

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 2015-09-16



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

©2012 - 2015 Ryan Bigg

Contents

Acknowledgements	ii
1. A Grand Overview	1
1.1 Multitenancy	1
1.2 Software as a Service	1
1.3 The multi-tenant blog application idea	2
1.4 Summary	3
2. Laying the foundations	4
2.1 Building an engine	5
2.2 Setting up a testing environment	5
2.3 Writing the first feature	7
2.4 Associating accounts with their owners	11
2.5 Adding subdomain support	19
2.6 Building subdomain authentication	27
2.7 Handling user signup	43
2.8 Summary	48
3. Applying account scoping	49
3.1 Scoping by a database field	49
3.2 Using Postgres	58
3.3 The PostgreSQL Caveat	82
3.4 Scoping by a database field: redux	82
3.5 Summary	89
4. The blog application	90
4.1 Setting up Blorgh	90
4.2 Testing Subscribem Integration	92
4.3 Testing user sign in	95
4.4 Testing user sign up	99
4.5 Testing Blorgh integration	100
4.6 Summary	104
5. Subscriptions	105
5.1 Formulating plans	105
5.2 Switching plans	119
5.3 Subscribing with Braintree	129
5.4 Updating Braintree Subscriptions	147

CONTENTS

5.5	Summary	153
6.	Account Restrictions	154
6.1	Limiting forum creation	154
6.2	Summary	157

Thank you for reading Multitenancy with Rails.

If you want to see some code examples and compare them to what you have, you can go view them [over on GitHub](#)¹. For now, they're the absolute latest code from the book.

If you find any mistakes while reading this book, please email them to me@ryanbigg.com. If email's not your thing, I have a review tool that you can use too. I'll need your email address to add you to that though. Tweet me (@ryanbigg) or email is cool.

If it's a problem to do with your code, then please put the code on GitHub and link me to it so I can clone it and attempt to reproduce the problem myself. If you don't understand something, then it's more likely that I'm the idiot and rushed it when I wrote it. Let me know!

This far into the book and there's already one error. You should expect that it is not alone. It has friends and their ways are devious. They are coming after your perception of reality. **Beware.**

During the writing of [Rails 4 in Action](#)² (previously Rails 3 in Action), I wrote an engine called [Forem](#)³ which provides forum functionality for Rails applications in the form of a drop-in engine. The work on the Forem engine serves not only as a basis for the [Getting Started with Engines Rails Guide](#)⁴, but also this book.

I can't put it any simpler than this: the book you're reading right now uses the Forem engine and a couple of other goodies to build a multi-tenanted forum application. It sounds hard (hint: it is), but with this book's guidance you should learn to avoid the pitfalls that many other developers have encountered.

Where Rails 3 in Action is more of a "you know Ruby, now let's learn Rails" book, this book is directed at people who have at least some Rails experience and feel comfortable building Rails applications. As such, core concepts of Rails aren't explained as we go along in this book.

While there may be some unfamiliar concepts in this book, the explanation that comes with them should make you familiar with them in no time. This book is neither for beginners or experts, but something in between. I thought I would mention that because people have been confused about it before.

¹https://github.com/radar/saas_book_examples

²<http://manning.com/big2>

³<https://github.com/radar/forem>

⁴<http://guides.rubyonrails.org/engines.html>

Refund Comment

Book is interesting but the material doesn't apply to what i'm working on. I need something more general and entry level.

The book is targeted to beginners. I was looking more advanced content.

Refunds

Again: thank you for spending your money and buying this book. I really hope you get something valuable out of it.

Ryan Bigg

Acknowledgements

I'd like to acknowledge the help of Phil Arndt, Josh Adams, Andrew Hooker, and Rob Yurkowski for the initial discussions that lead to the creation of this book.

Thanks to Travis Skindzier for the cover art.

Thanks to Gary Bernhardt for his idea for `Account#create_with_owner`: <https://gist.github.com/4534954>.

Thanks to Spree Commerce for the idea of having a `TestingSupport` directory within an engine to share testing setup code.

Thanks to Gaurish Sharma for his refactoring of the Warden password strategy.

And finally, thanks to everyone who's submitted errata or offered to review: Rick Lloyd, Manfred Klaffenböck, Bruno Bonamin, David Briggs, Richard Wilson, Neal Farrand, Chris Maxwell, Chris McCann, Nicholas Mott, Brandon Campbell, Garrett Heinlein, Juanito Fatas, Manuel Enrique Viduarre, Mark Sobkowicz, Brian Sweany, Dmitry Zhlobo, Matt Borja, Carlos Diógenes, Bart Vandendriessche, Tony Semana, David Kewal, João Pedro Barbosa, Yiannis Deliyiannis, Sviatoslav Grebenchucov, Weston Platter, Gaurish Sharma, Juan Mercedes, Daniel Schierbeck, Craig Read, Daniel Fone, David Silva, Ender Yurt, Thomas Cioppettini, Sean Kelly, Raymond Rogers, Deryl Downey, Julian Russel, B Strand, Karampasis Manos, Randy Antler, Leonid Dinershtein, Alex Handley, Benjamin Tan, Prakash Murthy, John Johnson, Jamal Shaheen, Ismael G Marín C, Rubén Dávila, Norman Ancajas, Will Thomas.

1. A Grand Overview

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

The application we’ll be building in this book isn’t based off an original, outlandish idea, but a rather simple one: hosted blogs. This is a similar idea to Tumblr (<http://www.tumblr.com/>), or other blogging platforms.

For a user to be able to create their own posts, they can sign up to our application for an account (Chapter 2), which would be sandboxed from the other accounts on the system (Chapter 3). Once they’re done with that, they will then be able to create posts for their blog (Chapter 4). In the later chapters, we cover things such as monthly subscriptions and billing for them (Chapter 5).

That’s a quick overview of what you’ll be learning in this book as a whole. At the end of it, you will know how you can build a multi-tenanted Software as a Service Rails application. Woah, buzzwords! Let’s explain what is meant here.

1.1 Multitenancy

A single-tenant application is an application that is used by a single entity, be it a single user or a single company. It’s not important in this case that the data in this application be kept separate from the other data since all the data belongs to one entity.

A multi-tenancy application is one that can be used by multiple entities at the same time, typically with the data for those entities kept separate from the other entities. When a user signs in, they should only be able to see one set of data, and not the data from other places.

1.2 Software as a Service

To explain the term “Software as a Service” within the context of a multitenanted Rails application, we’re going to look at a Rails application that itself is a multitenanted SaaS application: GitHub.

