Multitenancy *with Rails*

Building industrial strength Software-as-a-Service Applications

by Ryan Bigg

Multitenancy with Rails

And subscriptions too!

Ryan Bigg

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

This version was published on 2015-11-24

Contents

1. Laying the foundations

Now's the time where we're going to create the foundations of the subscription engine. We'll need to create a brand new engine using the `rails plugin new` generator, which will create our engine and create a dummy application inside the engine. The dummy application inside the engine will be used to test the engine's functionality, pretending for the duration of the tests that the engine is actually mounted inside a real application.

We're going to be building an engine within this chapter rather than an application for three reasons: engines are cool, we may want to have this same functionality inside another app later on and lastly, it lets us keep the code for subscriptions separate from other code.

If you don't know much about engines yet, then we recommend that you read the official Getting Started With Engines guide[1] at http://guides.rubyonrails.org/engines.html. We're going to skim over the information that this guide covers, because there's no use covering it twice!

Once we're done setting up the engine and the test application, we'll write the first feature for the engine: account sign up. By having accounts within the engine, it provides the basis for scoping resources for whatever application the engine is embedded into.

Then, when we're done doing that, we'll set it up so that accounts will be related to an "owner" when they are created. The owner will (eventually) be responsible for all admin-type actions within the account. During this part of the chapter, we'll be using Warden to manage the authentication proceedings of the engine.

At the end of the chapter, we'll write another group of features which will handle user sign in, sign up and sign out which will provide a lovely segue into our work in the next chapter, in which we'll actually do some scoping!

So let's begin by laying the foundations for this engine.

> **This book has source code on GitHub!** If you want to see some code examples and compare them to what you have, you can go view them here: https://github.com/radar/saas_book_examples.

If you're reading this book to add a subscription feature to an existing application, it'll be easier for you to build the features in this book in that application, rather than a new engine.

Where this book references namespaced (`Subscribem::`) models, just use their non-namespaced variants: `Account` instead of `Subscribem::Account`.

You'll likely have your own user model too, which is alright. You can use that user model in place of the `Subscribem::User` model in this chapter. You won't need to install Warden, either, as that (or something like it) should already be setup.

[1] http://guides.rubyonrails.org/engines.html

1.1 Building an engine

> **Ruby and Rails versions**
>
> This book will be using Ruby 2.2.2 and Rails 4.2.2. Please make sure that you're also using these versions, otherwise you may run into difficulties.

We're going to call this engine "subscribem", because we want to subscribe them. It's a cheeky pun!

Getting the basic requirements for a new engine in Rails is as easy as getting them for a new application. To generate this new engine, run this command:

```
rails plugin new subscribem --full --mountable \
                            --dummy-path spec/dummy --skip-test-unit
```

This is actually the command to generate a new plugin, but we're going to make it generate an engine instead.

The `--full` option makes the plugin an engine with things like the `app/controllers` and `app/models` directory. The `--mountable` option isolates the engine's namespace, meaning that all the controllers and models from this engine will be isolated within the namespace of the engine. For instance, the `Account` model we'll create later will be called `Subscribem::Account`, and not simply `Account`.

The engine comes with a dummy application, located at `spec/dummy` because we told it to do that with the `--dummy-path` option. This dummy application is just a bare-bones Rails application that can be used to test the engine as if it was mounted inside a real application.

1.2 Setting up a testing environment

The testing environment for this gem will include the RSpec and Capybara gems.

To add RSpec as a dependency of our engine, we're going to add it to the `subscribem.gemspec` file, rather than the `Gemfile`. The engine itself is actually a gem (and will be installed into applications *as a gem*) and therefore has a lovely `subscribem.gemspec` file. The dependencies specified in this file, and this file only, are loaded along with this gem when it's embedded into an application. That's why we need to put the dependencies inside `subscribem.gemspec`.

To add a dependency for RSpec to the gem, add this line inside `subscribem.gemspec`, inside the `Gem::Specification.new` block, underneath the `sqlite3` dependency:

```
s.add_development_dependency "rspec-rails", "3.3.2"
```

To add the dependency for Capybara, add this line underneath that previous line:

```
s.add_development_dependency "capybara", "2.4.4"
```

To install these gems if they're not available already on your system, run `bundle install`. This command will reference the `Gemfile` for the engine, which in turn references the `gemspec`, because the `Gemfile` is defined like this (comments stripped):

Gemfile

```
source "http://rubygems.org"
gemspec
```

This will also install the `rails` and `sqlite3` gems and their dependencies because they're specified in the `subscribem.gemspec` file too.

With RSpec and Capybara installed, we can now set up RSpec by running this command:

```
rails g rspec:install
```

This will set up RSpec as if it were inside an application, which is *almost* correct for the purposes of being an engine. There's one small problem, which is the line that requires an application's `config/environment.rb` file. Currently, the relevant line inside `spec/rails_helper.rb` is this:

```
require File.expand_path("../../config/environment", __FILE__)
```

The `config/environment.rb` file doesn't live two directories up, but rather inside the `spec/dummy` directory. Therefore this line should be changed to this:

```
require File.expand_path("../dummy/config/environment",  __FILE__)
```

We'll also need to fix the line for requiring `spec/support` files, since it's currently looking into the root of the rails app:

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

We want it to be looking in the `support` directory which is in the same directory as this `rails_helper.rb` file. Therefore, this line should be changed to this:

```
Dir[File.dirname(__FILE__) + "/support/**/*.rb"].each {|f| require f}
```

We will also need to add a require for Capybara to the top of this file, underneath the require for `rspec/rails` and `rspec/autorun`:

```
require File.expand_path("../dummy/config/environment",  __FILE__)
require "rspec/rails"
require "capybara/rspec"
```

Finally, to make our engine *always* use RSpec, we can add this code into the `Subscribem::Engine` class definition inside `lib/subscribem/engine.rb`:

```
module Subscribem
  class Engine < Rails::Engine
    ...
    config.generators do |g|
      g.test_framework :rspec, :view_specs => false
    end
  end
end
```

Every time we generate a new model, controller or helper with Rails it will generate RSpec tests for it, rather than the typical TestUnit tests.

Now we've got a great foundation for our engine and we'll be able to write our first test to test that users can sign up for a new account.

1.3 Writing the first feature

The first thing we want our new engine to be able to do is to take sign ups for the new accounts, given that accounts are the most important thing within our system.

The process of signing up 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 successfully 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

```
1  require "rails_helper"
2  feature "Accounts" do
3    scenario "creating an account" do
4      visit subscribem.root_path
5      click_link "Account Sign Up"
6      fill_in "Name", :with => "Test"
7      click_button "Create Account"
8      success_message = "Your account has been successfully created."
9      expect(page).to have_content(success_message)
10   end
11 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: visit subscribem.root_path
NoMethodError:
  undefined method `root_path' for #<ActionDispatch::...>
```

The `subscribem` method inside the spec is an `ActionDispatch::Routing::RoutesProxy` object which provides a proxy object to the routes of the engine. Calling `root_url` on this routing proxy object *should* return the root path of the engine. Calling `root_path` without the `subscribem` prefix – without the routing proxy – in the test will return the root of the application.

Let's see if we can get this test to pass now.

Implementing account sign up

The reason that this test is failing is because we don't currently have a `root` definition inside the engine's routes. So let's define one now inside `config/routes.rb` like this:

```
Subscribem::Engine.routes.draw do
  root "dashboard#index"
end
```

The dashboard controller will serve as a "welcome mat" for accounts inside the application. If a user has not yet created an account, they should see the "New Account" link on this page. However, right now the `DashboardController` and `index` action's template for this controller don't exist. To create the controller, run this command:

```
rails g controller dashboard
```

Controllers are automatically namespaced

Note here that due to the `isolate_namespace` call within the `Subscribem::Engine` class, this controller will be namespaced within the `Subscribem` module automatically, meaning it will be called `Subscribem::DashboardController`, rather than just `DashboardController`. This would keep it separate from any potential `DashboardController` that could be defined within an application where the engine is embedded into.

Rather than generating the normal Rails tests within the `test` directory, this command will generate new RSpec files in the `spec` directory. This is because of that line that we placed within `lib/subscribem/engine.rb`.

