

Running `bundle exec rspec spec/features/accounts/sign_up_spec.rb` at this point will show this error:

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

The `index` action of this new controller doesn't need to anything right now. What we really need here is a template that contains the "Create a new account" link. Let's create one now:

app/views/home/index.html.erb

```
<div class='content col-md-8'>
  <%= link_to "Create a new account", new_account_path %>
</div>
```

The next run of our test will tell us that we're missing the `new_account_path` routing helper:

```
Failure/Error: visit root_path
ActionView::Template::Error:
  undefined local variable or method `new_account_path' ...
```

Let's add a route for that to our `config/routes.rb` file now, underneath the `root` route:

config/routes.rb

```ruby
get "/accounts/new", to: "accounts#new", as: :new_account
```

This route is going to need a new controller to go with it, so create it using this command:

```
rails g controller accounts --no-helper --no-assets
```

The `new_account_path` is pointing to the currently non-existant `new` action within this controller. This action should be responsible for rendering the form which allows people to enter their account's name and create their account. Let's create that view now with the following code placed into `app/views/accounts/new.html.erb`:

app/views/accounts/new.html.erb

```erb
<div class='col-md-offset-4 col-md-4 content'>
  <h2>Create a new account</h2>

  <%= simple_form_for(@account) do |account| %>
    <%= account.input :name %>
    <%= account.submit class: "btn btn-primary" %>
  <% end %>
</div>
```

The `@account` variable here isn't set up inside `AccountsController` yet, so let's open up `app/controllers/accounts_controller.rb` and add a `new` action that sets it up:

app/controllers/accounts_controller.rb

```ruby
def new
  @account = Account.new
end
```

Ok then, that's the "Account Sign Up" link and sign up form created. What happens next when we run `bundle exec rspec spec/features/accounts/sign_up_spec.rb`? Well, if we run it, we'll see this:

```
Failure/Error: @account = Account.new

NameError:
  uninitialized constant AccountsController::Account
```

We're now referencing the `Account` model within the controller, which means that we will need to create it. Instances of the `Account` model should have a field called "name", since that's what the form is going to need, so let's go ahead and create this model now with this command:

```
rails g model account name:string
```

This will create a model called `Account` and with it a migration which will create the `accounts` table that contains a `name` field. To run this migration now for the application run:

```
rake db:migrate
```

We'll also need to set the environment for the test database's schema, so that the migration can run successfully there too:

```
bin/rails db:environment:set RAILS_ENV=test
```

What next? Find out by running the spec with `bundle exec rspec spec/features/accounts/sign_-up_spec.rb`. We'll see this error:

```
undefined method `accounts_path' for ...
```

This error is happening because the `form_for` call inside `app/views/accounts/new.html.erb` is attempting to reference this method so it can find out the path that will be used to make a `POST` request to create a new account. It assumes `accounts_path` because `@account` is a new object.

Therefore we will need to create this path helper so that the `form_for` has something to do. Let's define this in our routes now using this code:

config/routes.rb

```ruby
post "/accounts", to: "accounts#create", as: :accounts
```

With this path helper and route now defined, the form should now post to the `create` action within `AccountsController`, which doesn't exist right now, but will very soon. This action needs to accept the parameters from the form, create a new account using those parameters and display the "Your account has been successfully created" message. Write this action into `AccountsController` now:

app/controllers/accounts_controller.rb

```ruby
def create
  account = Account.create(account_params)
  flash[:notice] = "Your account has been created."
  redirect_to root_url
end
```

The `create` action inside this controller will take the params from the soon- to-be-defined `account_params` method and create a new `Account` object for it. It'll also set a notice message for the next request, and redirect back to the root path for the application.

Parameters within modern versions of Rails (since Rails 4) are not automatically accepted (thanks to the strong parameters feature), and so we need to permit them. We can do this by defining that `account_params` method as a private method after the `create` action in this controller:

app/controllers/accounts_controller.rb

```ruby
def create
  account = Account.create(account_params)
  flash[:notice] = "Your account has been created."
  redirect_to root_url
end

private

def account_params
  params.require(:account).permit(:name)
end
```

This `create` action is the final thing that the spec needs in order to pass. Let's make sure that it's passing now by re-running `rspec spec/features/accounts/sign_up_spec.rb`.

```
1 example, 0 failures
```

Great! Now would be a good time to commit this change that we've made.

```
git add .
git commit -m "Added accounts"
```

What we've done so far in this chapter is added a way for users to create accounts in Twist. These will provide the grounding for account multitenancy features that we'll be building later in this book.

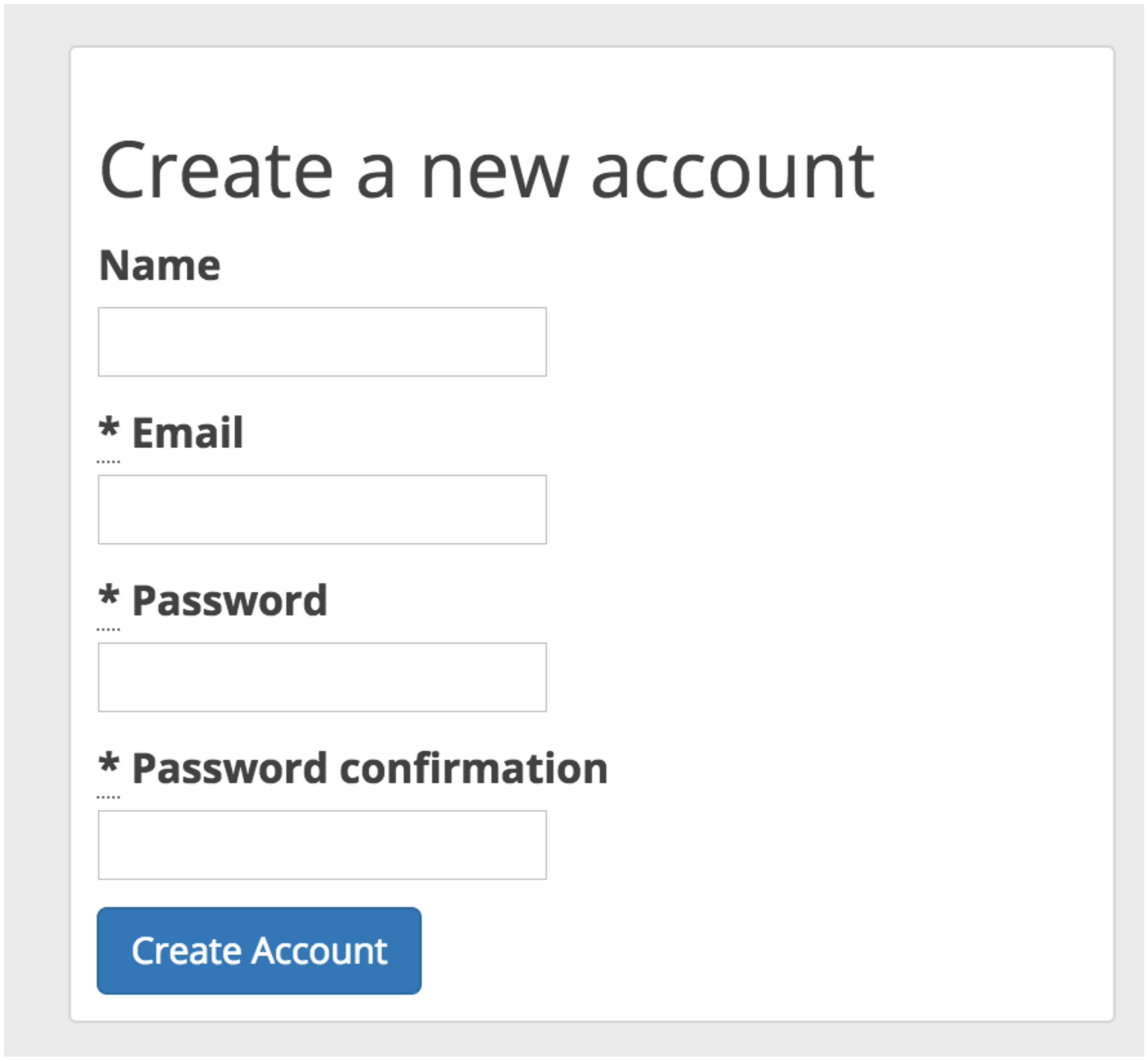
So far in this book we've seen how to add some very, very basic functionality to Twist and now users will be able to create an account in the system. Let's get a little more complex.