If you’re reading this book, there’s a high probability you’re a developer¹ and therefore (by inference) a high probability you are aware of GitHub (<https://github.com>). GitHub, for the uninitiated, offers both public and private Git repository hosting services for the masses.

When using Git, you will create a repository which contains any number of files. This repository serves two purposes: it provides a container for the data, and tracks the changes to that data every time you make a commit to it. This much, you should already be familiar with.²

Users of GitHub can push their Git repositories to GitHub’s servers, where it can be made available for the masses or locked away to be shown only to a select few. One of the more excellent things about GitHub is

¹Of the Ruby on Rails variety, we hope! Otherwise, there’s been a terrible mistake made by whomever gave you this book.

²And if you’re not, the most excellent *Pro Git* book will teach you.

that it doesn't have to be code that is uploaded. For instance, the text of this book is on GitHub!³ As long as you are willing to provide open-access to your files and don't exceed a specific limit, you can use GitHub for free.

If, however, you want to make your files private or exceed that limit, then you're going to need to pay for the service. This is how GitHub makes their money: offer a free service for people and limit the best features to people who are willing to pay for it. GitHub's customers choose to pay a monthly amount to have GitHub staffers deal with the complexities of hosting their files securely, in turn receiving an easy to use, reliable service. GitHub also provides GitHub Enterprise (<https://enterprise.github.com/>) which is based off their web application, but is designed for enterprise businesses to use it within their own internal networks.

Once a repository has been made private, only the repository's owner has access to it by default. The owner can then add other people to the repository who are then able to view or make modifications to it. When people who are not on the repository's access list attempt to access that repository, GitHub denies them access to it.

Therein lies two of the most interesting technical aspects of SaaS applications. The first is ensuring that specific users can access some resource, while others cannot. The second is providing plans for the application that provide additional functionality for a small monthly fee. We will be focusing on both of these in the coming chapters.

1.3 The multi-tenant blog application idea

In this book, we're going to build a multi-tenant blogging application, where users can create accounts which are able to have their own posts. The accounts can then take user signup, and then those users will be able to post whatever they'd like to their own blogs.

In this book, we're going to start by building a subscription engine that can plug into Rails applications. This engine will provide management for accounts, plans, subscriptions and users. We'll start building this engine in the next chapter, Chapter 2.

The first feature for this engine will be allowing a user to sign up for a new account. The user will then be automatically associated with this account as its owner, meaning it will have permission to do administrative tasks for the account. After we're done with that, we'll allow the users to sign up to an existing account as a new user, as well as sign in.

Once we're done with this, we'll make it so that each account has its own subdomain. This will be critical in the future scoping of resources to a specific account. We'll also link users to specific accounts and only allow them to sign in on the accounts' subdomains that they have access to.

In Chapter 3, we're going to cover two different techniques for scoping data to ensure that when a user signs in for an account, they can only see that one account's data. We'll cover one way of implementing this restriction, albeit a kind of hackish way, using a database field. Later on in that same chapter, we'll show how we can do it in a much easier way using PostgreSQL's schemas⁴ and a gem called apartment (<https://github.com/influitive/apartment>).

In Chapter 4, we'll apply what we've learned in Chapters 2 and 3 to begin building a SaaS application for blogs, using the Blorgh app as a very simple base.

³Albeit in a top-secret repository, available only to the authors and select people.

⁴PostgreSQL schemas: <http://www.postgresql.org/docs/9.1/static/ddl-schemas.html>

In the later chapters of the book, we'll cover adding subscriptions for plans to accounts, so that accounts are limited to only a specific number of users. With this subscription management will come managing payments.

1.4 Summary

The main concepts for the GitHub case study above and many SaaS applications are that they provide accounts that have access to a certain amount of features in the application for a monthly fee.

In this book, you'll be creating an account-management engine that provides the backend account-management functionality for a SaaS application. You'll then use this engine within an application. When you're done, you'll have an application that has accounts which can subscribe to plans. The different plans will limit how many resources the accounts are entitled to each. That's a while off yet.

Let's get started!

2. 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](http://guides.rubyonrails.org/engines.html)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>

2.1 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.

2.2 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

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

This will also install the `rails` and `sqlite3` gems and their dependencies because they're specified in the `subscriber.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.

2.3 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-existent `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/subscribe/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 `Subscribe::AccountsController` yet, so let’s open up `app/controllers/subscribe/accounts_controller.rb` and add a new action that sets it up:

```
def new
  @account = Subscribe::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 Subscribe::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 `Subscribe::Account` and with it a migration which will create the `subscribe_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/subscribe/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 `Subscribe::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/subscribe/accounts_controller.rb` now:

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

2.4 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

```

4 <p>
5   <%= account.label :name %><br>
6   <%= account.text_field :name %>
7 </p>
8 <%= account.fields_for :owner do |owner| %>
9   <p>
10    <%= owner.label :email %><br>
11    <%= owner.email_field :email %>
12  </p>
13  <p>
14    <%= owner.label :password %><br>
15    <%= owner.password_field :password %>
16  </p>
17  <p>
18    <%= owner.label :password_confirmation %><br>
19    <%= owner.password_field :password_confirmation %>
20  </p>
21 <% 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 gem² 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"
```

²The 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 its 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/subscribe/application.html.erb

```

1 <% if user_signed_in? %>
2   Signed in as <%= current_user.email %>
3 <% 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 `Subscribe::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.

2.5 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

```
1 <p>
2   <%= account.label :subdomain %><br>
3   <%= account.text_field :subdomain %>
4 </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

```
1 class AddSubdomainToSubscribemAccounts < ActiveRecord::Migration
2   def change
3     add_column :subscribem_accounts, :subdomain, :string
4     add_index :subscribem_accounts, :subdomain
5   end
6 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/subscribe/")
  expected: "http://test.example.com/subscribe/"
  got: "http://www.example.com/subscribe/" (using ==)
```

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

```
redirect_to subscribe.root_url
```

To this:

```
redirect_to subscribe.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 to 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.