With the controller generated, the next step will be to create a view which contains a "Account Sign Up" link. Create a new file now at `app/views/subscribem/dashboard/index.html.erb` and put this content inside it:

```
<%= link_to "Account Sign Up", sign_up_path %>
```

If we were to run the test again at this point – with `rspec spec/features/accounts/sign_up_spec.rb` – we will see the test error because there's no `sign_up_path` defined:

```
undefined local variable or method `sign_up_path' for ...
```

This means that we're going to need to define a route for this also in `config/routes.rb`, which we can do with this line:

```
get "/sign_up", :to => "accounts#new", :as => :sign_up
```

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

```
rails g controller accounts
```

The `sign_up_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/subscribem/accounts/new.html.erb`:

```
<h2>Sign Up</h2>
<%= form_for(@account) do |account| %>
  <p>
    <%= account.label :name %><br>
    <%= account.text_field :name %>
  </p>
  <%= account.submit %>
<% end %>
```

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

```
def new
  @account = Subscribem::Account.new
end
```

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

```
Failure/Error: click_link "Account Sign Up"
NameError:
  uninitialized constant Subscribem::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 `Subscribem::Account` and with it a migration which will create the `subscribem_accounts` table that contains a `name` field. To run this migration now for the dummy application, run this command:

```
rake db:migrate
```

What next? Find out by running the spec with `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/subscribem/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 `config/routes.rb` now using this code:

```
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 `Subscribem::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 `app/controllers/subscribem/accounts_controller.rb` now:

```
def create
  account = Subscribem::Account.create(account_params)
  flash[:success] = "Your account has been successfully created."
  redirect_to subscribem.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 `Subscribem::Account` object for it. It'll also set a success message for the next request, and redirect back to the root path for the engine. In an application, we would probably use `redirect_to '/'` here, but since we're in an engine we're going to want to explicitly link back to *the engine's root*, not the application's.

Parameters within Rails 4 are not automatically accepted (thanks to the `strong_parameters` gem), 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:

```
def create
  account = Subscribem::Account.create(account_params)
  flash[:success] = "Your account has been successfully created."
  redirect_to subscribem.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`.

```
Failure/Error:
  page.should have_content("Your account has been successfully created.")
  expected there to be content "<success message>" in ...
```

Ah, not quite! The flash message isn't displaying for this action because it's not being rendered anywhere in the dummy application right now. They would typically be rendered in the layout, but there's no code to do that currently. This means that we should go ahead and add some code to do it so that our test can pass. Add these lines to the engine's layout:

app/views/layouts/subscribem/application.html.erb

```
1 <% flash.each do |k,v| %>
2   <div class='flash <%= k %>'><%= v %></div>
3 <% end %>
```

Now the flash should show up when you re-run the spec.

```
4 examples, 0 failures, 3 pending
```

Great! Now would be a good time to commit this to some sort of version control system so you've got a step-by-step account of what you've done, so go ahead and do that.

Feel free to remove the pending specs at your leisure too.

What we've done so far in this chapter is built the engine that will provide the grounding for account subscription to any application that the engine is added to. We've seen how to add some very, very basic functionality to the engine and now users will be able to create an account in the system.

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.

1.4 Associating accounts with their owners

An account's owner will be responsible for doing admin operations for that account. An owner is just a user for an account, and we're going to need another model to track users for accounts. It just makes sense that we'll need a `User` model within the engine.

When a user is signed in under an account, that'll be their active account. When they're signed in for this account, they'll be able to see all the resources for that account.

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.

These fields won't be stored on an `Subscribem::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 – `spec/features/accounts/sign_up_spec.rb` – 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

```
7  fill_in "Name", :with => "Test"
8  fill_in "Email", :with => "subscribem@example.com"
9  fill_in "Password", :with => 'password'
10 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 engine 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 subscribem@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/subscribem/accounts/new.html.erb`, so when you run this spec using `rspec spec/features/accounts/sign_up_spec.rb` you'll see this error:

```
Failure/Error: fill_in 'Email', :with => "subscribem@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/subscribem/accounts/new.html.erb`, add the fields for the owner using this code:

app/views/subscribem/accounts/new.html.erb

```
<p>
  <%= account.label :name %><br>
  <%= account.text_field :name %>
</p>
<%= account.fields_for :owner do |owner| %>
  <p>
    <%= owner.label :email %><br>
    <%= owner.email_field :email %>
  </p>
  <p>
    <%= owner.label :password %><br>
    <%= owner.password_field :password %>
  </p>
  <p>
    <%= owner.label :password_confirmation %><br>
    <%= owner.password_field :password_confirmation %>
  </p>
<% end %>
```

With the fields set up in the view, we're going to need to define the `owner` association within the `Subscribem::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 `Subscribem::Account` model definition:

```
belongs_to :owner, :class_name => "Subscribem::User"
accepts_nested_attributes_for :owner
```

The owner object for an `Subscribem::Account` will be an instance of the not-yet-existing `Subscribem::User` model, which will be used to keep track of users within the engine. Because there's now a `belongs_to :owner` association on the `Subscribem::Account`, we'll need to add an `owner_id` field to the `subscribem_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_subscribem_accounts owner_id:integer
```

Let's run the migration with `rake db:migrate` now so that we have this field available.

In order for the fields for the owner to appear on the new account form, we're going to need to build an associated `owner` object for the `@account` object inside the `new` action of `Subscribem::AccountsController`, which we can do with this code:

```
def new
  @account = Subscribem::Account.new
  @account.build_owner
end
```

Let's find out what we need to do next by running `rspec spec/features/accounts/sign_up_spec.rb`.

```
Failure/Error: click_link "Account Sign Up"
NameError:
  uninitialized constant Subscribem::Account::Subscribem::User
```

It seems like the `Subscribem::User` class is missing. This class is going to be just a model that will use the `has_secure_password` method provided by Rails to generate secure passwords. For our purposes, this model will need an email field and a `password_digest` field, the latter of which `has_secure_password` uses to store its password hashes.

Why not Devise?

Some people prefer using the Devise gem to set up authentication. You don't *always* need to use Devise, and so we're going down the alternate path here: building the authentication from scratch. This way, there will be no cruft and you will know how everything fits together.

Attempting to implement what we're doing with Devise will mostly be an exercise in frustration as there would need to be a lot of code to customize it to work the way that we want.

To generate this `Subscribem::User` model, run this command:

```
rails g model user email:string password_digest:string
```

Inside this new model, we'll need to add a call to `has_secure_password` and define its accessible attributes, which we can do by changing the whole file into this code:

app/models/subscribem/user.rb

```
1 module Subscribem
2   class User < ActiveRecord::Base
3     has_secure_password
4   end
5 end
```

Because we're using `has_secure_password`, we will also need to add the `bcrypt-ruby` gem to the list of dependencies for the engine. Let's add it now underneath the dependency for rails in `subscribem.gemspec`:

```
s.add_dependency "bcrypt", "3.1.10"
```

This gem will provide the password hashing code that `has_secure_password` uses to securely hash the passwords within our engine. Let's install this gem now:

```
bundle install
```

The `Subscribem::User` model has now been generated and that should mean the test should get further. Running `rspec spec/features/accounts/sign_up_spec.rb` again will result in a new error:

```
... Migrations are pending; run 'rake db:migrate RAILS_ENV=test'
```

This error is easy enough to fix, just run `rake db:migrate` again. There's no need to run it with the `RAILS_ENV` environment variable, because `rake db:migrate` as of Rails 4.1.0 will migrate both the development and test environments at the same time.

Running the spec again, you'll see that we've gotten past that point and now it's on the final line of the spec:

```
expected to find text "Signed in as subscribem@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 by implementing a way for a user to authenticate. We could use Devise, but as mentioned earlier, we're going an alternative route. Devise offers a lot of good features but along with those good features comes a lot of cruft that we don't yet need for this engine. So let's not use it in this circumstance and keep things dead simple. Instead, let's just use the underlying foundations of Devise: the Warden gem.

Authenticating with warden

The `warden` gem[2] was created by Daniel Neighman and is used as a general Rack session management framework, meaning it can be used in any Rack-based application to provide the kind of authentication features we're after.

It works by inserting a piece of middleware which sets up a Warden proxy object to manage a user's session. You can then call methods on this proxy object to authenticate a user. The authentication process is handled by what Warden refers to as "strategies", which themselves are actually quite simple to set up. We'll see strategies in use at the end of this chapter when we set up proper authentication for the user.

To install Warden, let's add it as a dependency to our gem by adding this line underneath the dependency for rails within `subscribem.gemspec`:

```
s.add_dependency "warden", "1.2.3"
```

Install it using the usual method:

```
bundle install
```

This gem will need to be required within the engine as well, which can be done inside `lib/subscribem/engine.rb`.

```
require "warden"
```

[2]The warden gem: https://github.com/hassox/warden

> **Why is this require here?**
> We *could* put the `require` for Warden within the file at `lib/subscribem.rb`, but due to how the engine is loaded by Rake tasks, that file is not required at all. This means that any `require` statements within that file would not be executed. Placing it within `lib/subscribem/engine.rb` will make it work for our test purposes, when we use the engine inside an application, and when we want to run Rake tasks on the engine itself.