What we're going to need next is a way of linking accounts to owners who will be responsible for managing anything under that account.

Associating accounts with their owners

An account's owner will be responsible for doing adminstrative operations for that account such as adding new books and managing the users for that account. An owner is just a user for an account, and Twist luckily already comes with a `User` model that we can use to represent this owner.

Right now, all we're prompting for on the new account page is a name for the account. If we want to link accounts to an owner as soon as they're created, it would be best if this form contained fields for the new user as well, such as email, password and password confirmation.


Create a new account

Name

* Email

* Password

* Password confirmation

New account form

These fields won't be stored on an `Account` record, but instead on a `User` record, and so we'll be using ActiveRecord's support for nested attributes to create the new user along with the account.

Let's update the spec for account sign up now and add code to fill in an email, password and password confirmation field underneath the code to fill in the name field.

spec/features/accounts/sign_up_spec.rb

```ruby
fill_in "Name", with: "Test"
fill_in "Email", with: "test@example.com"
fill_in "Password", with: "password", exact: true
fill_in "Password confirmation", with: "password"
```

Once the "Create Account" button is pressed, we're going to want to check that something has been done with these fields too. The best thing to do would be to get the application to automatically sign in the new account's owner. After this happens, the user should see somewhere on the page that they're signed in. Let's add a check for this now after the "Create Account" button clicking in the test, as the final line of the test:

```
expect(page).to have_content("Signed in as test@example.com")
```

Alright then, that should be a good start to testing this new functionality.

These new fields aren't yet present on the form inside `app/views/accounts/new.html.erb`, so when you run this spec using `bundle exec rspec spec/features/accounts/sign_up_spec.rb` you'll see this error:

```
Failure/Error: fill_in "Email", with: "test@example.com"
Capybara::ElementNotFound:
  Unable to find field "Email"
```

We're going to be using nested attributes for this form, so we'll be using a `fields_for` block inside the form to add these fields to the form. Underneath the field definition for the `name` field inside `app/views/accounts/new.html.erb`, add the fields for the owner using this code:

app/views/accounts/new.html.erb

```
<%= account.input :name %>

<%= account.fields_for :owner do |owner| %>
  <%= owner.input :email %>
  <%= owner.input :password %>
  <%= owner.input :password_confirmation %>
<% end %>
```

With the fields set up in the view, we're going to need to define the `owner` association within the `Account` model as well as defining in that same model that instances will accept nested attributes for owner. We can do this with these lines inside the `Account` model definition:

app/models/account.rb

```
class Account < ApplicationRecord
  belongs_to :owner, class_name: "User"
  accepts_nested_attributes_for :owner
end
```

The owner object for an `Account` will be an instance of the `User` model that already exists in this application. Because there's now a `belongs_to :owner` association on the `Account`, we'll need to add an `owner_id` field to the `accounts` table so that it can store the foreign key used to reference account owners. Let's generate a migration for that now by running this command:

```
rails g migration add_owner_id_to_accounts owner_id:integer
```

Due to how we invoked this migration command, it will know that we want a migration which adds the `owner_id` field to the `accounts` table:

db/migrate/[timestamp]_add_owner_id_to_accounts.rb

```ruby
class AddOwnerIdToAccounts < ActiveRecord::Migration[5.0]
  def change
    add_column :accounts, :owner_id, :integer
  end
end
```

That's just another really helpful Rails feature!

Let's run the migration with `rake db:migrate` now so that we can properly associate accounts with their owners in the database.

When we run our test again, we'll see that the email field is still not present:

```
Failure/Error: fill_in "Email", with: "test@example.com"
Capybara::ElementNotFound:
  Unable to find field "Email"
```

The fields aren't displaying because there isn't an owner associated with the account to display fields for. To display these fields, we'll need to add an extra line to the `new` action within `AccountsController` to initialize the owner association for the account:

app/controllers/accounts_controller.rb

```ruby
def new
  @account = Account.new
  @account.build_owner
end
```

The form will now render these fields, which we can see when we run the spec again. It'll get a little further, and this time it will tell us:

```
expected to find text "Signed in as test@example.com" in ...
```

The check to make sure that we're signed in as a user is failing, because of two reasons: we're not automatically signing in the user when the account is created, and we're not displaying this text on the layout anywhere.

We can fix this first problem very simply by signing in the user after the account has been created:

app/controllers/accounts_controller.rb

```ruby
def create
  account = Account.create(account_params)
  sign_in(account.owner)
  flash[:notice] = "Your account has been created."
  redirect_to root_url
end
```

The `sign_in` helper comes from Devise and will setup the session to sign in the account's owner.

While we're in this controller, we should alter the `account_params` method to accept the owner's attributes from the form as well.

```ruby
def account_params
  params.require(:account).permit(:name,
    { owner_attributes: [
      :email, :password, :password_confirmation
    ]}
  )
end
```

If we don't make this modification, the owner will not be created because the `owner_attributes` parameters will be ignored. Without this change, the code will believe that we only want to create an account and not the owner along with it.

The "Signed in as..." text just requires a small modification to the application layout. There's some code in there which currently shows something along the lines of what we want to show already:

app/views/layouts/application.html.erb

```erb
<strong>Twist</strong> |
<% if user_signed_in? %>
  <%= link_to "Sign out (#{current_user.email})",
    destroy_user_session_path, method: :delete %>
<% else %>
  <%= link_to "Sign in", new_user_session_path %>
<% end %>
```

However, our test requires the "Signed in as ..." message instead. Therefore, we'll change the layout to match the test:

```
<strong>Twist</strong> |
<% if user_signed_in? %>
  Signed in as <%= current_user.email %>
  <%= link_to "Sign out",
  destroy_user_session_path, method: :delete %>
<% else %>
  <%= link_to "Sign in", new_user_session_path %>
<% end %>
```

When we run our test again, it will pass once more:

```
1 example, 0 failures
```

Yes! Great work. Now we've got an owner for the account being associated with the account when the account is created. What this allows us to do is to have a user responsible for managing the account. When the account is created, the user is automatically signed in as that user.

Let's commit that:

```
git add .
git commit -m "Accounts are now linked to owners"
```

In Twist, we're going to be restricting access to the books based on which account they belong to. For instance a book called "Multitenancy with Rails" might belong to the "Ruby Sherpas" account, and to view the book you would go to the `rubysherpas` account on Twist and then sign in for that account. If you're the owner or one of the users that is associated with the account, you should be able to sign in to that account.

To uniquely identify an account and to provide a way to navigate to it, we can add a `subdomain` field to the accounts table.

Adding subdomain support

Subdomains are our tool of choice for uniquely identifying accounts in this section. Alternatively, we could let account owners create accounts with a particular name, and route it in our application like `example.com/rubysherpas`. Rather than doing it that way, we'll stick with subdomains.[5]

[5](Minor) spoiler alert! We're using subdomains here because it's easier to detect if a route is using a subdomain rather than if it's for a particular account. For instance, is `example.com/help` an account, or just a regular route?

It's worth mentioning at this point that it's trickier and more expensive to get an SSL certificate for a site that uses subdomains like this.

That kind of certificate is called a "Wildcard subdomain" certificate, and it's a certificate that is for *any* subdomains of a domain, rather than just one particular subdomain. If that's something that may be of a concern to you, then perhaps try going down the path-for-an-account (i.e. http://twistbooks.com/rubysherpas) route instead.

To access an account, you'll need to navigate to a route like `rubysherpas.example.com`. From there – if you have permission – you'll be able to see all the books for that account.

We don't currently have subdomains for accounts, so that'd be the first step in setting up this new feature.