2.6 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 association 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

```

1 FactoryGirl.define do
2   factory :user, :class => Subscribem::User do
3     sequence(:email) { |n| "test#{n}@example.com" }
4     password "password"
5     password_confirmation "password"
6   end
7 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

```

1 constraints(Subscribem::Constraints::SubdomainRequired) do
2   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

```

1 module Subscribem
2   module Constraints
3     class SubdomainRequired
4       def self.matches?(request)
5         request.subdomain.present? && request.subdomain != "www"
6       end
7     end
8   end
9 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 Wiki³, 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
end
```

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.")
end
```

```

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 validity 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`

```

1 scenario "cannot sign in if not a part of this subdomain" do
2   other_account = FactoryGirl.create(:account)
3   visit subscribem.root_url(:subdomain => account.subdomain)
4   expect(page.current_url).to eq(sign_in_url)
5   expect(page).to have_content("Please sign in.")
6   fill_in "Email", :with => other_account.owner.email
7   fill_in "Password", :with => "password"
8   click_button "Sign in"
9   expect(page).to have_content("Invalid email or password.")
10  expect(page.current_url).to eq(sign_in_url)
11 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`

```

1 Warden::Strategies.add(:password) do
2   def subdomain
3     ActionDispatch::Http::URL.extract_subdomains(request.host, 1)
4   end
5   def valid?
6     subdomain.present? && params["user"]
7   end
8   def authenticate!
9     return fail! unless account = Subscribem::Account.find_by(subdomain: subdomain)
10    return fail! unless user = account.users.find_by(email: params["user"]["email"])
11    return fail! unless user.authenticate(params["user"]["password"])
12    success! user
13  end
14 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/subscribe-m/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/subscribe-m/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?

2.7 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 responsibility 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/subscribe/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 controller 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/subscribe/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

```

1 <h2>Sign Up</h2>
2 <%= form_for(@user, :url => do_user_sign_up_url) do |user| %>
3   <%= render "subscribem/account/users/form", :user => user %>
4   <%= user.submit "Sign up" %>
5 <% 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 this 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.

2.8 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.

3. Applying account scoping

Now that the application has accounts, we're going to be able to use them to scope specific resources within our application. In this chapter, we're going to cover two ways of doing that scoping.

The first way is a simple, semi-hackish kind of way. For every database table that requires this scoping, we'll add an `account_id` field to it. Then it's a simple matter of returning only results that have a requested `account_id`. This design has its own pitfalls, as we'll see when we develop this solution.

The second way that we'll show in this chapter is much neater in terms of how it operates, but is a little more complex to set up compared to simply adding a field to a database table *and* it'll only work on PostgreSQL. It has its own caveat: on large databases backups can take a large amount of time. This is due to the number of unique tables that are created. The other method does not have this problem because uses the one table for the entire set of data within the application.

We'll be using a feature of PostgreSQL called *schemas*¹ in combination with a gem called Apartment which makes dealing with PostgreSQL schemas a lot easier than not using it.

Let's look at how we can do scoping by a database field now.

3.1 Scoping by a database field

Scoping by a database field is a relatively easy thing to understand. All we need to do is to add a field called `account_id` to the table that contains the records that should be scoped by an account and ensure that every time a user is logged into an account, a query is made to this table that contains a condition for `account_id` matching the account's id. However, this comes with its own complexity, as there's no easy programmatical way to enforce this restriction. We, the programmers, must make *absolutely certain* that we have this scoping in place. Otherwise, this whole system falls apart. Resources in one area may be scoped by account id, while resources in another might not be.

In order to test that our scoping will work, we're going to need to have a model within the dummy application to play the role of the thing that will be scoped. We will also need to have some way of checking that these resources are scoped correctly.

What we'll do is write a test that will start with creating two accounts and will then create two records in the database, one for each account. When a user signs in for the first account, they should only be able to see that account's records. When they login with the second account, they should only be able to see the *other* account's records.

Writing a test

Let's write a test for this now in a new file called `spec/features/accounts/scoping_spec.rb`.

¹PostgreSQL schemas: <http://www.postgresql.org/docs/9.1/static/ddl-schemas.html>

spec/features/accounts/scoping_spec.rb

```

1  require "rails_helper"
2  feature "Account scoping" do
3    let!(:account_a) { FactoryGirl.create(:account) }
4    let!(:account_b) { FactoryGirl.create(:account) }
5    before do
6      Thing.create(:name => "Account A's Thing", :account => account_a)
7      Thing.create(:name => "Account B's Thing", :account => account_b)
8    end
9    scenario "displays only account A's records" do
10     sign_in_as(:user => account_a.owner, :account => account_a)
11     visit "/things"
12     expect(page).to have_content("Account A's Thing")
13     expect(page).to_not have_content("Account B's Thing")
14   end
15   scenario "displays only account B's records" do
16     sign_in_as(:user => account_b.owner, :account => account_b)
17     visit "/things"
18     expect(page).to have_content("Account B's Thing")
19     expect(page).to_not have_content("Account A's Thing")
20   end
21 end

```

In this test, we are using the FactoryGirl gem to create two new accounts, “Account A” and “Account B”. Before the tests are run, we create two “things”, one for each account. In the first test, we sign in as Account A’s owner and assert that we can only see Account A’s Thing record. In the second test, we assert that we can only see Account B’s Thing when signed in as Account B’s owner.

Creating a resource to scope by

The first thing we’ll see when we run this test with `rspec spec/features/accounts/scoping_spec.rb` is that we’re missing the Thing constant:

```

Failure/Error:
  Thing.create(:name => "Account A's Thing", :account => account_a)
NameError:
  uninitialized constant Thing

```

This constant will be an Active Record-backed model that will manage the objects that will be scoped by an account. This model will need a name field and a foreign key field to link to an account. The model is going to be outside the engine, which means that we’ll need to generate it inside the dummy app. We can do that by running these two commands:

```

cd spec/dummy
rails g model thing name:string account_id:integer

```

This generator will of course generate a migration which will need to be run. Switch back to the engine and run the migrations by running these two commands:


```
cd -
rake db:migrate RAILS_ENV=test
```



We need to run this `rake db:migrate` command with `RAILS_ENV=test`, because otherwise the migration would not be run on the test database. This happens only because the migration is a part of the dummy application. All our other migrations have been a part of the engine previously, and that is why they have worked.

This model will need to have the account association defined within it, so that assigning a `Thing` an account will work. Let's change the `Thing` model to accommodate this operation now:

`spec/dummy/app/models/thing.rb`

```
1 class Thing < ActiveRecord::Base
2   belongs_to :account, :class_name => "Subscribem::Account"
3 end
```

The `Thing` class now exists, which solves our last issue with our tests. Let's see what the next step is with another quick run of `rspec spec/features/accounts/scoping_spec.rb`.