Right now, all we're going to do is add the Warden middleware to our engine, which we can do by putting these lines inside `lib/subscribem/engine.rb`. While we're doing that, we'll also need to set up some session serialization rules. Let's do all of this now:

```
initializer "subscribem.middleware.warden" do
  Rails.application.config.middleware.use Warden::Manager do |manager|
    manager.serialize_into_session do |user|
      user.id
    end
    manager.serialize_from_session do |id|
      Subscribem::User.find(id)
    end
  end
end
```

This will insert the `Warden::Manager` middleware into the application's middleware stack. This middleware adds a key called `warden` to the request's environment object, which already contains things such as HTTP headers. By doing so, what's known as a "warden proxy object" – actually an instance of the `Warden::Proxy` class – will be available within our controller as `request.env["warden"]`, or through it's shorter variant `env["warden"]`. We can then use this object to manage the session for the current user.

The rules for serializing into and from the session are very simple. We only want to store the bare minimum amount of data in the session, since it is capped at about 4kb of data. Storing an entire User object in there is not good, as that will take up the entire session! The minimal amount of information we can store is the user's ID, and so that's how we've defined `serialize_into_session`. For `serialize_from_session`, we query the `Subscribem::User` table to find the user with that ID that was stored in the session by `serialize_into_session`.

To automatically sign in as a user for a particular account by using this new Warden proxy object, we can modify the `create` action inside `Subscribem::AccountsController` to be this:

```
def create
  account = Subscribem::Account.create(account_params)
  env["warden"].set_user(account.owner, :scope => :user)
  env["warden"].set_user(account, :scope => :account)
  flash[:success] = "Your account has been successfully created."
  redirect_to subscribem.root_url
end
```

This is the first place we are using the Warden proxy object. The `set_user` method tells Warden that we want to set the current session's user to that particular value. With the information Warden will store the ID in session and we can retrieve it at a later time, using the `user` method, like this:

```
Subscribem::Account.find(env["warden"].user(:scope => :account))
Subscribem::User.find(env["warden"].user(:scope => :user))
```

We can make it easier to access these values by defining two helper methods called `current_user` and `current_account` within `Subscribem::ApplicationController`. These methods should only return an object if a user is signed in, so we'll add a third method called `user_signed_in?` to determine that too. Let's add these methods now:

app/controllers/subscribem/application_controller.rb

```
1  def current_account
2    if user_signed_in?
3      @current_account ||= env["warden"].user(:scope => :account)
4    end
5  end
6  helper_method :current_account
7  def current_user
8    if user_signed_in?
9      @current_user ||= env["warden"].user(:scope => :user)
10   end
11 end
12 helper_method :current_user
13 def user_signed_in?
14   env["warden"].authenticated?(:user)
15 end
16 helper_method :user_signed_in?
```

Here we can see the `user` and `authenticated?` methods from warden's proxy object used. The `authenticated?` method takes the name of a scope and will return `true` or `false` if that scope has any data associated with it. If there is no data, then the user has not been authenticated.

Calling `helper_method` will make these methods available in the views as well, which means we can then use the methods to display if the user is signed in or not within the engine's layout:

app/views/layouts/subscribem/application.html.erb

```
<% if user_signed_in? %>
  Signed in as <%= current_user.email %>
<% end %>
```

If the user is set on the Warden proxy object, then the `user_signed_in?` method will return `true`, and the test will finally see the message it has been searching for. The only case in our engine where it would be set correctly is the `create` action within `Subscribem::AccountsController`. If running the test (`rspec spec/features/accounts/sign_up_spec.rb`) works, then we know that the authentication we have set up with Warden is working.

Let's run that command and find out.

```
Failure/Error:
  expect(page).to have_content("Signed in as subscribem@example.com")
     expected to find text "Signed in as subscribem@example.com" in
       ...
```

Not quite yet. It would appear that we're not being signed in correctly with this line from the controller:

```
env["warden"].set_user(account.owner, :scope => :user)
```

We should check to see what `account.owner` is and ensure that it's actually a user object. We can do this with a simple `p` call above that line:

```
p account.owner
```

When we run our test again, we'll see `nil` output at the very top of the test run. This indicates to us that `account.owner` is `nil`. The `owner` is not being set for the `account`, even though we're passing that in through the form. This is happening because we are not permitting any owner attributes to come through in the `account_params` method at the bottom of this controller:

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

We need to enable this method to accept parameters for `owner_attributes` too. We can do that with:

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

Making the method now permit the `owner_attributes` key to come through with `params[:account]` with its nested keys, will mean that the account's owner will be created correctly. When we run the test again, we'll see that our `p account.owner` call is now returning an object:

```
#<Subscribem::User id: 1, ...>
```

And, as an added bonus, our test is also now passing:

```
1 examples, 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 and that account through Warden.

Before we move on, let's remove the `p account.owner` from `Subscribem::AccountsController`.

This would be another great time to commit the code that we're working on.

1.5 Adding subdomain support

Using this engine, we're going to be restricting access to blogs to just the particular posts for particular accounts, with the use of subdomains. This is actually coming in Chapter 4, but really deserves a mention now to indicate the direction in which we're heading. Don't want any surprises now!

In order to separate the accounts in the engine, we're going to be using subdomains. When a user visits, for example, account1.example.com, they should be able to sign in for the account matching that subdomain and see that account's posts. If they also are a member of account2.example.com, they'll also be able to authenticate there too and see its posts.

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 within the engine, 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

```
7  fill_in "Name", :with => "Test"
8  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/subscribem/")
```

The path will still contain the "/subscribem" part because of how the engine is mounted inside the dummy application (`spec/dummy/config/routes.rb`):

```
mount Subscribem::Engine, :at => "subscribem"
```

This isn't a problem now, so we will leave this as it is.

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/subscribem/accounts/new.html.erb

```
<p>
  <%= account.label :subdomain %><br>
  <%= account.text_field :subdomain %>
</p>
```

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

```
rails g migration add_subdomain_to_subscribem_accounts subdomain:string
```

We're going to be doing lookups on this field to determine what the active `account` object should be, therefore we should also add an index for this field inside the migration. Adding an index will greatly speed up database queries if we were to have a large number of accounts in the system. Let's open it up now and change its contents to this:

db/migrate/[timestamp]_add_subdomain_to_subscribem_accounts.rb

```
class AddSubdomainToSubscribemAccounts < ActiveRecord::Migration
  def change
    add_column :subscribem_accounts, :subdomain, :string
    add_index :subscribem_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:

```
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/subscribem/")
  expected: "http://test.example.com/subscribem/"
       got: "http://www.example.com/subscribem/" (using ==)
```

In order to fix this, we need to tell the `AccountsController` to redirect to the correct place. Change this line within `app/controllers/subscribem/accounts_controller.rb`, from this:

```
redirect_to subscribem.root_url
```

To this:

```
redirect_to subscribem.root_url(:subdomain => account.subdomain)
```

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

```
Failure/Error:
  page.should have_content("Your account has been successfully created.")
  expected there to be content
    "Your account has been successfully created."
  in ...
```

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 dummy application. To do this, open `spec/dummy/config/initializers/session_store.rb` and change this line:

```
Rails.application.config.session_store :cookie_store,
  key: "_dummy_session"
```

To this:

```
Rails.application.config.session_store :cookie_store,
  key: "_dummy_session",
  domain: "example.com"
```

This will store all session information within the dummy app under the `example.com` domain, meaning that our subdomain sessions will be the same as the root domain sessions.

This little change means that running `rspec spec/features/accounts/sign_up_spec.rb` again will result in the test passing:

```
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.

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 `Subscribem::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 the test inside `spec/features/accounts/sign_up_spec.rb`:

spec/features/accounts/sign_up_spec.rb

```
1  scenario "Ensure subdomain uniqueness" do
2    Subscribem::Account.create!(:subdomain => "test", :name => "Test")
3    visit subscribem.root_path
4    click_link "Account Sign Up"
5    fill_in "Name", :with => "Test"
6    fill_in "Subdomain", :with => "test"
7    fill_in "Email", :with => "subscribem@example.com"
8    fill_in "Password", :with => "password"
9    fill_in "Password confirmation", :with => 'password'
10   click_button "Create Account"
11   expect(page.current_url).to eq("http://www.example.com/subscribem/accounts")
12   expect(page).to have_content("Sorry, your account could not be created.")
13   expect(page).to have_content("Subdomain has already been taken")
14 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:19` will result in it failing like this:

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

This indicates that the account sign up functionality is working, and perhaps too well. Let's fix that up now by first re-defining the `create` action within `Subscribem::AccountsController` like this:

```
def create
  @account = Subscribem::Account.new(account_params)
  if @account.save
    env["warden"].set_user(@account.owner, :scope => :user)
    env["warden"].set_user(@account, :scope => :account)
    flash[:success] = "Your account has been successfully created."
    redirect_to subscribem.root_url(:subdomain => @account.subdomain)
  else
    flash[:error] = "Sorry, your account could not be created."
    render :new
  end
end
```

Rather than calling `Subscribem::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's rendered again. We then call `save` to return `true` or `false` depending on if the validations for that object pass or fail. 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've also switched from using a local variable for the account object to using an instance variable. This is so that when the `new` action's template is rendered again, it will have access to this object.