Adding subdomains to accounts

When an account is created for Twist, we'll get the user to fill in a field for a subdomain as well. When a user clicks the button to create their account, they should then be redirected to their account's subdomain.

Let's add a subdomain field firstly to our account sign up test, underneath the `Name` field:

spec/features/accounts/sign_up_spec.rb

```
fill_in "Name", with: "Test"
fill_in "Subdomain", with: "test"
```

We should also ensure that the user is redirected to their subdomain after the account sign up as well. To do this, we can put this as the final line in the test:

```
expect(page.current_url).to eq("http://test.example.com/")
```

If we were to run the test now, we would see it failing because there is no field called "Subdomain" on the page to fill out.To make this test pass, there will need to be a new field added to the accounts form:

app/views/accounts/new.html.erb

```
<%= account.input :subdomain %>
```

This field will also need to be inside the `accounts` table. To add it there, run this migration:

```
rails g migration add_subdomain_to_accounts subdomain:string:index
```

The `:index` suffix for `subdomain:string` will automatically generate an index for this field, which will make looking up accounts based on their subdomains really speedy[6]. If we look in the migration, this is what we see:

db/migrate/[timestamp]_add_subdomain_to_accounts.rb

```ruby
class AddSubdomainToAccounts < ActiveRecord::Migration[5.0]
  def change
    add_column :accounts, :subdomain, :string
    add_index :accounts, :subdomain
  end
end
```

Run this migration now using the usual command:

```
rake db:migrate
```

This field will also need to be assignable in the `AccountsController` class, which means we need to add it to the `account_params` method:

app/controllers/accounts_controller.rb

```ruby
def account_params
  params.require(:account).permit(:name, :subdomain,
    { owner_attributes: [
      :email, :password, :password_confirmation
    ]}
  )
end
```

When we run the test using `rspec spec/features/accounts/sign_up_spec.rb`, it will successfully create an account, but it won't redirect to the right place:

```
Failure/Error: expect(page.current_url).to eq("http://test.example.com/")
  expected: "http://test.example.com/"
       got: "http://www.example.com/"
```

The test should be redirecting us to the account's subdomain after we've signed in, but instead it's taking us back to the domain's root. In order to fix this, we need to tell the `AccountsController` to redirect to the correct place. Change this line within `app/controllers/accounts_controller.rb`, from this:

[6]For small datasets, the difference is neglible. However, for a dataset of a couple of thousand accounts the difference can be really noticeable. We're adding an index *now* so that the lookup doesn't progressively get slower as more accounts get added to the system.

```
redirect_to root_url
```

To this:

```
redirect_to root_url(subdomain: account.subdomain)
```

The `subdomain` option here will tell Rails to route the request to the account's subdomain. Running `bundle exec rspec spec/features/accounts/sign_up_spec.rb` again should make the test pass, but not quite:

```
Failure/Error: within(".flash_notice") do
Capybara::ElementNotFound:
  Unable to find css ".flash_notice"
```

The successful account sign up `flash` message has disappeared! This was working before we added the `subdomain` option to `root_url`, but why has it stopped working now?

The answer to that has to do with how flash messages are stored within Rails applications. These messages are stored within the session in the application, which is scoped to the specific domain that the request is under. If we make a request to our application at example.com that'll use one session, while a request to test.example.com will use another session.

To fix this problem and make the root domain and subdomain requests use the same session store, we will need to modify the session store for the application. To do this, open `config/initializer-s/session_store.rb` and change this line:

config/initializers/session_store.rb

```ruby
Twist::Application.config.session_store :cookie_store,
  key: "_twist_session"
```

To these lines:

config/initializers/session_store.rb

```ruby
options = {
  key: "_twist_session"
}

case Rails.env
when "development", "test"
  options.merge!(domain: "lvh.me")
when "production"
```

```
  # TBA
end

Twist::Application.config.session_store :cookie_store, options
```

This will store all session information in the development and test environments under the `lvh.me` domain[7]. The `lvh.me` domain is used for both environments so that you can access subdomains for Twist using `http://rubysherpas.lvh.me:3000`, and so that later on if we need to run some JavaScript tests they'll be able to access the application.

The catch for this change is that we'll need to access the site through `lvh.me` locally if we want to test it out. It's not that big of a deal. Just remember to use `lvh.me` instead of `localhost` from this point onwards.

The change to setting a domain for the `test` environment's session store configuration means that we'll also need to tell Capybara about it with this extra line at the end of `rails_helper.rb`:

spec/rails_helper.rb

```
Capybara.app_host = "http://lvh.me"
```

We'll also need to change our `spec/features/accounts/sign_up_spec.rb` to use this domain instead of `www.example.com`. This is because we changed `Capybara.app_host`. Change this line in the first test:

spec/features/accounts/sign_up_spec.rb

```
expect(page.current_url).to eq("http://test.example.com/")
```

To this:

```
expect(page.current_url).to eq("http://test.lvh.me/")
```

With all these changes, the test will now work:

```
1 example, 0 failures
```

What we have done in this small section is set up subdomains for accounts so that users will have somewhere to go to sign in and perform actions for accounts.

We should commit this change now:

[7] lvh.me is a DNS hack which redirects to localhost. It's very handy for testing subdomain codes, because it uses a domain that has a TLD length of 2 ("lvh" and "me") rather than just "localhost". This'll be important later on.

```
git add .
git commit -m "Added subdomains to accounts"
```

Later on, we're going to be using the account's subdomain field to scope the data correctly to the specific account. However, at the moment, we've got a problem where one person can create an account with a subdomain, and there's nothing that's going to stop another person from creating an account with the exact same subdomain. Therefore, what we're going need to do is to add some validations to the `Account` model to ensure that two users can't create accounts with the same subdomain.

Ensuring unique subdomain

Let's write a test for this flow now after our original test:

spec/features/accounts/sign_up_spec.rb

```ruby
scenario "Ensure subdomain uniqueness" do
  Account.create!(subdomain: "test", name: "Test")

  visit root_path
  click_link "Create a new account"
  fill_in "Name", with: "Test"
  fill_in "Subdomain", with: "test"
  fill_in "Email", with: "test@example.com"
  fill_in "Password", with: "password"
  fill_in "Password confirmation", with: 'password'
  click_button "Create Account"

  expect(page.current_url).to eq("http://lvh.me/accounts")
  expect(page).to have_content("Sorry, your account could not be created.")
  expect(page).to have_content("Subdomain has already been taken")
end
```

In this test, we're going through the flow of creating an account again, but this time there's already an account that has the subdomain that the test is attempting to use. When that subdomain is used again, the user should first see a message indicating that their account couldn't be created, and then secondly the reason why it couldn't be.

Running this test using `rspec spec/features/accounts/sign_up_spec.rb:20` will result in it failing like this:

```
Failure/Error: expect(page.current_url).to
  eq("http://lvh.me/accounts")

  expected: "http://lvh.me/accounts"
       got: "http://test.lvh.me/"

  (compared using ==)
```

This indicates that the account sign up functionality is working, and perhaps too well: it's allowing accounts to be created with the same subdomain! Let's fix that up now by first re-defining the `create` action within `AccountsController` like this:

app/controllers/accounts_controller.rb

```ruby
def create
  @account = Account.new(account_params)
  if @account.save
    sign_in(@account.owner)
    flash[:notice] = "Your account has been created."
    redirect_to root_url(subdomain: @account.subdomain)
  else
    flash.now[:alert] = "Sorry, your account could not be created."
    render :new
  end
end
```

Rather than calling `Account.create` now, we're calling `new` so we can build an object. We're assigning this to an *instance* variable rather than a *local* variable, so that it will be available within the `new` view if that view is rendered again; which is what will happen if the `save` fails.

We then call `save` to return `true` or `false` depending on if the validations for that object pass or fail respectively. If it's valid, then the account will be created, if not then it won't be and the user will be shown the "Sorry, your account could not be created." message.