To make this error pop up, we're going to need to add a validation to the `Subscribem::Account` model for subdomains. A good place for this is right at the top of the model:

```
class Account < ActiveRecord::Base
  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.

Now 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:

```
expected there to be content "Subdomain has already been taken." in ...
```

This is the final line of our test that's failing now. This error is happening because we're not displaying any validation messages inside our form. To fix this, we'll use the `dynamic_form` gem, which allows us to call `error_messages` on the form builder object to get neat error messages. We'll add this gem to `subscribem.gemspec` now:

```
s.add_dependency "dynamic_form", "1.1.4"
```

We'll need to install this gem if we haven't already. Running this familiar command will do that:

```
bundle install
```

Then we can require it within `lib/subscribem/engine.rb` to load it:

```
require "dynamic_form"
```

To use it, we'll change the start of the form definition inside `app/views/subscribem/accounts/new.html.erb` to this:

```
<%= form_for(@account) do |account| %>
  <%= account.error_messages %>
```

Running the test once more will result in its passing:

```
1 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, in Chapter 3.

While we're in this frame of mind, we should also restrict the names of subdomains that a user can have. This will allow us to prevent abuse of the subdomains, which could happen if the user called their subdomain "admin" or something more nefarious.

Restricting subdomain names

When we restrict subdomain names, we should restrict them to just letters, numbers, underscores and dashes. As well as this, we should not allow certain words like "admin" or swear words.

Let's write a new test for this in `spec/features/accounts/sign_up_spec.rb`:

```
scenario "Subdomain with restricted name" do
  visit subscribem.root_path
  click_link "Account Sign Up"
  fill_in "Name", :with => "Test"
  fill_in "Subdomain", :with => "admin"
  fill_in "Email", :with => "subscribem@example.com"
  fill_in "Password", :with => "password"
  fill_in "Password confirmation", :with => "password"
  click_button "Create Account"
  expect(page.current_url).to eq("http://www.example.com/subscribem/accounts")
  expect(page).to have_content("Sorry, your account could not be created.")
  expect(page).to have_content("Subdomain is not allowed. Please choose another subdomain.")
end
```

If we were to run this test now, we would see that it's failing:

```
Failure/Error:
  expect(page.current_url).to eq("http://www.example.com/subscribem/accounts")
  expected: "http://www.example.com/subscribem/accounts"
       got: "http://admin.example.com/" (using ==)
```

This is happening because we've not put any restriction in place yet. We can enforce this restriction by placing the following code in the `Account` model:

```
EXCLUDED_SUBDOMAINS = %w(admin)
validates_exclusion_of :subdomain, :in => EXCLUDED_SUBDOMAINS,
  :message => "is not allowed. Please choose another subdomain."
```

This code is a new validation for the `subdomain` field which rejects any account that has a subdomain that is within the list of excluded subdomains. This validation will return the message of "Subdomain is not allowed. Please choose another subdomain." that our test needs.

Is this code enough to get our test to pass? Let's run it and find out:

```
1 example, 0 failures
```

Yup, it sure is. But there's one problem with this validation: *it isn't case-sensitive*. This means if someone entered "Admin" or "ADMIN" it would break. We can see this ourselves if we change the subdomain field in our test and re-run it:

```
Failure/Error: expect(page.current_url).to eq("http://www.example.com/subscribem/accounts")
  expected: "http://www.example.com/subscribem/accounts"
       got: "http://ADMIN.example.com/subscribem" (using ==)
```

This is bad. What we'll do to fix this problem is relatively easy: we'll just convert the subdomain to lowercase before validating it. This can be accomplished with this callback in the model:

```
before_validation do
  self.subdomain = subdomain.to_s.downcase
end
```

Running the test again will once again show that it's passing:

```
1 example, 0 failures
```

Good-o. One more thing to do: we should only allow subdomains with letters, numbers, underscores and dashes in their name. No funky characters, please! Another test in `spec/features/accounts/sign_up_spec.rb` will help us enforce this:

```
scenario "Subdomain with invalid name" do
  visit subscribem.root_path
  click_link "Account Sign Up"
  fill_in "Name", :with => "Test"
  fill_in "Subdomain", :with => "<admin>"
  fill_in "Email", :with => "subscribem@example.com"
  fill_in "Password", :with => "password"
  fill_in "Password confirmation", :with => "password"
  click_button "Create Account"
  expect(page.current_url).to eq("http://www.example.com/subscribem/accounts")
  expect(page).to have_content("Sorry, your account could not be created.")
  expect(page).to have_content("Subdomain is not allowed. Please choose another subdomain.")
end
```

When we run this test, we'll see why it's a bad idea to have angle-brackets in a name:

```
Failure/Error: click_button "Create Account"
URI::InvalidURIError:
  bad URI(is not URI?): http://<admin>.example.com/
```

Ruby's URI library doesn't seem to like this at all, and that means browsers probably won't either! Therefore, we'll add a restriction to only allow letters, numbers, underscores and dashes in the subdomain. The way we do that is with another validation in the `Subscribem::Account` model:

```
validates_format_of :subdomain, :with => /\A[\w\-]+\Z/i,
                    :message => "is not allowed. Please choose another subdomain."
```

This little regular expression here ensures that we have any word character with `\w` (meaning a letter, number or underscore), or a dash. The `+` on the end means "any number of these in a row". The `\A` and `\Z` at the beginning and end of the regular expression means that we want to make sure that the entire subdomain only consists of these characters.

Running our test again will show it passing:

```
1 example, 0 failures
```

Good stuff! Now we're only allowing subdomains that are valid according to our constraints. This will stop people from potentially abusing the system in ways that only a madman could imagine.

You may be wondering why at this point we've written feature specs for this, when unit tests would probably do the same job. It comes down to a matter of personal choice, really. I much prefer validating that the flow that the user goes through is operating correctly, and I think that a unit test doesn't accomplish that. How can you be sure with a unit test that the user is *actually* seeing the validation message? You can't!

Therefore a feature spec is better to use *in this case* because we really want to make sure the user isn't just being flat-out denied the ability to create a new account without a good reason!

Let's now move on to authenticating users on a per-subdomain basis.

1.6 Building subdomain authentication

When a user creates an account, they should be able to sign in after they come back to the account's subdomain. To allow this, we'll add a sign in page for subdomains that allow users for that account to sign in. When visiting an account's subdomain without a currently active session, users should also be prompted to sign in.

Testing subdomain authentication

Let's write a test for this now within a new file:

spec/features/users/sign_in_spec.rb

```
1  require "rails_helper"
2  feature "User sign in" do
3    let!(:account) { FactoryGirl.create(:account) }
4    let(:sign_in_url) { "http://#{account.subdomain}.example.com/sign_in" }
5    let(:root_url) { "http://#{account.subdomain}.example.com/" }
6    within_account_subdomain do
7      scenario "signs in as an account owner successfully" do
8        visit root_url
9        expect(page.current_url).to eq(sign_in_url)
10       fill_in "Email", :with => account.owner.email
11       fill_in "Password", :with => "password"
12       click_button "Sign in"
13       expect(page).to have_content("You are now signed in.")
14       expect(page.current_url).to eq(root_url)
15     end
16   end
17 end
```

In this test, we'll be using Factory Girl to easily set up an account with an owner, which we'll use inside our test. For instance, we use this object to define a couple of important URLs that we'll need later on in our test. For the test itself, we'll have a `within_account_subdomain` method which will correctly scope Capybara's requests to being within the subdomain. This method doesn't exist yet, and so we'll need to create it in a short while.

In the actual example itself, we visit the root path of the account's subdomain which should then redirect us to the `/sign_in` path. This page will present us with an "Email" and "Password" field which when filled out correctly, will allow the user to authenticate for the account.

Let's run this spec now with `rspec spec/features/users/sign_in_spec.rb` and we'll see that there is definitely no method called `within_account_subdomain`:

```
undefined method `within_account_subdomain' for ...
```

This method should be provided by a helper module within the spec files for this test. It will be responsible for altering Capybara's `default_host` variable so that requests are scoped within a subdomain for the duration of the block, and then it'll reset it after its done.

To do this, let's create a new file called `spec/support/subdomain_helpers.rb` and put this content in it:

spec/support/subdomain_helpers.rb

```
1  module SubdomainHelpers
2    def within_account_subdomain
3      let(:subdomain_url) { "http://#{account.subdomain}.example.com" }
4      before { Capybara.default_host = subdomain_url }
5      after { Capybara.default_host = "http://www.example.com" }
6      yield
7    end
8  end
```

We're going to be using the `within_account_subdomain` method as kind of a "super-scenario" block. We even call the `scenario` method inside the method! What this will do is scope the tests inside the `within_account_subdomain` block to be within a context that sets Capybara's `default_host` to contain a subdomain, runs the tests, and then resets it to the default. What this will allow is a very easy way of testing subdomain features within our Capybara request specs.

To use this module, we can put this line inside the `describe` block for `spec/features/users/sign_in_spec.rb`, like this:

```
feature "User sign in" do
  extend SubdomainHelpers
```

We're doing it this way so that the `SubdomainHelpers` code gets included into just this one test where we need it.

The next run of `rspec spec/features/users/sign_in_spec.rb` will tell us that `FactoryGirl` is an undefined constant:

```
Failure/Error: let!(:account) { FactoryGirl.create(:account) }
NameError:
  uninitialized constant FactoryGirl
```

Using Factory Girl

To fix this problem, we will need to add `factory_girl` to the `subscribem.gemspec` file as a development dependency:

```
s.add_development_dependency "factory_girl", "4.4.0"
```

We can install this gem by running the familiar command:

```
bundle install
```

We will also need to require this gem inside our `rails_helper.rb` file, which we can do right after RSpec's and Capybara's requires:

```
require "rspec/rails"
require "capybara/rspec"
require "factory_girl"
```

These two things will make the uninitialized constant go away. Running the test again will result in the factory not being found:

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

To fix this error, we will need to define an account factory inside `spec/support/factories/account_factory.rb`, like this:

spec/support/factories/account_factory.rb

```
1  FactoryGirl.define do
2    factory :account, :class => Subscribem::Account do
3      sequence(:name) { |n| "Test Account ##{n}" }
4      sequence(:subdomain) { |n| "test#{n}" }
5      association :owner, :factory => :user
6    end
7  end
```

Within this factory definition, we need to use the `:class` option so that Factory Girl knows what class to use when building objects. If we didn't specify this, it would attempt to build them with the `Account` class, which doesn't exist.

Inside the factory, we use the `sequence` method which will generate a unique name for every single account. We also use the `association` method to create a new object for the `owner` asocation using the `user` factory.

This factory won't work right now because we don't have this user factory, so let's define this now in a new file called `spec/support/factories/user_factory.rb`.

spec/support/factories/user_factory.rb

```
FactoryGirl.define do
  factory :user, :class => Subscribem::User do
    sequence(:email) { |n| "test#{n}@example.com" }
    password "password"
    password_confirmation "password"
  end
end
```

Inside this new factory we use the `class` option and `sequence` methods again. A user needs to be created with just an email and a password, and that's exactly what this factory does. This user has the password of "password" just so its an easy value for when we need to use it in our tests.

When we run our tests again, we'll get this error:

```
Failure/Error: visit root_url
ActionController::RoutingError:
  No route matches [GET] "/"
```

This issue is happening because we're trying to route to somewhere that has no route attached to it.

To fix this, we'll change the engine's mounting point from `/subscribem` to `/` by changing this line inside `spec/dummy/config/routes.rb`:

```
mount Subscribem::Engine => "/subscribem"
```

To this:

```
mount Subscribem::Engine => "/"
```

This way, we won't have that pesky `/subscribem` part crowding up our URL any more.

This change will cause the `spec/features/account/sign_up_spec.rb` to break if we run it:

```
1) Accounts creating an account
   Failure/Error: expect(page.current_url).to
     eq("http://test.example.com/subscribem/")

     expected: "http://test.example.com/subscribem/"
          got: "http://test.example.com/" (using ==)
```

Let's correct that URL now by changing the line in the test from this:

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

To this:

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

We'll also need to make similar changes to the rest of the tests in this file, otherwise they will fail in a very similar way. After we've made those changes, running that file again will make all the tests inside it pass once again.

Let's re-run our user sign in spec using `rspec spec/features/users/sign_in_spec.rb` and see what that's doing now:

```
Failure/Error: expect(page.current_url).to eq(sign_in_url)
  expected: "http://test1.example.com/sign_in"
       got: "http://test1.example.com/" (using ==)
```

This test isn't yet redirecting people to the sign in URL when they visit the root path on a subdomain. To make this happen, we'll need to define a new root route for accounts in the engine and handle that redirect in a new controller. We can do this by defining what's known as a *subdomain constraint*.

A subdomain constraint requires that certain routes be accessed through a subdomain of the application, rather than at the root. If no subdomain is provided, then the routes will be ignored and other routes may match this path.

Let's see about adding this new subdomain constraint now.

Adding a subdomain constraint

The subdomain constraint for the engine's routes will constrain some routes of the engine to requiring a subdomain. These routes will route to different controllers within the application from their non-subdomain compatriots.

For these constrained routes, we'll be routing to controllers within another namespace of our engine just to keep it separate from the 'top-level' controllers which will be responsible for general account actions.

Constraints within Rails routes can be defined as simply as this:

```
constraints(:ip => /192.168.\d+.\d+/) do
  resources :posts
end
```

Such a constraint would make this route only accessible to computers with an IP address beginning with `192.168.` and ending with any number of digits. Our subdomain constraint is going to be a little more complex, and so we'll be using a class for this constraint.

Let's begin defining this constraint at the top of the `config/routes.rb` file (before the `root` route) within the engine like this:

config/routes.rb

```
constraints(Subscribem::Constraints::SubdomainRequired) do
end
```

When you pass a class to the `constraints` method, that class must respond to a `matches?` method which returns `true` or `false` depending on if the request matches the criteria required for the constraint. In this instance, we want to constrain the routes inside the block to only requests that include a subdomain, and that subdomain *can't* be www.

Let's define this constraint now in a new file located at `lib/subscribem/constraints/subdomain_required.rb`:

lib/subscribem/constraints/subdomain_required.rb

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

This subdomain constraint checks the incoming request object and sees if it contains a subdomain that's not `www`. If those criteria are met, then it will return `true`. If not, `false`.

The constraint request object

The `request` object passed in to the `matches?` call in any constraint is an `ActionDispatch::Request` object, the same kind of object that is available within your application's (and engine's) controllers.

You can get other information out of this object as well, if you wish. For instance, you could access the Warden proxy object with an easy call to `request.env["warden"]` (or `env["warden"]` if you like) which you could then use to only allow routes for an authenticated user.

The next thing we'll need to set up in the `config/routes.rb` file is some actual routes for this constraint, which we can do by making the `constraints` block this:

```
constraints(Subscribem::Constraints::SubdomainRequired) do
  scope :module => "account" do
    root :to => "dashboard#index", :as => :account_root
  end
end
```

Inside the `constraints` block, we're using the `scope` method which will scope routes by some given options. In this instance, we're scoping by a module called "account", meaning controllers for the routes underneath this scope will be within the `Subscribem::Account` namespace, rather than just the `Subscribem` namespace. This routing code that we've just added is a shorthand for this:

```
constraints(Subscribem::Constraints::SubdomainRequired) do
  root :to => "account/dashboard#index", :as => :account_root
end
```

Rather than repeating ourselves many times by specifying the "account" module on each line inside this constraint, we use the `scope` block. Even though we've only got one line in there so far, it's just good practice to do this.

Inside the `scope` block we define one and only one route: the `root` route. This is what the test inside `spec/features/users/sign_in_spec.rb` is relying on to visit and then be redirected to the sign in path, because the user isn't authenticated.

The reason why the `constraints` call is at the top of the `config/routes.rb` file is because we want these subdomain-specific routes to be matched first before the non-subdomain routes are. If a user makes a request to test.oursite.com, we're going to be showing them the `Accounts::DashboardController#index` action, rather than the non-namespaced `DashboardController#index`. This is because `Accounts::DashboardController#index` would contain information relevant to that subdomain, whereas `DashboardController#index` wouldn't.

To make the `Subscribem::Constraints::SubdomainRequired` constraint available within the `config/routes.rb` file, we'll need to require the file where it's defined. Put this at the top of `config/routes.rb` now:

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

When we run `rspec spec/features/users/sign_in_spec.rb` we'll see that the controller for this new route is missing:

```
Failure/Error: visit root_url
ActionController::RoutingError:
  uninitialized constant Subscribem::Account::DashboardController
```

This controller will, one day, provide a user with a dashboard for their account. Right now, we'll just use it as a 'dummy' controller for this test.