We're going to want to have this "subdomain is already taken" message displayed on the new account form, and to do that we're going to need to add a validation to the `Account` model for that subdomain attribute. A good place for this is right at the top of the model:

app/models/account.rb

```ruby
class Account < ApplicationRecord
  validates :subdomain, presence: true, uniqueness: true
```

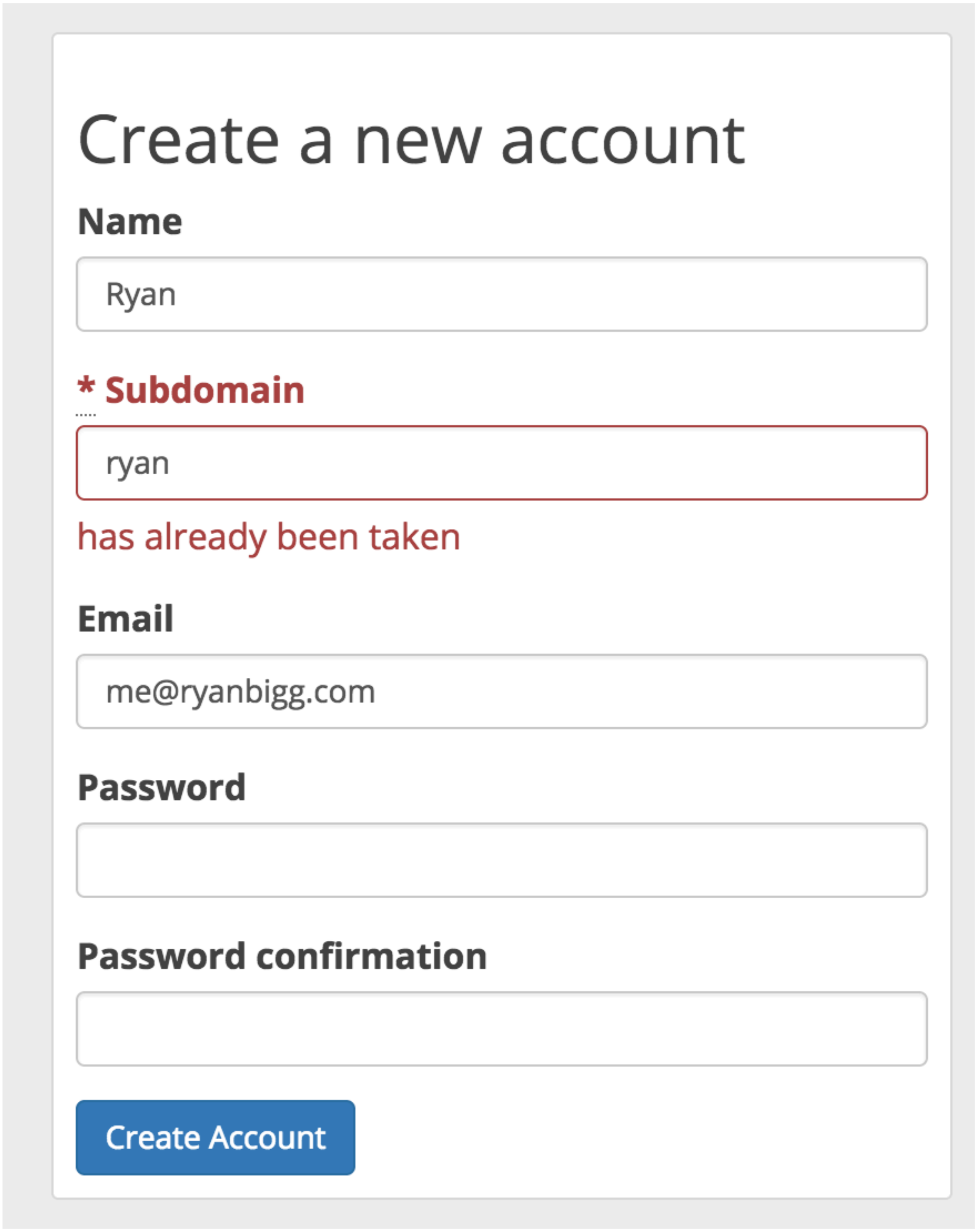
We want to ensure that people are entering subdomains for their accounts, and that those subdomains are unique. If either of these two criteria fail, then the `Account` object should not be valid at all.

When we run this test again, it should at least not tell us that the account has been successfully created, but rather that it's sorry that it couldn't create the account. The new output would indicate that it's getting past that point:

```
Failure/Error: expect(page).to have_content("Subdomain has already been taken")
  expected to find text "Subdomain has already been taken" in
    "...* Subdomainhas already been taken"
```

Hmm, our test is not quite passing just yet. The error message appears on the page, but it's appearing as just "has already been taken". It's got "Subdomainhas" there as that's the text on the page around that element: the 'Subdomain' is the label, and the 'has already been taken' is the error message.

Let's create an account ourselves now and then try to create another with the same subdomain. This is what we'll see:


Create a new account

Name

Ryan

* Subdomain

ryan

has already been taken

Email

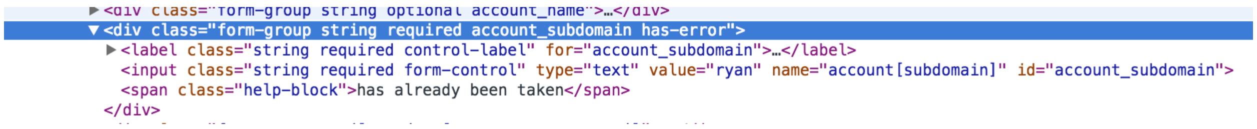
me@ryanbigg.com

Password

Password confirmation

Subdomain has already been taken

If we inspect that field in our web browser, we'll see this:



```
▶<div class="form-group string optional account_name">…</div>
▼<div class="form-group string required account_subdomain has-error">
  ▶<label class="string required control-label" for="account_subdomain">…</label>
   <input class="string required form-control" type="text" value="ryan" name="account[subdomain]" id="account_subdomain">
   <span class="help-block">has already been taken</span>
 </div>
```

Account subdomain inspected

This is a great indicator to show that our test (rather than our code) is wrong. It's looking for the text "Subdomain has already been taken", but really what it should be looking for is the `.account_subdomain .help-block` element and then checking that its content is "has already been taken". Let's adapt our test to this new information now by changing this line:

```
expect(page).to have_content("Subdomain has already been taken")
```

To these lines:

```
subdomain_error = find('.account_subdomain .help-block').text
expect(subdomain_error).to eq('has already been taken')
```

Running the test once more will result in its passing:

```
2 example, 0 failures
```

Good stuff. Now we're making sure that whenever a user creates an account, that the account's subdomain is unique. This will prevent clashes in the future when we use the subdomain to scope our resources by, later on in Chapter 4.

We've done a bit of tweaking with the Twist application here, but nothing too extreme. This is not unlike the tweaking that you'll need to do with your own application when you add in these foundational multitenancy features.

We now have a way for people to sign up for an account which has a unique subdomain. We've purposely not made names unique here because we're instead making the `subdomain` for the account the unique field.