Let's generate this controller now by running this command:

```
rails g controller account/dashboard
```

This controller should not allow any requests to it from users that aren't authenticated. To stop and redirect these requests, we'll add another helper method to `Subscribem::ApplicationController` called `authenticate_user!`:

```
def authenticate_user!
  unless user_signed_in?
    flash[:notice] = "Please sign in."
    redirect_to "/sign_in"
  end
end
```

We'll then use this method as a `before_filter` within this new controller located at app/controllers/subscribem/account/dashboard_controller.rb.

```
module Subscribem
  class Account::DashboardController < Subscribem::ApplicationController
    before_filter :authenticate_user!
  end
end
```

We will also need to add an index action to this controller, because otherwise the test will complain like this when it's run again:

```
Failure/Error: visit subscribem.root_url(:subdomain => account.subdomain)
AbstractController::ActionNotFound:
  The action 'index' could not be found
  for Subscribem::Account::DashboardController
```

Defining a template for this action will be enough, so do that now:

app/views/subscribem/account/dashboard/index.html.erb

```
1 Your account's dashboard. Coming soon.
```

Let's make sure that this `before_filter` is working correctly by re-running our test with `rspec spec/features/users/sign_in_spec.rb`.

```
Failure/Error: visit subscribem.root_url(:subdomain => account.subdomain)
ActionController::RoutingError:
  No route matches [GET] "/sign_in"
```

This test is now failing because there's no route for `/sign_in`. This route should go to an action within a new controller. That action will render the sign in form for this account and will be used to sign in users that belong to this account.

Creating a sign in page

The first step in order to get this test closer to passing is to define a route inside the subdomain constraint within `config/routes.rb`, which we can do with this code:

```
constraints(Subscribem::Constraints::SubdomainRequired) do
  scope :module => "account" do
    root :to => "dashboard#index", :as => :account_root
    get "/sign_in", :to => "sessions#new", :as => :sign_in
  end
end
```

This route will need a controller and action to go along with it, so we'll generate the controller using this command:

```
rails g controller account/sessions
```

To define the action in the controller and set it up for user authentication, we'll change the entire controller's file to read like this:

app/controllers/subscribem/account/sessions_controller.rb

```
1  module Subscribem
2    class Account::SessionsController < Subscribem::ApplicationController
3      def new
4        @user = User.new
5      end
6    end
7  end
```

The form for this action is going to need an email and password field. Let's create the template for this action now:

app/views/subscribem/account/sessions/new.html.erb

```
1  <h2>Sign in</h2>
2  <%= form_for @user, :url => sessions_url do |f| %>
3    <p>
4      <%= f.label :email %><br>
5      <%= f.email_field :email %>
6    </p>
7    <p>
8      <%= f.label :password %><br>
9      <%= f.password_field :password %>
10   </p>
11   <p>
12     <%= f.submit "Sign in" %>
13   </p>
14 <% end %>
```

This is a pretty standard form, nothing magical to see here. The `sessions_url` helper used within it isn't currently defined, and so we will need to add a route to the `config/routes.rb` file to define the helper and its route:

```
constraints(Subscribem::Constraints::SubdomainRequired) do
  scope :module => "account" do
    root :to => "dashboard#index", :as => :account_root
    get "/sign_in", :to => "sessions#new"
    post "/sign_in", :to => "sessions#create", :as => :sessions
  end
end
```

The `create` action with `Subscribem::Account::SessionsController` will need to take the parameters from the form and attempt to sign in a user. The catch for this is that the user lookup for the authentication must be scoped by the current subdomain's account.

Authenticating by subdomain

To do the authentication through a subdomain, we're going to use a Warden strategy. When we ask warden to authenticate a request, it will go through all the strategies it knows of to authenticate it, and if one strategy works for a given request, then that session will be authenticated.

As per the Warden Wiki[3], our strategy needs an `authenticate!` method which does the bulk of the heavy lifting. We're also going to add a `valid?` method which will make this particular strategy valid only when there is a subdomain in the request and there are parameters coming through for a user.

We can define this strategy in another file (just to keep the engine class clean!), by placing it inside `config/initializers/warden/strategies/password.rb`.

config/initializers/warden/strategies/password.rb

```
1  Warden::Strategies.add(:password) do
2    def valid?
3      host = request.host
4      subdomain = ActionDispatch::Http::URL.extract_subdomains(host, 1)
5      subdomain.present? && params["user"]
6    end
7    def authenticate!
8      if u = Subscribem::User.find_by(email: params["user"]["email"])
9        u.authenticate(params["user"]["password"]) ? success!(u) : fail!
10     else
11       fail!
12     end
13   end
14 end
```

For our strategy, it will only be valid if a subdomain is present as well as the `params["user"]` hash. The `request` object in this instance is actually a `Rack::Request` object, rather than an `ActionDispatch::Request` object. This is because the strategy is processed before the request gets to the Rails stack. Therefore we need to use the `ActionDispatch::Http::URL.extract_subdomains` method to get the subdomain for this request. We also need to use strings for the `params` keys for the same reasons.

In the `authenticate` method, we attempt to find a user by their email address. If there isn't one, then the authentication fails. After that, we attempt to authenticate the user with their password. If their password is

[3] https://github.com/hassox/warden/wiki/Strategies

wrong, we'll return `fail!`. If we find a valid user and they have a valid password then we'll use `success!`, passing through the `User` object.

We can tell Warden to use this strategy for authentication by calling `default_strategies` within our Warden configuration in `lib/subscribem/engine.rb`:

```
Rails.application.config.middleware.use Warden::Manager do |manager|
  manager.default_strategies :password
```

We can use this strategy inside the `SessionsController` create action by calling the `authenticate` method on the `Warden::Proxy`, like this:

```
def create
  if env["warden"].authenticate(:scope => :user)
    flash[:notice] = "You are now signed in."
    redirect_to root_path
  end
end
```

The `authenticate` method will use the password strategy we have defined in order to authenticate this request. If it's successful, it will set the flash notice to something indicating that to the user and redirect them to the root path for this subdomain. These are all things that our test wants to happen.

Let's see if this works by running the spec again with `rspec spec/features/users/sign_in_spec.rb`:

```
1 examples, 0 failures
```

Awesome! We're now authenticating users for their own subdomain. Our next step is to provide useful information back to them if they're unsuccessful in their authentication attempts.

Invalid usernames or passwords

If a user enters an invalid email or password, they shouldn't be able to authenticate to an account. Let's cover this in another two tests inside `spec/features/users/sign_in_spec.rb` now:

```
scenario "attempts sign in with an invalid password and fails" do
  visit subscribem.root_url(:subdomain => account.subdomain)
  expect(page.current_url).to eq(sign_in_url)
  expect(page).to have_content("Please sign in.")
  fill_in "Email", :with => account.owner.email
  fill_in "Password", :with => "drowssap"
  click_button "Sign in"
  expect(page).to have_content("Invalid email or password.")
  expect(page.current_url).to eq(sign_in_url)
end
scenario "attempts sign in with an invalid email address and fails" do
  visit subscribem.root_url(:subdomain => account.subdomain)
  expect(page.current_url).to eq(sign_in_url)
  expect(page).to have_content("Please sign in.")
```

```
  fill_in "Email", :with => "foo@example.com"
  fill_in "Password", :with => "password"
  click_button "Sign in"
  expect(page).to have_content("Invalid email or password.")
  expect(page.current_url).to eq(sign_in_url)
end
```

In this spec, we've got the user signing in as the account owner with an invalid password. When that happens, they should be told that they've provided an invalid email or password.

When we run this whole file with `rspec spec/features/users/sign_in_spec.rb`, we'll see that the test is failing, as we would hope:

```
Failure/Error: click_button "Sign in"
  ActionView::MissingTemplate:
  Missing template subscribem/account/sessions/create, ...
```

The actions of this test mean that the user isn't authenticated correctly, which means the code inside the `if` inside the `create` action for `Subscribem::Account::SessionsController` is not going to execute, meaning the implicit render of the action's template will take place.

To fix this, we will need to add an `else` to this `if` statement:

app/controllers/subscribem/account/sessions_controller.rb

```
1  def create
2    if env["warden"].authenticate(:scope => :user)
3      flash[:success] = "You are now signed in."
4      redirect_to root_path
5    else
6      @user = User.new
7      flash[:error] = "Invalid email or password."
8      render :action => "new"
9    end
10 end
```

Did this fix these two tests? Find out with another run of `rspec spec/features/users/sign_in_spec.rb`.

```
3 examples, 0 failures
```

We're now testing that a user can sign in successfully and that a user is barred from signing in when they provide an invalid password or an invalid email address. The validility of that is determined by the `authenticate` method on objects of the `Subscribem::User` class, which is provided by the `has_secure_password` call in the class itself.

One thing we're not testing at the moment is that users should only be able to sign in for accounts that they have access to. Let's add a test for this final case now.

Restricting sign in by account

When a new user creates an account, they should only be able to sign in as that account. Currently, within the password strategy we have defined for Warden we're *not* restricting the login by an account, which is wrong. This should definitely be restricted to only users from an account.

In order to do this, we'll create a new table which will link accounts and users. We'll then use this table to scope the query for finding users inside the password strategy.