Let's commit this change now:

```
git add .
git commit -m "Added uniqueness validation for subdomain"
```

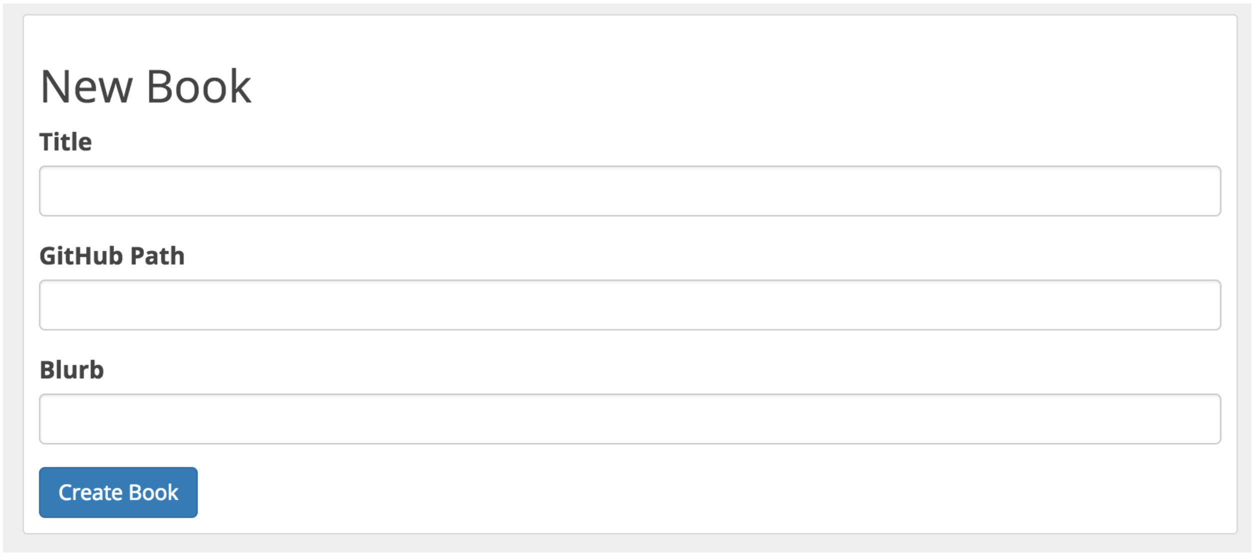
Now that we have accounts, let's work on a way to tie accounts and their books together.

Tying books to accounts

Accounts are just another resource in Twist at the moment and they don't serve any real purpose. This section will be all about fixing that. After users have signed up for an account in Twist, their next step will be to create books for that account which will be unique to that account. This is the first step in many towards our goal of making Twist properly mulitenanted.

Books may come from many sources, but in the interest in keeping this section short and not making the rest of this book about how to import other books into Twist, we'll only be looking at importing books from public GitHub repositories. And to keep it even shorter, we'll be focussing on just the *one* public GitHub repository: `radar/markdown_test`.[8]

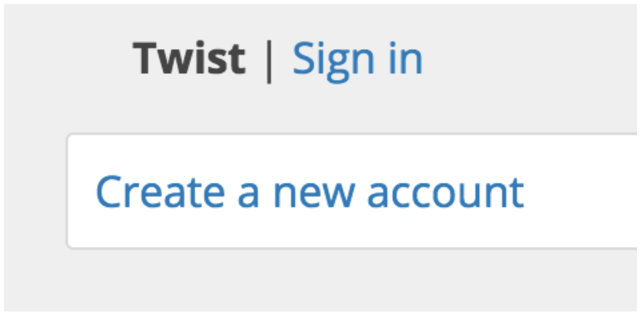
The form to create new books will look like this:



New book

How will users get to this particular form? Well first of all, when they go to an account's subdomain (i.e. test.example.com) they shouldn't see the "Create a new account" link, as they do now:

[8]Private accounts that wish to use Twist must first add the `twist-fetcher` user to their book's GitHub repo. It's a bit of a convoluted process, so I've left this out on purpose.



Create a new account link

Instead, they should see a list of their account's books and then if they're an owner they should be allowed to create a new book. Before we go adding the form to create a new book, we'll add a new `HomeController` specifically for accounts.

Let's start there and then we'll move right into adding new books to accounts.

Book creation groundwork

Let's start by adding a feature to create a new book for a particular account. We'll create a new file at `spec/features/accounts/adding_books_spec.rb` and fill it with this content:

spec/features/accounts/adding_books_spec.rb

```ruby
require "rails_helper"

feature "Adding books" do
  let(:account) { FactoryGirl.create(:account) }

  context "as the account's owner" do
    before do
      login_as(account.owner)
    end

    it "can add a book" do
      visit root_url(subdomain: account.subdomain)
      click_link "Add Book"
      fill_in "Title", with: "Markdown Book Test"
      fill_in "GitHub Path", with: "radar/markdown_book_test"
      click_button "Add Book"
      expect(page).to have_content(
        "Markdown Book Test has been enqueued for processing."
      )
    end
  end
end
```

We're putting this feature file inside the `spec/features/accounts` directory, because it involves an action within the Twist application that requires an account. The feature tests that an "Add Book" link is clickable when an account owner visits their subdomain's root path, and then that they can go through the motions of creating a book through that form.

The first line inside this feature uses an `account` factory from `FactoryGirl`, which doesn't exist yet. If we try to run our test with `bundle exec rspec spec/features/accounts/adding_books_spec.rb`, we'll see this error:

```
Failure/Error: let(:account) { FactoryGirl.create(:account) }
  ArgumentError:
  Factory not registered: account
```

Let's add this factory to the `spec/support/factories` directory now, in a brand new file:

spec/support/factories/account_factory.rb

```ruby
FactoryGirl.define do
  factory :account do
    sequence(:name) { |n| "Test Account ##{n}" }
    sequence(:subdomain) { |n| "test#{n}" }
    association :owner, :factory => :user
  end
end
```

This will allow us to create new accounts – along with some owners for those accounts! – in our tests whenever we feel like it by calling this factory. The `sequence` calls here will generate unique numeric values (starting at 1) for their `n` variables, which will allow us to create accounts using this factory without worrying about having to also ensure their subdomains are unique. The `owner` association here will use the user factory which is already defined over in `spec/support/factories/user_factory.rb`:

spec/support/factories/user_factory.rb

```ruby
FactoryGirl.define do
  factory :user do
    sequence(:email) { |n| "user#{n}@example.com" }
    password "password"
  end
end
```

If you're not using Factory Girl inside your own application, you could setup a test helper to do something similar for you:

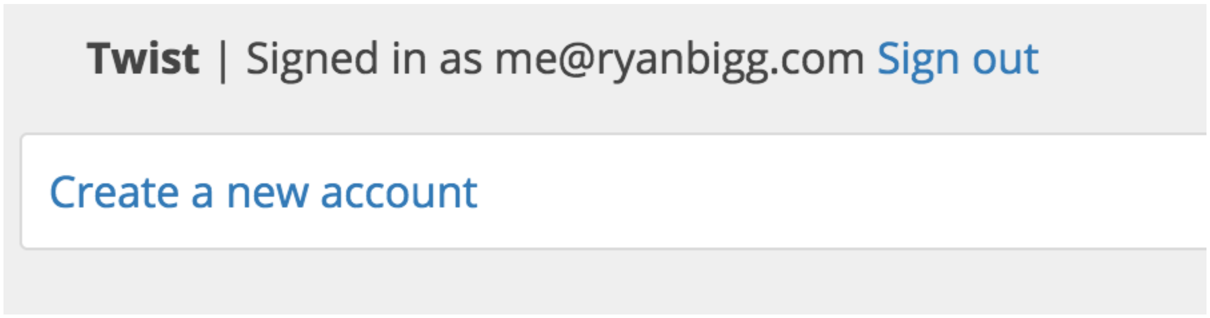
```ruby
def create_account
  account = Account.create(
    name: "Test Account",
    subdomain: "test"
  )

  account.owner = User.create(
    email: "test@example.com",
    password: "password"
  )

  account
end
```

Of course, this won't let you create more than one account at a time. That part is for you to figure out if you choose to go down that path.

Running the test again will show it getting a little further; it will be able to create the account and visit the root path, but it won't be able to click the "Add Book" link:

```
Failure/Error: click_link "Add Book"
Capybara::ElementNotFound:
  Unable to find link "Add Book"
```

If you start up `rails server` and go to an account's subdomain (i.e. http://test.lvh.me:3000), you'll see exactly what this test is seeing:



Twist | Signed in as me@ryanbigg.com Sign out

Create a new account

Incorrect message at subdomain root

When someone goes to the root of their account's subdomain, we shouldn't prompt them to create an account for hopefully obvious reasons. Instead, we should show them a different page entirely that shows the user the list of books for their account.

We can do this by simply adding a new route above the existing `root` route that defines a new `root` route that is only used when the application is being accessed from a subdomain. But we can't have *two* root routes can we? Yes we can: with routing constraints.

Adding a routing constraint

A routing constraint allows for conditional matching of Rails routes. Constraints can work on any part of a Rails request, including subdomain. We'll use a routing constraint here to show a different root path if the user is viewing the application from a subdomain.

The best way to explain how these work is to show how these work. Let's open our routes and add this code before the original `root` route in this file:

config/routes.rb

```ruby
constraints(SubdomainRequired) do
  scope module: "accounts" do
    root to: "books#index", as: :account_root
  end
end
```

The `constraints` block defines routes that will only be matched if the constraint returns `true`. There's only one route defined in there: a route to send `root` requests to the `Account::BooksController`'s `index` action. If this `constraint` is listed after the non-constrainted `root` route, the non- constrainted version will be matched first and the `root` route within the constraint will be completely ignored. The order of routes in the routes file is very important!

The `SubdomainRequired` class isn't defined yet, but it can be defined in a new file at `lib/constraints/subdomain_required.rb` like this:

lib/constraints/subdomain_required.rb

```
class SubdomainRequired
  def self.matches?(request)
    request.subdomain.present? && request.subdomain != "www"
  end
end
```

A constraint works by inspecting incoming requests and seeing if they match the specified criteria. In this `SubdomainRequired` constraint, we're seeing if a subdomain is present and if it's not `"www"`. If the `matches?` method returns `true` here, then the routes in `config/routes.rb` inside the `constraints` block will match and we'll be able to have the constrained root route match before its unconstrained cousin.

We'll need to require this file in `config/routes.rb`, which we can do by adding this line as the first line in that file:

config/routes.rb

```
require "constraints/subdomain_required"
```

If this constraint is in place correctly, our test will now fail with this:

```
Failure/Error: visit root_url(subdomain: account.subdomain)
ActionController::RoutingError:
  uninitialized constant Accounts
```

We'll need to create this module namespace before we can continue. All the controllers which perform actions on objects inside an account should namespaced.

These controllers are:

- `BooksController`
- `ChaptersController`
- `CommentsController`
- `NotesController`

Moving controllers into the account namespace

The first step to moving the `BooksController` into the `Accounts` namespace is to rename the file from `app/controllers/books_controller.rb` to `app/controllers/accounts/books_controller.rb`. The next step is to wrap everything in this file inside an `Accounts` module. We'll also change the class that the controller inherits from to `Accounts::BaseController`, so that we can provide account-specific functionality for all controllers that inherit from `Accounts::BaseController`.

app/controllers/accounts/books_controller.rb

```ruby
module Accounts
  class BooksController < Accounts::BaseController
    #...
  end
end
```

Wrapping the class in a `module` definition like this is so that the `Accounts` constant is defined, as well as the `Accounts::BooksController` constant that will be searched for when the constrained `root` route is hit. We're naming this module `Accounts` rather than `Account` so that it doesn't conflict with our class called `Account`.

The `Accounts::BaseController` class doesn't exist yet. What this will do is provide a place to put methods that can be shared between all of the classes that inherit from `Accounts::BaseController`. Let's create it now:

app/controllers/accounts/base_controller.rb

```ruby
module Accounts
  class BaseController < ApplicationController

  end
end
```

The test file at `spec/controllers/books_controller_spec.rb` will need to move to `spec/controllers/accounts/books_controller_spec.rb` so that it's consistent with the naming changes we've made. The beginning of this file will need to change so that it references the right constant:

spec/controllers/accounts/books_controller_spec.rb

```ruby
require 'rails_helper'

describe Accounts::BooksController do
#...
```

The next thing to do is to move the routes for books into the constraint. Let's rework our routes so that it then looks like this:

config/routes.rb

```ruby
require "constraints/subdomain_required"

Twist::Application.routes.draw do
  devise_for :users

  constraints(SubdomainRequired) do
    scope module: "accounts" do
      root to: "books#index", as: :account_root

      notes_routes = lambda do
        collection do
          get :completed
        end

        member do
          put :accept
          put :reject
          put :reopen
        end

        resources :comments
      end

      resources :books do
        member do
          post :receive
        end

        resources :chapters do
          resources :elements do
            resources :notes
          end

          resources :notes, &notes_routes
        end

        resources :notes, &notes_routes
      end
    end
  end
end

```

```
  root to: "home#index"
  get "/accounts/new", to: "accounts#new", as: :new_account
  post "/accounts", to: "accounts#create", as: :accounts

  get "signed_out", to: "users#signed_out"
end
```

This reworking has moved all the books routes, including the nested routes for chapters and notes, into the `constraints` and `scope` blocks. Books in our application are now only ever accessible within the context of an account's subdomain. That isn't to say yet that books belonging to particular accounts are only available in that account's subdomain; that is not true. All books are accessible at the moment. That something we'll be fixing up in a later chapter.

By moving all these routes over, Twist is going to expect to find their matching controllers inside the `Accounts` namespace too. We'll need to move the `ChaptersController`, `CommentsController`, and `NotesController` into the `Accounts` namespace too, as well as make them inherit from `Accounts::BaseController`.

app/controllers/accounts/chapters_controller.rb

```
module Accounts
  class ChaptersController < Accounts::BaseController
    #...
  end
end
```

app/controllers/accounts/comments_controller.rb

```
module Accounts
  class CommentsController < Accounts::BaseController
    #...
  end
end
```

app/controllers/accounts/notes_controller.rb

```
module Accounts
  class NotesController < Accounts::BaseController
    #...
  end
end
```

This moving makes sense. These controllers are all acting on chapters, comments or notes of books, which exist inside accounts. A book no longer exists outside the concept of an account.

Any actions inside controllers which inherit from `Accounts::BaseController` should require users to be signed in first. All of the above controllers have this line (or similar) inside of them:

```
before_action :authenticate_user!, except: [:receive]
```

Remove this line from all the controllers, and put it into `Accounts::BaseController`:

app/controllers/accounts/base_controller.rb

```
before_action :authenticate_user!
```

For `Accounts::BooksController`, the `receive` action will need to be accessible by the GitHub webhooks. The webhooks don't authenticate as a user on Twist.[9] Let's use a `skip_before_action` to skip the authentication requirement for this action:

```
skip_before_action :authenticate_user!, only: [:receive]
```

Moving the views

The next thing is to move over all the views from `app/views/books/`, `app/views/chapters`, `app/views/comments`, `app/views/elements` and `app/views/notes` to their corresponding directories inside `app/views/accounts/`. Because these paths have changed, places that refer to partials in our views need to change. You could look through these and find them yourself, but if you don't feel like doing it, here's what needs changing.

Change this:

[9] Although you *can* set a `secret` parameter to be passed through from GitHub to verify that GitHub is making the request. I've been lazy in developing Twist and haven't done this yet.

app/views/accounts/elements/_element.html.erb

```
<%= render "notes/button", element: element %>
```

To this:

app/views/accounts/elements/_element.html.erb

```
<%= render "accounts/notes/button", element: element %>
```

Make the same change in `elements/_img` too. Change this:

app/views/accounts/elements/_img.html.erb

```
<%= render "notes/button", element: element %>
```

To this:

app/views/accounts/elements/_img.html.erb

```
<%= render "accounts/notes/button", element: element %>
```

Change this:

app/helpers/elements_helper.rb

```
render("elements/#{partial}", element: element, show_notes: show_notes)
```

To this:

app/helpers/elements_helper.rb

```
render("accounts/elements/#{partial}", element: element, show_notes: show_notes)
```

Change this:

app/helpers/elements_helper.rb

```
partial = Rails.root + "app/views/elements/_#{element.tag}.html.erb"
```

To this:

app/helpers/elements_helper.rb

```
partial = Rails.root + "app/views/accounts/elements/_#{element.tag}.html.erb"
```

Change this:

app/views/accounts/notes/new.js.erb

```
<% partial = render('notes/form') %>
```

To this:

app/views/accounts/notes/new.js.erb

```
<% partial = render('accounts/notes/form') %>
```

Change this:

app/views/notes/show.html.erb

```
<%= render "comments/form" %>
```

To this:

app/views/notes/show.html.erb

```
<%= render "accounts/comments/form" %>
```

If you tried viewing a book without these changes there would be missing template errors aplenty. These fixes are to prevent that from happening.