First, we're going to need a test to make sure this works as intended. Let's add another test right underneath the invalid email test inside `spec/features/users/sign_in_spec.rb`:

spec/features/users/sign_in_spec.rb

```
scenario "cannot sign in if not a part of this subdomain" do
  other_account = FactoryGirl.create(:account)
  visit subscribem.root_url(:subdomain => account.subdomain)
  expect(page.current_url).to eq(sign_in_url)
  expect(page).to have_content("Please sign in.")
  fill_in "Email", :with => other_account.owner.email
  fill_in "Password", :with => "password"
  click_button "Sign in"
  expect(page).to have_content("Invalid email or password.")
  expect(page.current_url).to eq(sign_in_url)
end
```

In this test, we visit the first account's subdomain and attempt to sign in as another account's owner. The sign in process should fail, producing the "Invalid email or password." error.

If we run this test with `rspec spec/features/users/sign_in_spec.rb`, it will fail like this:

```
Failure/Error: page.should have_content("Invalid email or password.")
expected there to be content "Invalid email or password." in
"You are now signed in..."
```

Our password strategy is still letting the user in, although they're not a member of this account. Let's fix up this password strategy now by altering the entire strategy to this:

config/initializers/warden/strategies/password.rb

```
Warden::Strategies.add(:password) do
  def subdomain
    ActionDispatch::Http::URL.extract_subdomains(request.host, 1)
  end
  def valid?
    subdomain.present? && params["user"]
  end
  def authenticate!
    return fail! unless account = Subscribem::Account.find_by(subdomain: subdomain)
    return fail! unless user = account.users.find_by(email: params["user"]["email"])
    return fail! unless user.authenticate(params["user"]["password"])
    success! user
  end
end
```

The first thing to notice in this strategy is that we've moved the subdomain code out from the `valid?` method and into its own method. Not only is this neater, but it also allows us to access this subdomain within both methods of our strategy.

In the new `authenticate!` method, we attempt to find an account by the subdomain. If there is no account by that subdomain, then the authentication fails. Next, we attempt to find a user for that account by the email passed in. If there is no user, then the authentication fails too. We then attempt to authenticate that user and fail again if their password is incorrect. If we are able to find the account *and* the user *and* their password is valid, then the authentication is a success.

The `users` method on the `Account` object that we use within `authenticate!` isn't defined yet. To define this method, we'll need to define an association on the `Account` class to the users, which we could do with a `has_and_belongs_to_many`, but that's rather inflexible if we ever want to add another field to this join table. Let's do it using a `has_many :through` association instead with these lines inside `app/models/subscribem/account.rb`:

```
has_many :members, :class_name => "Subscribem::Member"
has_many :users, :through => :members
```

We need to specify the `class_name` option here because Rails would assume that the class name is called `Member`, and not `Subscribem::Member`.

We need to create the intermediary model now, which we can do by running this command in the console:

```
rails g model member account_id:integer user_id:integer
```

Inside this model's file (`app/models/subscribem/member.rb`), we're going to need to define the `belongs_to` associations for both `account` and `user` so that both models can find each other through this model:

```
module Subscribem
  class Member < ActiveRecord::Base
    belongs_to :account, :class_name => "Subscribem::Account"
    belongs_to :user, :class_name => "Subscribem::User"
  end
end
```

Now that we've got this intermediary model set up, let's run the migrate command to add this table to our database.

```
rake db:migrate
```

Now that we have the table, all that's left to do is to associate an account owner with an account, as a user, after the account has been created. Typically, we would use an `after_create` callback within the `Subscribem::Account` model to do this, but this irrevocably ties us to always needing an owner when we create an account. It would be *much* better if this was done only when we needed it.

The first place we'll need to do it is in our account factory. We'll be using an `after_create` here, because we're using this factory within our Capybara-based specs, and we always need this association so that the user can sign in to an account.

In our account factory we can add the owner to the list of users by calling an `after_create` here: this method using an `after_create` callback:

```
factory :account, :class => Subscribem::Account do
  sequence(:name) { |n| "Test Account ##{n}" }
  sequence(:subdomain) { |n| "test#{n}" }
  association :owner, :factory => :user
  after(:create) do |account|
    account.users << account.owner
  end
end
```

When an account is created, the owner will automatically be added to the `users` list. This means that every account created using this factory will now at least have one user who can sign in to it.

We should also associate the owner with the account in the `create` action inside `Subscribem::AccountsController`, which is more of a model concern than a controller concern. Therefore, we should create a model method to create an account and associate its owner with its list of users.

> **Why not a callback?**
>
> We *could* have this owner association done with a callback, placing the concern of this association directly in the model, meaning it wouldn't need to be put into the factory or the controller. The problem with this is that every single account that we create in the system would then need to be tied to an owner, which is extra overhead that we don't need when we want to just work with a very basic `Account` object. By doing it this way, we then have two ways of creating an `Account` object: one with an owner, and one without.

Before we write that code, let's write a quick test for it within a new file at `spec/models/subscribem/account_spec.rb`:

```
require "rails_helper"
describe Subscribem::Account do
  it "can be created with an owner" do
    params = {
      :name => "Test Account",
      :subdomain => "test",
      :owner_attributes => {
        :email => "user@example.com",
        :password => "password",
        :password_confirmation => "password"
      }
    }
    account = Subscribem::Account.create_with_owner(params)
    expect(account.persisted?).to eq(true)
    expect(account.users.first).to eq(account.owner)
  end
end
```

We'll also write another test to ensure that if we didn't enter a subdomain that this method would return `false`:

```
it "cannot create an account without a subdomain" do
  account = Subscribem::Account.create_with_owner
  expect(account.valid?).to eq(false)
  expect(account.users.empty?).to eq(true)
end
```

The corresponding method simply needs to create an account and associate its owner with the list of users for that account, but only after ensuring that the account is valid. This can be done by defining the method inside `app/models/subscribem/account.rb` like this:

```
def self.create_with_owner(params={})
  account = new(params)
  if account.save
    account.users << account.owner
  end
  account
end
```

If the account is valid, then we go ahead and associate the owner with the list of users for that account. If not, then we return the account object as it stands. It's then the responsibility of whatever calls this `create_with_owner` method in our code to decide what to do with that object.

Do those tests work now? Find out with a quick run of `rspec spec/models/subscribem/account_spec.rb`:

```
2 examples, 0 failure
```

Indeed they does! Now let's use this new method within the `create` action of `Subscribem::AccountsController`:

```
def create
  @account = Subscribem::Account.create_with_owner(account_params)
  if @account.valid?
    env["warden"].set_user(@account.owner, :scope => :user)
    env["warden"].set_user(@account, :scope => :account)
    flash[:success] = "Your account has been successfully created."
    redirect_to subscribem.root_url(:subdomain => @account.subdomain)
  else
    flash[:error] = "Sorry, your account could not be created."
    render :new
  end
end
```

The `create_with_owner` call here will always return a `Subscribem::Account` object, and so it's the responsibility of the controller to check if that object is valid and then to decide what to do with it. If the object is valid, then we go through the authentication process, tell the user their account has been created and send them off to the root path for that account. If it's not valid, too bad so sad and we show them the form again.

That should be everything necessary to associate users with accounts, and therefore we're done setting up the code needed for the strategy to work correctly. Does it work? Find out with another run of `rspec spec/features/users/sign_in_spec.rb`.

```
4 examples, 0 failures
```

Yay, it's working! We've now finished setting up authentication for accounts within this engine. Users who belong to an account (just owners for now) are able to sign in again to that account. Users who do not belong to that account or those who provide an invalid email or password are not able to sign in.

We have sign in working, now how about making sign up work too?

1.7 Handling user signup

Right now, we have a way for one user in one account to be created. The way to do that is to sign up for a new account. Within that process, a new account record is created, as well as a user record. What we need on top of this is the ability to let other users sign up to these new accounts, so that they can access that account's resources along with the account owner.

In this section, we'll add a feature to allow a user to sign up. Once the user is signed up, they'll be able to see the same list of "things" that the account owner can.

We're not going to provide a link that a user can use to navigate to this action, because that will be the responsibilty of the application where this engine is mounted, not the engine itself. We'll create that link in Chapter 4, when we build an application to mount our engine into.

As per usual, we're going to write a test for the user signup before we write any implementation, just to make sure it's working. Let's write this simple test now within `spec/features/users/sign_up_spec.rb`:

```
require "rails_helper"
feature "User signup" do
  let!(:account) { FactoryGirl.create(:account) }
  let(:root_url) { "http://#{account.subdomain}.example.com/" }
  scenario "under an account" do
    visit root_url
    expect(page.current_url).to eq(root_url + "sign_in")
    click_link "New User?"
    fill_in "Email", :with => "user@example.com"
    fill_in "Password", :with => "password"
    fill_in "Password confirmation", :with => "password"
    click_button "Sign up"
    expect(page).to have_content("You have signed up successfully.")
    expect(page.current_url).to eq(root_url)
  end
end
```

Typical behaviour for our application when users visit the root of the domain is to redirect to the sign in page. This is what the first two lines of this example are testing; that the redirect happens. From this page, we want users to be able to sign up if they haven't already. To prompt them to do this, we will have a "New User?" link on the sign in page that then sends them to the sign up page where they can then fill in the new user form. Once they're done filling it in, they'll be signed in automatically and redirected to the root of the subdomain.