Let's run our tests now to see if we broke anything with `bundle exec rspec spec`.

```
rspec ./spec/features/accounts/adding_books_spec.rb:11
rspec ./spec/features/books_spec.rb:12
rspec ./spec/features/books_spec.rb:29
rspec ./spec/features/comments_spec.rb:29
rspec ./spec/features/comments_spec.rb:38
rspec ./spec/features/comments_spec.rb:48
rspec ./spec/features/notes_spec.rb:10
rspec ./spec/features/notes_spec.rb:38
rspec ./spec/features/notes_spec.rb:89
rspec ./spec/features/notes_spec.rb:75
rspec ./spec/features/notes_spec.rb:82
```

We broke a few things with our reshuffling. It shouldn't be too hard to fix these, given that our changes were so minor. Let's take a break from the "Adding Books" feature and take a look at the books, comments and notes failing tests.

Fixing the tests

We'll start with the `books_spec.rb` tests. All of these tests fail because we changed the path:

```
Failure/Error: visit book_path(book)
ActionController::RoutingError:
  No route matches [GET] "/books/markdown-book-test"
```

These tests are trying to navigate to a route that no longer exists. Well, the route exists, it just isn't accessible without a subdomain anymore. The fault in this test is in these lines:

spec/features/books_spec.rb

```
let!(:author) { create_author! }
let!(:book) { create_book! }

before do
  actually_sign_in_as(author)
end
```

This test is signing in as an author, but it really should be signing in as an account owner. Authors in the old Twist system have permission to Accept or Reject comments, but now only account owners will have that ability. Let's fix this up now by changing the top of this test to this:

spec/features/books_spec.rb

```
let!(:account) { FactoryGirl.create(:account) }
let!(:book) { create_book! }

before do
  login_as(account.owner)
  set_subdomain(account.subdomain)
end
```

We've snuck in a reference to a method called `set_subdomain`, which is not defined yet. If we run our tests, they'll even tell us that:

```
Failure/Error: set_subdomain(account.subdomain)
  NoMethodError:
    undefined method `set_subdomain' for #...
```

Rather than passing subdomains through to routing helpers all the time, we'll be using this method to set the stage for future requests in our tests. We'll define this new test helper like this:

spec/support/subdomain_helpers.rb

```ruby
module SubdomainHelpers
  def set_subdomain(subdomain)
    site = "#{subdomain}.lvh.me"
    Capybara.app_host = "http://#{site}"
    Capybara.always_include_port = true

    default_url_options[:host] = "#{site}"
  end
end

RSpec.configure do |c|
  c.include SubdomainHelpers, type: :feature

  c.before type: :feature do
    Capybara.app_host = "http://lvh.me"
  end
end
```

This helper allows us to set which subdomain our tests will use, and will stop our tests from making real requests out to `lvh.me`. Instead, the tests will go to the application which gets spawned as part of the test process.

The `Capybara.app_host` is reset before every test to ensure that it's not left using a subdomain from a previous test.

When we run this test again, it will now pass:

```
2 examples, 0 failures
```

Well, that was very easy! All we had to do was to change the test to login as the account owner and access the path under the account's subdomain.

Next up is the `comments_spec.rb` tests. All of these tests fail for a very similar reason to `books_-spec.rb`.

```
Failure/Error: visit book_note_path(book, note)

ActionController::RoutingError:
  No route matches [GET] "/books/markdown-book-test/notes/1"
```

The fault of the test is in these lines:

spec/features/comments_spec.rb

```ruby
context "as an author" do
  before do
    actually_sign_in_as(author)
    visit book_note_path(@book, note)
    fill_in "comment_text", :with => comment_text
  end
```

It's signing in as an author, which is defined at the top of `comments_spec.rb` like this:

```ruby
let!(:author) { create_author! }
```

We no longer want to sign in as an author, but instead sign in as an account owner. Let's change this line at the top of the spec:

```ruby
let!(:author) { create_author! }
```

To this:

```ruby
let!(:account) { FactoryGirl.create(:account) }
```

In the test, we'll then sign in as this account's owner and navigate to the same path that the test used to ask for, but within the context of that account's subdomain. To accomplish that goal, we'll change these lines in the spec:

```ruby
context "as an author" do
  before do
    actually_sign_in_as(author)
    visit book_note_path(book, note)
    fill_in "comment_text", :with => comment_text
  end
```

To these:

```
context "as an account's owner" do
  before do
    login_as(account.owner)
    set_subdomain(account.subdomain)
    visit book_note_url(book, note)
    fill_in "comment_text", :with => comment_text
  end
```

We're now logging in as the account's owner. If we re-run our test again, we'll see it gets closer to completion, but not quite there yet:

```
Failure/Error: click_button "Accept"
Capybara::ElementNotFound:
  Unable to find button "Accept"
```

The issue is these lines:

app/views/accounts/comments/_form.html.erb

```
<% if current_user.author? %>
  <% if @note.completed? %>
    <%= f.submit "Reopen", :class => "reopen-button" %>
  <% else %>
    <%= f.submit "Accept", :class => "btn btn-primary", :tabindex => 2 %>
    <%= f.submit "Reject", :class => "btn btn-danger", :tabindex => 3 %>
  <% end %>
<% end %>
```

The check that wraps these lines still checks if the current user is an author. We should change this to check if the user is an owner using our `owner?` helper because we want owners of accounts to be able to perform these actions.

app/views/accounts/comments/_form.html.erb

```
<% if owner? %>
  <% if @note.completed? %>
    <%= f.submit "Reopen", class: "reopen-button" %>
  <% else %>
    <%= f.submit "Accept", :class => "btn btn-primary", :tabindex => 2 %>
    <%= f.submit "Reject", :class => "btn btn-danger", :tabindex => 3 %>
  <% end %>
<% end %>
```

The `owner?` method isn't defined yet. Rather than just defining this one method, we'll define two new methods. Both of these will go into `Accounts::BaseController`

app/controllers/accounts/base_controller.rb

```ruby
module Accounts
  class BaseController < ApplicationController
    before_action :authenticate_user!

    def current_account
      @current_account ||= Account.find_by!(subdomain: request.subdomain)
    end
    helper_method :current_account

    def owner?
      current_account.owner == current_user
    end
    helper_method :owner?
  end
end
```

The `owner?` method checks to see if the current account's owner is the current user. If that is the case, then these "Reopen", "Accept" and "Reject" buttons will show up on the comment form. The `current_account` method has been added here too as we might require it later on.

This won't quite be enough to make the test pass:

```
Failure/Error: expect(page).to have_content("Note state changed to Accepted")

  expected to find text "Note state changed to Accepted"
    in "Comment has been created."
```

The code that handles the note state changing lives in the `Accounts::CommentsController`, and goes like this:

app/controllers/accounts/comments_controller.rb

```ruby
def check_for_state_transition!
  if current_user.author?
    if params[:commit] == "Accept"
      @note.accept!
      notify_of_note_state_change("Accepted")
    elsif params[:commit] == "Reject"
      @note.reject!
      notify_of_note_state_change("Rejected")
    elsif params[:commit] == "Reopen"
```

```
      @note.reopen!
      notify_of_note_state_change("Reopened")
    end
  end
end
```

This is also checking if the user is an author! Let's change it to check if the user is the owner. We'll also tidy it up to not have the whole block of code wrapped inside an `if`:

```
def check_for_state_transition!
  return unless owner?

  if params[:commit] == "Accept"
    @note.accept!
    notify_of_note_state_change("Accepted")
  elsif params[:commit] == "Reject"
    @note.reject!
    notify_of_note_state_change("Rejected")
  elsif params[:commit] == "Reopen"
    @note.reopen!
    notify_of_note_state_change("Reopened")
  end
end
```

This code is now slightly neater due to one less `end` *and* it has the right check in it. It's checking if the current user is the owner of the account. If the user is the owner, then they should be able to make the note's state changed.

Do our tests pass now? They sure do!

```
3 examples, 0 failures
```

Wonderful!

Let's take a look at the other failures over in `spec/features/notes_spec.rb` before we get back to our original test. The tests in `notes_spec.rb` are failing for almost exactly the same reason as the ones back in `comments_spec.rb`:

```
Failure/Error: visit book_path(@book)
ActionController::RoutingError:
  No route matches [GET] "/books/rails-3-in-action"
```

Let's make the same kind of changes to this spec too. We'll start by changing these lines in the test:

spec/features/notes_spec.rb

```ruby
let(:author) { create_author! }
before do
  create_book!
  login_as(author)
end
```

To this:

spec/features/notes_spec.rb

```ruby
let(:account) { FactoryGirl.create(:account) }
let(:book) { create_book! }

before do
  login_as(account.owner)
  set_subdomain(account.subdomain)
end
```

When we run our test again, we'll see that it can't find the `author`:

```
Failure/Error: expect(page).to
  have_content("#{author.email} commented less than a minute ago")
NameError:
  undefined local variable or method `author' ...
```

This issue is easier to fix than the previous issue: change all references to `author` in this test to `account.owner` instead.

After that change, we can run `bundle exec rspec spec/features/notes_spec.rb` again and see this:

```
5 examples, 0 failures
```

Excellent. This test is working again. If we run both of these tests again with `bundle exec rspec spec/features/comments_spec.rb spec/features/notes_spec.rb` we'll see that they're working:

```
8 examples, 0 failures
```

If we run all the tests, we'll only see our `adding_books_spec.rb` failing now. This means that we've now brought our application back to a good, stable place. Let's move back to the `adding_books_-spec.rb` now and fix that one up too.

Getting back to the test

Back in our `adding_books_spec.rb`, we're still using the old way of referencing a subdomain:

spec/features/accounts/adding_books_spec.rb

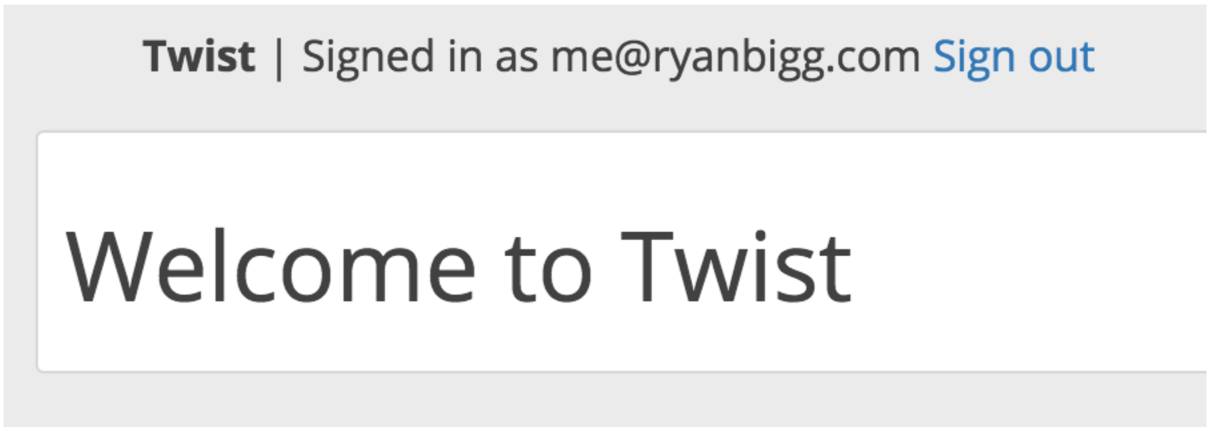
```
visit root_url(subdomain: account.subdomain)
```

Let’s change this to use the new way:

```
set_subdomain(account.subdomain)
visit root_url
```

If we don’t do this, we might see that the test can’t find the add books form. This is because requests made to the server will be going to `example.com` by default, rather than the domain specified by our `session_store.rb` configuration, which is `lvh.me`.

If you refresh your browser which is visiting an account’s subdomain, you’ll see “Welcome to Twist”:



Welcome to Twist

Nothingness

If you run your test at this point, it still won’t be able to find the link:

```
Failure/Error: click_link "Add Book"
Capybara::ElementNotFound:
  Unable to find link "Add Book"
```

Why is this? Well, we can find out by looking at `log/development.log`, which will tell us this:

```
Processing by Accounts::BooksController#index as HTML
  User Load ...
  Book Load (0.2ms)  SELECT "books".* FROM "books" WHERE "books"."hidden" = 'f'
  Rendered accounts/books/index.html.erb within layouts/application (0.9ms)
  Completed 200 OK in 9ms (Views: 6.2ms | ActiveRecord: 0.4ms)
```

Scoping books to their accounts

The `SELECT "books" ...` query here doesn't just load books for one account, but rather loads all the books for the application. The scoping of this to load only particular books is intentionally left until a later chapter.

The request was made to the right controller, and it looks like it rendered `app/views/accounts/books/index.html.erb`, so let's look at what's in that file:

app/views/accounts/books/index.html.erb

```
<div class='row'>
  <div class='col-md-8 content'>
    <h1>Welcome to Twist</h1>
    <% @books.each do |book| %>
      <h2><%= link_to book.title, book %></h2>
      <span class='blurb'><%= book.blurb %></span>
    <% end %>
  </div>

  <div id='sidebar' class='col-md-4'>
    <% if current_user.author? %>
      <ul>
        <li><%= link_to "Add Book", new_book_path %></li>
      </ul>
    <% end %>
  </div>
</div>
```

There's our culprit! The page has code in its sidebar that checks if the current user is an author. This is a leftover feature from when Twist was just an application that a single author would use. Let's change this code to check if the current user is the owner for an account.

Let's change this code in the view:

app/views/accounts/books/index.html.erb

```
<div id='sidebar' class='col-md-4'>
  <% if current_user.author? %>
    <ul>
      <li><%= link_to "Add Book", new_book_path %></li>
    </ul>
  <% end %>
</div>
```

To this:

app/views/accounts/books/index.html.erb

```
<div id='sidebar' class='col-md-3'>
  <% if owner? %>
    <%= link_to "Add Book", new_book_path %>
  <% end %>
</div>
```

When we run our test again, we'll see it can now find the "Add Book" link. It will fail a few steps down from that:

```
Failure/Error: expect(page).to
  have_content("Markdown Book Test has been enqueued for processing.")
  expected to find text "Markdown Book Test has been enqueued for processing."
  in "...Thanks! Your book is now being processed. Please wait."
```

This is happening because the `flash[:notice]` in the controller is different to what we expect in the test. Let's change the `flash[:notice]` inside the `create` action of `Accounts::BooksController` to match what the test is expecting:

app/controllers/accounts/books_controller.rb

```
flash[:notice] = "#{@book.title} has been enqueued for processing."
```

When we run our test again, it will pass this time:

```
1 example, 0 failures
```

This is a step in the right direction! We're now able to able to create books for an account as that account's owner. Later on, we'll make sure that only an account's own books appear on that account's page, but for now this feature is good enough.

Now that we've gone through and fixed up a bunch of tests, let's see if all of them are passing by running `bundle exec rspec spec`:

```
52 examples, 0 failures
```

Hooray! All of the tests are indeed passing. There's one pending test that we don't need in `spec/models/account_spec.rb`, so let's delete that file now.

We're now done here. Let's make a commit:

```
git add .
git commit -m "Books can now be added to accounts"
```

Summary

We've started down the road of making Twist a multitenanted application.

The first thing we changed was to add an `Account` model which will later on provide a good basis for the multitenancy features of our application. We've started down that path by linking the `Book` model to the `Account` model, but there's a ways to go yet.

We've also moved a lot of our routes into a subdomain-constrained block. This means that we'll always have a subdomain present for these routes, and that means that we can figure out what the current account is by checking the subdomain.

An account is not very useful if only the owner of that account can read the books they upload! The next thing we're going to look at is adding invitations to Twist. These invitations will allow the account owner to invite other users to their account and that will give those users permission to read books from that account.