When we run this spec with `rspec spec/features/users/sign_up_spec.rb`, we'll see that it's failing on the line for the "New User?" link:

```
Failure/Error: click_link "New User?"
Capybara::ElementNotFound:
  Unable to find link "New User?"
```

We can add this link to `app/views/subscribem/account/sessions/new.html.erb` like this:

```
<%= link_to "New User?", user_sign_up_url %>
```

Easy. The next time we run this spec, we'll see this error:

```
Failure/Error: click_button "Sign up"
Capybara::ElementNotFound:
  no button with value or id or text 'Sign up' found
```

This is happening because when the test visits the `/sign_up` page, the request is going to the `Subscribem::AccountsController#new` action, because it has a route defined that matches this request:

```
get "/sign_up", :to => "accounts#new", :as => :sign_up
```

This route is used for when people want to create a new account, rather than for new users signing up to existing accounts. What we'll need is a route for user sign up within the context of an account, which we can do by placing a new route within the `SubdomainRequired` constraint of `config/routes.rb`, like this:

```
constraints(Subscribem::Constraints::SubdomainRequired) do
  ...
  get "/sign_up", :to => "users#new", :as => :user_sign_up
  ...
end
```

This new route will be matched first because it's higher in the `config/routes.rb` file than the other route.

The new route will send the subdomain request to another controller – Subscribem::Account::UsersController. If this route is set up correctly, running our test will tell us that the controlle isn't available yet:

```
Failure/Error: visit "http://#{account.subdomain}.example.com/sign_up"
ActionController::RoutingError:
  uninitialized constant Subscribem::Account::UsersController
```

We can generate this controller with this command:

```
rails g controller account/users
```

The next step is to create the `new` action within this controller, which will set the scene for this action's template to present a form for the user. Let's define this new action as this within `app/controllers/subscribem/account/users_controller.rb`:

```
def new
  @user = Subscribem::User.new
end
```

For the form for this action, we're going to take the pre-existing fields out of `app/views/subscribem/accounts/new.html.erb` and put them into a partial so that we can re-use it within our new user sign up form. Let's replace these things in that file:

```
<p>
  <%= owner.label :email %><br>
  <%= owner.email_field :email %>
</p>
<p>
  <%= owner.label :password %><br>
  <%= owner.password_field :password %>
</p>
<p>
  <%= owner.label :password_confirmation %><br>
  <%= owner.password_field :password_confirmation %>
</p>
```

With this line:

```
<%= render "subscribem/account/users/form", :user => owner %>
```

For that line to work, we'll need to move that above content into a new file, and change the `owner` references in that file to `user`:

app/views/subscribem/account/users/_form.html.erb

```
1  <p>
2    <%= user.label :email %><br>
3    <%= user.email_field :email %>
4  </p>
5  <p>
6    <%= user.label :password %><br>
7    <%= user.password_field :password %>
8  </p>
9  <p>
10   <%= user.label :password_confirmation %><br>
11   <%= user.password_field :password_confirmation %>
12 </p>
```

Using `owner` as a variable name here does not make much sense, since it's not clear what an "owner" is within this context. Therefore, we're calling the variable `user` instead, since this will be the form used for a user signing up.

Let's create the template for the `new` action within `Subscribem::Account::UsersController` now:

app/views/subscribem/account/users/new.html.erb

```
<h2>Sign Up</h2>
<%= form_for(@user, :url => do_user_sign_up_url) do |user| %>
  <%= render "subscribem/account/users/form", :user => user %>
  <%= user.submit "Sign up" %>
<% end %>
```

When we run the test again with `rspec spec/features/users/sign_up_spec.rb`, we'll see this:

```
undefined local variable or method `do_user_sign_up_url'
```

This is happening because we have not defined this routing helper within our `config/routes.rb` file. This is going to be the route where the form sends its data to. Let's define that now underneath the route we defined a moment ago:

```
get "/sign_up", :to => "users#new", :as => :user_sign_up
post "/sign_up", :to => "users#create", :as => :do_user_sign_up
```

Let's also define the `create` action within `Subscribem::Account::UsersController` too, as well as the `user_params` method to permit the attributes from the form:

```
def create
  account = Subscribem::Account.find_by!(:subdomain => request.subdomain)
  user = account.users.create(user_params)
  env["warden"].set_user(user, :scope => :user)
  env["warden"].set_user(account, :scope => :account)
  flash[:success] = "You have signed up successfully."
  redirect_to root_path
end
private
  def user_params
    params.require(:user).permit(:email, :password, :password_confirmation)
  end
```

In this `create` action, we attempt to find an account by its subdomain using `find_by!`, and passing it a hash, telling it the subdomain we want. If no account is found matching that subdomain, then an `ActiveRecord::RecordNotFound` exception would be raised. We then create a user for that account and sign in as that user. Finally, we send them back to the root path of our engine.

If we run the test again, we'll see it failing in this way:

```
Failure/Error: page.should have_content("You have signed up successfully.")
  expected there to be content "You have signed up successfully." in
    "Subscribem\n\n  Please sign in. ..."
```

This failure is happening because we've got a `before_filter` checking to see if users are signed in before they can access paths in the application. The `create` action within `Subscribem::Account::UsersController` isn't signing anybody in at the moment. We should fix that!

If an account *is* found, then the action calls `account.users.create` which helpfully adds the user to the `accounts.users` list, which would allow the user to sign in through the Subscribem::Account::SessionsController after they're done with the current session. The `create` call here uses `user_params`, which needs to be defined at the bottom of this controller like this:

```
private
  def user_params
    params.require(:user).permit(:email, :password, :password_confirmation)
  end
```

The final thing is action needs to do is actually authenticate the user, which can be done with these two lines, also found within the `Subscribem::AccountsController` class:

```
env["warden"].set_user(account.owner, :scope => :user)
env["warden"].set_user(account, :scope => :account)
```

Rather than just copying and pasting these two lines into the new controller, let's define a new method inside `Subscribem::ApplicationController` called `force_authentication!` that we can call in both `Subscribem::AccountsController` and `Subscribem::Account::UsersController`:

```
def force_authentication!(account, user)
  env["warden"].set_user(user, :scope => :user)
  env["warden"].set_user(account, :scope => :account)
end
```

Let's call this new method inside the `create` action for `Subscribem::UsersController` now:

```
def create
  account = Subscribem::Account.find_by!(subdomain: request.subdomain)
  user = account.users.create(user_params)
  force_authentication!(account, user)
  flash[:success] = "You have signed up successfully."
  redirect_to root_path
end
```

And we'll also replace the two lines inside the `account.valid?` check which is inside the `create` action for `Subscribem::AccountsController`:

```
def create
  @account = Subscribem::Account.create_with_owner(account_params)
  if @account.valid?
    force_authentication!(@account, @account.owner)
    flash[:success] = "Your account has been successfully created."
    redirect_to subscribem.root_url(:subdomain => @account.subdomain)
  else
    flash[:error] = "Sorry, your account could not be created."
    render :new
  end
end
```

By authenticating the user, the test now should be redirected to the proper place and shown the "You have signed up successfully." message. Let's run it again and find out:

```
1 example, 0 failures
```

Good! The test is now passing, which means that new users will be able to sign up to an account. Now we are enabling users to sign up for a new account, and letting other users then sign up to that account.

1.8 Summary

In this chapter we've laid the foundations for our subscriptions engine, going from having nothing to having an engine that allows a user to sign up for an account. Accounts provide the basis for the software as a service engine that we're building. The engine is going to provide applications with a way of scoping down resources, making them visible only to the accounts that they should be visible to.

We saw how we could use Warden to sign in as users after creating an account, how to work with subdomains within Capybara tests, and how to use strategies with Warden to authenticate users when they choose to sign in to a particular account, barring them when they don't meet the exact criteria of the proper subdomain, email address and password.

At the end of the chapter, we built sign up functionality and refactored the authentication code into a shareable method called `force_authentication!`. These three things (account sign up, user sign up and user sign in) provide a great foundation for our subscriptions engine.

In the next chapter, we're going to take a look at two potential ways we can do account scoping to limit resources to only specific accounts. The first way will be adding a field to a table that we want to scope and then scoping by that field, and the second will be a little more complex using a combination of PostgreSQL schemas and the Apartment gem. We'll delve into the pros and cons of both setups as well.