```
Failure/Error: sign_in_as(:user => account_a.owner, :account => account_a)
NoMethodError:
  undefined method `sign_in_as' for ...
```

This method inside our test will be used to sign in as a user under a particular account, so that our engine has a basis on which to scope things.

Authenticating as a user

Let's define this method in a new file called `spec/support/authentication_helpers.rb` like this:

```
module AuthenticationHelpers
  include Warden::Test::Helpers
  def sign_in_as(options={})
    options.each do |scope, object|
      login_as(object, :scope => scope)
    end
  end
  RSpec.configure do |config|
    config.include AuthenticationHelpers, :type => :feature
    config.after :type => :feature do
      logout
    end
  end
end
```

We're including this one helper into all feature specs with this line, placed into the `RSpec.configure` block inside `spec/rails_helper.rb`:

```
RSpec.configure do |config|
  config.include AuthenticationHelpers, :type => :feature
  ...
end
```

Because it's more often than not that we're going to need to authenticate inside our engine's feature specs.

In the `sign_in_as` method, we go through each of the options that are passed to the method, which are just a user and an account. We then use the `login_as` method – provided by the `Warden::Test::Helpers` module, included at the top – to sign in through Warden. This means that upon the next request that our test makes, it'll be authenticated with this user and this account.

At the end of every test, we ensure that Warden is logged out by calling the `logout` method in an `after` block. If we didn't do this, Warden would keep the session across multiple tests, which could cause undesired results.

When we run `rspec spec/features/accounts/scoping_spec.rb` again, we'll see that it's now gotten a little bit further along:

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

The test is now attempting to access the `/things` route, which should display a page with a list of one account's things on it. This route doesn't exist within the dummy application which is why the test is failing. We can generate a controller, an action, a view and a route for that action by going back into the `spec/dummy` directory and running the controller generator:

```
cd spec/dummy
rails g controller things index
```

The route this generates is unfortunately incorrect, but easily fixable. It's located within the routes definition of `spec/dummy/config/routes.rb` and will currently be this:

```
get "things/index"
```

The route that we want defined is `/things`, not `/things/index`, and we want that route to go to `ThingsController#index`. For good measure, we'll also cause this to define a path helper for us:

```
get "/things" => "things#index", :as => :things
```

By having the controller, action, view and route now existing, the spec should be able to make a successful request to the `/things` route and get at least *something* back. Another run of the test will produce this error:

```
Failure/Error: page.should have_content("Account A's Thing")
expected there to be content "Account A's Thing" in ...
```

The content for the page hasn't yet been correctly setup. This is now ultimately what we want to test: that we're seeing only one account's objects within this view, and no others.

The responsibility of this scoping falls to the `ThingsController`. The `ThingsController` is what will find all the `Thing` objects for a given account, but right now instances of that controller have no concept of an account, besides what's available within the `env["warden"]` object.

If we attempt to set up the `ThingsController` to fetch all the things for the `current_account` object, like this:

```
class ThingsController < ApplicationController
  def index
    @things = current_account.things
  end
end
```

And then run the scoping test once again with `rspec spec/features/accounts/scoping_spec.rb`, we'll see that `current_account` isn't available to this controller:

```
Failure/Error: visit '/things'
NameError:
  undefined local variable or method `current_account' for ...
```

But it *is* available to controllers that inherit from `Subscribem::ApplicationController`. The `ThingsController` should not inherit from the engine's `Subscribem::ApplicationController` because that's some very tight coupling between the application and the engine. What would be better is having these methods available in a common, shareable location, such as `ApplicationController`.

Sharing authentication methods

Inside the `Subscribem::ApplicationController` there are the `current_user` and `current_account` methods, which do understand the concept of not only an account, but also a user. What we'll need to do is to make these methods available to the application as well, so that `ThingsController` can access these values. We'll also make the `user_signed_in?` and `authenticate_user!` methods available, as the application may need them.

To do this, we'll create something known as an *extender*. A *extender* modifies a piece of already existing code, such as a class or a module, and adds functionality to it. This process is also known as monkey patching². This extender that we'll create in a moment is going to re-open the `ApplicationController` class and add the `current_user`, `current_account`, `user_signed_in?` and `authenticate_user!` to it.

Let's define this new file within a new directory called `app/extendors`³:

`app/extendors/controllers/application_controller_extender.rb`

```
1 ::ApplicationController.class_eval do
2   def current_account
3     if user_signed_in?
4       @current_account ||= begin
5         account_id = env["warden"].user(:scope => :account)
6         Subscribem::Account.find(account_id)
7       end
8     end
9   end
10  helper_method :current_account
11  def current_user
12    if user_signed_in?
13      @current_user ||= begin
```

²http://en.wikipedia.org/wiki/Monkey_patch

³The directory is called `extendors` because `monkeypatches` is longer to write, and just seems *dirtier*.

```

14         user_id = env["warden"].user(:scope => :user)
15         Subscribem::User.find(user_id)
16     end
17 end
18 end
19 helper_method :current_user
20 def user_signed_in?
21     env["warden"].authenticated?(:user)
22 end
23 helper_method :user_signed_in?
24 def authenticate_user!
25     unless user_signed_in?
26         flash[:info] = "Please sign in."
27         redirect_to '/sign_in'
28     end
29 end
30 def force_authentication!(account, user)
31     env["warden"].set_user(user, :scope => :user)
32     env["warden"].set_user(account, :scope => :account)
33 end
34 end

```

We're putting this new file inside the `app/extenders` directory inside another directory called `controllers` so that we can logically group the code inside this directory which is extending controllers, and other, future code, which may be extending models or whatever else.

This new file now contains all the methods from `Subscribem::ApplicationController`, so we can delete the method definitions from that class now, making it empty once again.

This extender will need to be loaded so that the methods are made available on `ApplicationController`. To do that, we can put a new `to_prepare` hook within the engine's definition, inside the `Subscribem::Engine` class inside `lib/subscribem/engine.rb`, like this:

```

config.to_prepare do
  root = Subscribem::Engine.root
  extenders_path = root + "app/extenders/**/*.rb"
  Dir.glob(extenders_path) do |file|
    Rails.configuration.cache_classes ? require(file) : load(file)
  end
end

```

The `to_prepare` block here will run every time a new request happens under the development environment, and only once (while the server is starting up) in the production environment.

We use `Subscribem::Engine.root` to get the root path to the engine and then combine that with a string and pass that to `Dir.glob`, which will retrieve a list of all the files that match the combined string. For each of these files, they're required or loaded, depending on the setting for `cache_classes` in the current environment.

These changes mean that the methods will no longer be available within controllers that subclass `Subscribem::ApplicationController` – that is, all controllers within the engine – because `Subscribem::ApplicationController` inherits directly from `ActionController::Base`. These methods now live in `ApplicationController`, by way of an extender.

To make these methods available in `Subscribem::ApplicationController` once again, this controller needs to be switched to inherit from `ApplicationController`, rather than `ActionController::Base`:

```
module Subscribem
  class ApplicationController < ::ApplicationController
    end
end
```

With these methods now being added to the `ApplicationController` class through this decorator, the `current_account` method will now be available in `ThingsController`. When we run `rspec spec/features/accounts/scoping_spec.rb` again, we'll get this error:

```
Failure/Error: visit '/things'
NoMethodError:
  undefined method `things' for #<Subscribem::Account:...>
  # ./spec/dummy/app/controllers/things_controller.rb:3:in `index'
```

Ah, now we're getting somewhere! We're now going to need to add an association to the `Subscribem::Account` model for the `Thing` objects, but if we were to define this association right within the `Subscribem::Account` model, that would couple the engine's model to the application, always requiring a `Thing` model to exist, which isn't right.

Programatically creating an association

What we'll need to do is have a method that we can call inside the `Thing` model that would add this association to the `Subscribem::Account` model. Let's define a method just for that in a new file in our engine at `lib/subscribem/active_record_extensions.rb`:

`lib/subscribem/active_record_extensions.rb`

```
1 ActiveRecord::Base.class_eval do
2   def self.scoped_to_account
3     belongs_to :account, :class_name => "Subscribem::Account"
4     association_name = self.to_s.downcase.pluralize
5     Subscribem::Account.has_many association_name.to_sym, :class_name => self.to_s
6   end
7 end
```

We're putting this code inside a new file so that it can be made available to all classes that inherit from `ActiveRecord::Base`. Right now, all we have is the `Thing` class, but later on we *could* have more than just that.

Inside this new file, we call `class_eval` to evaluate the code within the class of `ActiveRecord::Base`. Inside that block, we define a method called `scoped_to_account` which defines the `belongs_to` association for an account, as well as an association on `Subscribem::Account` for many objects of the current class.

When this method is called within the context of the `Thing` class, it will define an association called "things" on the `Subscribem::Account` model, which is what we need for our test to be happy.

Let's replace the `belongs_to` association definition within `spec/dummy/app/models/thing.rb`:

```
class Thing < ActiveRecord::Base
  belongs_to :account, :class_name => "Subscribem::Account"
end
```

With a call to this new `scoped_to_account` method:

```
class Thing < ActiveRecord::Base
  scoped_to_account
end
```

In order for this method to be made available, the `active_record_extensions` file must be required, which we can do in `lib/subscribem/engine.rb` with this code:

```
require "subscribem/active_record_extensions"
```

When we run our scoping test again with `rspec spec/features/accounts/scoping_spec.rb`, we'll see that it's no longer complaining about a missing `things` method on the `Subscribem::Account` object:

```
Failure/Error: page.should have_content("Account A's Thing")
expected there to be content "Account A's Thing" in
"...Find me in app/views/things/index.html.erb..."
```

Alright, now it's no longer complaining about the missing `things` method. The test is now complaining that it can't see "Account A's Thing" within the template located at `app/views/things/index.html.erb`, and that's missing because we've not altered the template to go through the list of `@things` and show the objects.

Displaying all the account's things

Let's open up `app/views/things/index.html.erb` and show the list of `@things` now:

`app/views/things/index.html.erb`

```
1 <h1>Things</h1>
2 <ul>
3   <% @things.each do |thing| %>
4     <li><%= thing.name %></li>
5   <% end %>
6 </ul>
```

One more run of the test (`rspec spec/features/accounts/scoping_spec.rb`) will show us that all the parts are in place to allow this scoping to work:

2 examples, 0 failures

This scoping is working successfully now. This scoping is done by scoping the SQL query through a simple association call. We've got a test in place within the `spec/features/accounts/scoping_spec.rb` file – actually *two* tests – that ensure that we can sign in for two different accounts and only see the things associated with either account.

Now, think what would happen if we didn't have this test. We would need to validate that this scoping works *manually*, which is not only annoying, but also prone to error. With the test, we can be sure that this scoping is always in place.

Let's run all our tests now and see if they're passing by using `bundle exec rspec spec:`

```
Failures:
  1) ThingsController GET 'index' returns http success
     Failure/Error: get 'index'
     NoMethodError:
       undefined method `authenticated?' for nil:NilClass
     ...
24 examples, 1 failure, 10 pending
```

It would appear that we have a single failure, as well as 10 pending specs. Feel free to clear away those pending specs; they're just generated automatically from the generator commands we've issued previously. The failing test is a result of a generator too: the `rails g controller things` command that we ran to create the `ThingsController` for our new test. This spec is located in `spec/dummy/spec`, which is a bit wrong. We just don't care about our dummy app and testing it. All we care about is that the Subscribem engine works perfectly.

Therefore, we can remove this test. Running it again, we should see this:

```
13 examples, 0 failures
```

Great! A nice solid test base for the code we've written so far.

The Caveats

The first caveat with this kind of scoping is that we need to be extremely careful all over our application, ensuring that we always scope where we need to scope. This isn't such a big deal now as we only have one place that scoping is required, but as our application grows and we add more points of scoping, we also add more points where we could forget that scoping *and* make a mistake in the test; a point of failure within our application.

In addition to that, we're required to have this additional `account_id` field on the tables requiring scoping, which is necessary for this type of scoping, but can be achieved using other means.

We can solve the first caveat of multiple points of failures by moving the scoping logic to one point in our application, scoping the database queries at that point and that point only. We can use the `Apartment` gem in combination with PostgreSQL schemas for this, as we'll see shortly.

With this new type of scoping with PostgreSQL schemas, we will no longer need the `account_id` field either, solving the second caveat.

As we'll see in the next section, PostgreSQL schema scoping has its own caveats, too.

3.2 Using Postgres

In the previous section we covered one way of scoping in the database, using a foreign key called `account_id` to link records in a “things” table back to the “subscribers_accounts” table. This meant that the application would only be displaying one account’s things at a time, and not the other’s.

The second way of implementing this scoping that we’ll show in this chapter is much neater in terms of how it operates, but is a little more complex to set up over simply adding a field to a database table *and* it’ll only work on PostgreSQL. That’s ok, because for the purposes of this book, that’s all we’re going to be using anyway.

We’ll be using a feature of PostgreSQL called *schemas*⁴ in combination with a gem called Apartment which makes dealing with PostgreSQL schemas a lot easier than not using it.

However, there’s a caveat which was mentioned once at the beginning of this chapter but is very worth mentioning here again. It is best explained by this note from an article in Heroku’s devcenter:

The most common use case for using multiple schemas in a database is building a software-as-a-service application wherein each customer has their own schema. While this technique seems compelling, we strongly recommend against it as it has caused numerous cases of operational problems. For instance, even a moderate number of schemas (50) can severely impact the performance of Heroku’s database snapshots tool, PG Backups.

Therefore, this book recommends against this method of using schemas in PostgreSQL for a large SaaS application. The alternative method of scoping by database columns will not run into these same difficulties as mentioned. Nevertheless, it is still worthwhile investigating how to do this.

When you create a table in a PostgreSQL database, PostgreSQL will create it underneath that database’s `public` schema automatically. When you perform a query on a PostgreSQL database, it will check the *schema search path* and will look for tables inside schemas that match the schemas listed with the schema search path. By default, this schema search path contains two schemas: the `public` schema, and a schema matching the current PostgreSQL’s user’s login.

We can use the schema search path to our advantage to perform the scoping restriction. We can do this by putting an account’s resources within one schema and then when a user signs in for that account, restricting the schema search path to look up just the one schema for that user’s account.

This means that we wouldn’t need to have an `account_id` field on all the tables requiring scoping, and we wouldn’t need to ensure that we’re scoping in all the right places, just the one. This scoping will be done using the [apartment](https://github.com/influitive/apartment)⁵ gem, in conjunction with PostgreSQL schemas.

When a user signs in for an account, we’ll switch PostgreSQL’s schema search path to look up in that account’s schema. This feature will mean that users will only be able to see data belonging to that account. This switching will be handled automatically with a feature of a gem called “Apartment”, which we’ll see later.

⁴PostgreSQL schemas: <http://www.postgresql.org/docs/9.1/static/ddl-schemas.html>

⁵<https://github.com/influitive/apartment>

Introducing the schema search path

Before we take a look at the Apartment gem, we must first understand what shortcuts this gem provides. The first shortcut is the ability to easily switch the schema search path with PostgreSQL. Let's see how we can use this schema search path *without* the Apartment gem first. And even *without* Rails and *without* Ruby. Let's get our hands dirty with some raw SQL.

Start up a PostgreSQL console with the `psql` command. In this prompt, run this command:

```
SHOW search_path;
```

When you run this query, the result will be this:

```
search_path
-----
"$user",public
(1 row)
```

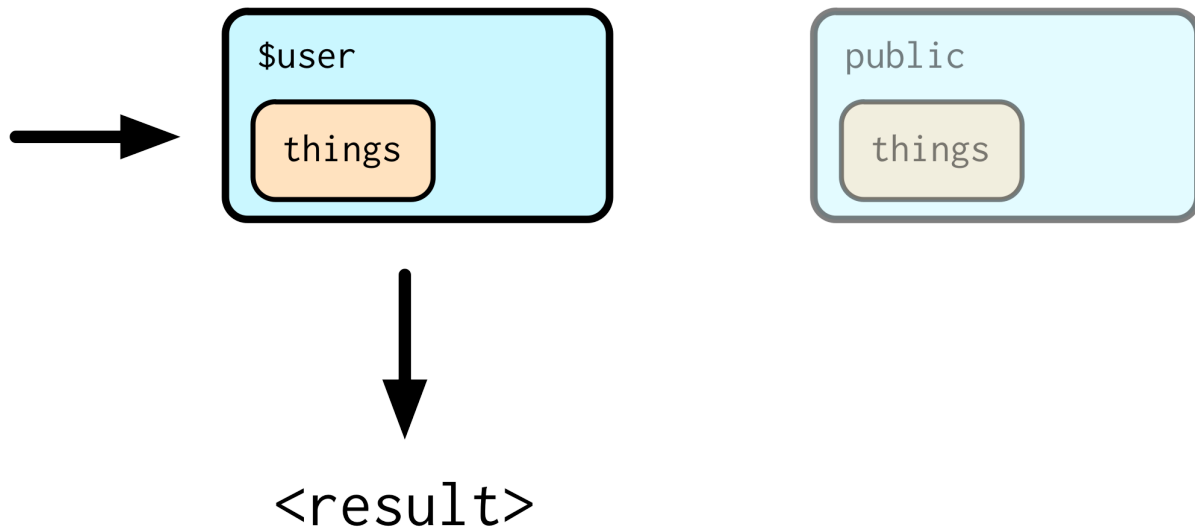
This is the default schema search path for PostgreSQL. The "\$user" part means that PostgreSQL will attempt a lookup in the database for a schema that matches the connection's username.

The connection's username is typically your shell login name. It's the same as `whoami`, in this instance, because of how we've set up PostgreSQL to use `ident` authentication.

If there is a schema matching the user's name, then PostgreSQL will look in that schema for a table that matches the query. If there isn't, then PostgreSQL will move on to the next schema in the schema search path, which in this case is `public`. If there's a table then, PostgreSQL will query that table. If there isn't a table, an error will be thrown.

Let's imagine there's a table called "things" within the user's schema, and another table called "things" within the public schema. With this default lookup path, attempting to run a query such as `SELECT * FROM things;` will do this:

```
SELECT * FROM things;
```

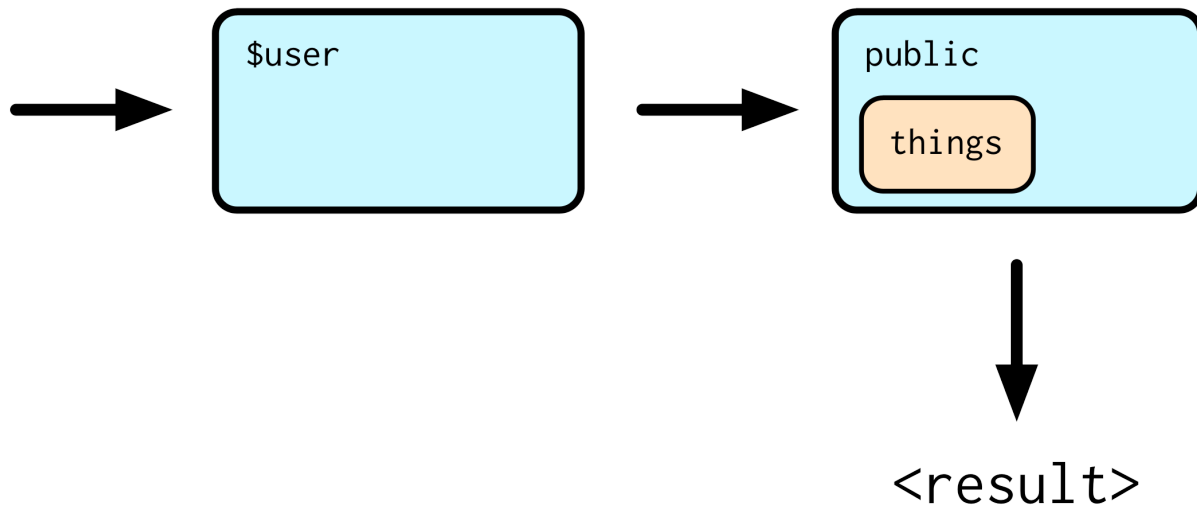


User schema lookup

Because the “things” table within the `$user` schema exists, PostgreSQL will return results from that table, ignoring the table within the `public` schema.

If the `$user` schema didn’t have that table, then the lookup would go like this:

```
SELECT * FROM things;
```



public schema lookup

It checks the user schema and then when it can't find the table, falls back to the public schema, which does contain the table and so some data is returned.

Using the public schema

If you've ever worked on a Rails application prior to reading this book that's used PostgreSQL, and you've not used different schemas, then you would still be using the public schema. By default, when you issue a `CREATE TABLE` statement, it will create a table in the `public` schema, which is the default schema for all PostgreSQL databases.

Let's verify this now by creating a new database called "subscribem" by running this command in the `psql` prompt:

```
CREATE DATABASE subscribem;
```

This command will create a new database, which will have a default "public" schema. We can see this if we change into the database using this command:

```
\c subscribem
```

Then if we run the `\dn` command, we can see the schemas:

```
List of schemas
  Name | Owner
-----+-----
 public | postgres
1 row)
```

We can see here that the “public” schema is indeed within this new database. Let’s create a new table now called `accounts`, and just put an `id` column in it by running this query:

```
CREATE TABLE accounts (
  id integer
);
```

Calling `CREATE TABLE` like this, will create a table in the `public` schema, as was explained before. To see proof of this table being within the `public` schema, run the `\dt` command:

```
subscriber=# \dt
      List of relations
 Schema | Name   | Type  | Owner
-----+-----+-----+-----
 public | accounts | table | us
(1 row)
```

In this output, we see that the only table in the database, called “accounts”, is within the “public” schema, and is owned by “us”, as that’s the user it was created with.

To insert data into this table, we can issue this query:

```
INSERT INTO accounts VALUES (1);
```

We can then get back this data by running this query:

```
subscriber=# SELECT * FROM accounts WHERE id = 1;
 id
----
  1
(1 row)
```

What this indicates to us is that we’ve got data inside a table called “accounts” inside the “public” schema of our “subscriber” database, and we’re able to pull that data out if we choose.

Creating an alternative schema

Now let’s see what happens when we create and use another schema. Creating this new schema is as easy as issuing a `CREATE SCHEMA` query:

```
CREATE SCHEMA alternative;
```

To create a new table in this schema, we can issue this query:

```
CREATE TABLE alternative.accounts (
  id integer
);
```

In this query, we’ve put the schema name before the table name so that PostgreSQL knows that we mean to create a table in this new schema. If we didn’t do this, then PostgreSQL would attempt to create this table in the public schema, which would fail because there’s already an accounts table that exists.

If we run the `\dt` command again in the `psql` prompt, we’ll see this output:

```
subscribem=# \dt
      List of relations
Schema | Name   | Type | Owner
-----+-----+-----+-----
public | accounts | table | us
(1 row)
```

Rather than it containing the table that we just created, it only contains the table from the `public` schema. Why PostgreSQL chooses to do this may be perplexing at first, but in fact the reason is rather a simple one.

It all has to do with the default schema search path in PostgreSQL. The schema search path right now is `$user,public`, which means that PostgreSQL will look up tables in a schema that matches the current user’s name, as well as in the “public” schema.

The new table is created within the “accounts” schema, which isn’t even in the schema search path, so why would PostgreSQL even *bother* looking there? To make PostgreSQL aware of this new table, we need to change the schema search path. To do that, we can run this query:

```
SET search_path = 'alternative';
```

When we run `\dt` again, we’ll see the correct table this time:

```
subscribem=# \dt
      List of relations
Schema | Name   | Type | Owner
-----+-----+-----+-----
alternative | accounts | table | ryan
(1 row)
```

This table isn’t going to have any data in it at all, since it’s just freshly created, and we can see this when we run a `SELECT` query on that table like this:

```
SELECT * FROM accounts;
 id
----
(0 rows)
```

PostgreSQL is now querying the `alternative.accounts` table, which has no data, compared with the `public.accounts` table, which has one row.

We could also run this query with the schema name before the table as well:

```
SELECT * FROM alternative.accounts;
 id
----
(0 rows)
```



Querying with a schema

If we were to run a query such as `SELECT * FROM public.accounts` at this point, it will still return the data from the `accounts` table within the `public` schema. This is because PostgreSQL is smart enough to know we're being explicit about what schema we're querying from, meaning we don't need to edit the schema search path to query a different schema.

Let's add some data to this new table now with this query:

```
INSERT INTO accounts VALUES (2);
```

Of course now when we run the `SELECT * FROM alternative.accounts` query, we'll see that this table definitely has the new row:

```
SELECT * FROM alternative.accounts;
 id
----
  2
(1 row)
```

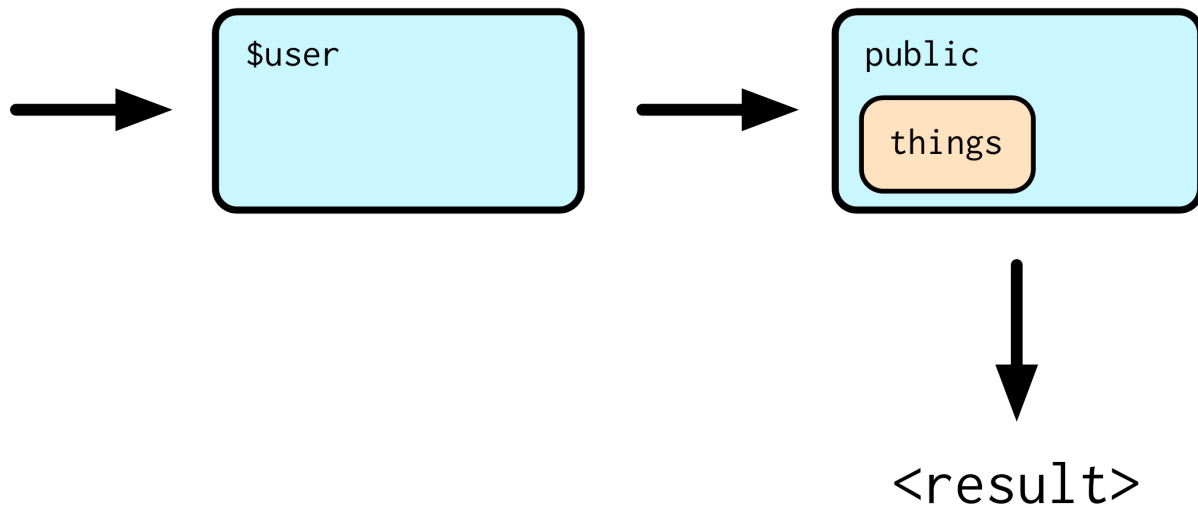
Setting precedence in the lookup path

In some cases, we may wish for PostgreSQL to check one schema for data before another. A typical usecase for this may be dealing with data for accounts within our Rails application. Each account would have its own schema, and then we would configure PostgreSQL's `search_path` to look up data in this schema first before moving on to `public`.

For instance, we may wish for PostgreSQL to attempt looking up the users for an account within that account's schema, rather than querying a whole table of users residing in the `public` schema and then scoping them by a foreign key. This is one of the benefits of using schemas.

Another benefit is the automatic fallback feature of schemas. If a table is missing from one schema in the search path, PostgreSQL will see if the table is contained in any of the other schemas (in the order they're listed) and if so, it'll then query that table. We saw this earlier on when we talked about schema path lookup:

```
SELECT * FROM things;
```



public schema lookup

If we set the search path to use the alternative schema first, and then the public schema, we can see both of these benefits in action:

```
SET search_path = 'alternative', 'public';
```

Selecting all the records from the accounts table with the alternative schema specified first, will only return the row from the alternative.accounts table, with id of 2:

```

SELECT * FROM accounts;
 id
----
  2
(1 row)

```

However, if we were to create a new table in the public schema with this query:

```

CREATE TABLE public.users (
  id integer
);

```

And then insert data into this table with this query:

```
INSERT INTO public.users VALUES(1);
```

We would be able to query this table's data like this, and it would still work:

```
SELECT * FROM users;
 id
----
  1
(1 row)
```

The search path will still contain the alternative schema, followed by the public schema:

```
SHOW search_path;
 search_path
-----
 alternative, public
(1 row)
```

The alternative schema doesn't have this table, but the public schema does. This is a great demonstration of how the schema search path can be used to look up tables across schemas.

Now that we've seen how we can work with PostgreSQL schemas, let's see how the Apartment gem helps us within the context of the subscribem engine.

Using the apartment gem

Before we use the apartment gem within our engine, we'll need to switch to using PostgreSQL for the dummy application, as this is what we're going to be using for the remainder of this book.

To do this, we're going to need to make two changes. First, we'll need to swap out the `sqlite3` gem dependency in `subscribem.gemspec` for one that is for the `pg` gem instead. Secondly, we'll need to change our dummy application to use a PostgreSQL database, rather than an SQLite3 one, for its environments.

Let's switch out the gem dependency first, which we can do by opening `subscribem.gemspec` and replacing this line:

```
s.add_development_dependency "sqlite3"
```

With this line:

```
s.add_dependency "pg"
```

We've switched this from a *development* dependency to a proper dependency because our engine will not work properly with any other database system. It's PostgreSQL or bust.

To tell our dummy application to use PostgreSQL, open up `spec/dummy/config/database.yml` and replace its contents with this:


```
development:
  adapter: postgresql
  database: subscribem_development
  min_messages: warning
test:
  adapter: postgresql
  database: subscribem_test
  min_messages: warning
```

We have removed the production key from this yaml file, as our dummy application will never be used in production.

About that `min_messages` database.yml key

In the database.yml file above, we've used a key called `min_messages`. Setting this key to 'warning' will silence NOTICE log messages from PostgreSQL, reducing the noise of things like migrations that create tables in the database.

We can create these two new PostgreSQL databases and run migrations on each database by running these commands:

```
rake db:create:all
rake db:migrate
RAILS_ENV=test rake db:migrate
```

Let's add the apartment gem to our engine now as a real dependency inside `subscribem.gemspec`:

```
s.add_dependency "apartment", "0.25.2"
```

At this point, we'll need to run `bundle install` to install the pg and apartment gems if they aren't already installed.

We'll need to require this gem inside our `lib/subscribem/engine.rb` too:

```
require "apartment"
```

Now that these gems are installed, we'll be able to use the Apartment gem to work with PostgreSQL schemas without having to write the SQL ourselves.

Using Apartment with schemas

The first thing that Apartment affords us is the ability to not only create schemas, but run the database migrations in those schemas as well. We can see this in effect when we go into the `spec/dummy` directory, run a rails console session and then create a new schema in this database using this code:

```
Apartment::Tenant.create("one")
```

Rails is helpfully showing us the SQL output of this command, which is doing more than simply just creating a schema. Sure enough, it *does* start with creating a schema:

```
CREATE SCHEMA "one"
```

But then after this there's a bunch more SQL queries because the Apartment gem is incredibly smart and knows what we want to do next. The next query is a query to create the subscribem_accounts table (made pretty in the following example):

```
CREATE TABLE "subscribem_accounts" (
  "id" serial primary key,
  "name" character varying(255),
  "created_at" timestamp NOT NULL,
  "updated_at" timestamp NOT NULL,
  "owner_id" integer,
  "subdomain" character varying(255)
)
```

The one after that is one to create the subscribem_accounts_users table. The two queries after that create the subscribem_users and things tables.

These queries are the same queries that would be generated when we run the `RAILS_ENV=test rake db:migrate` command. What Apartment is doing here is creating the schema called “one” *and then also running the migrations for that schema*. This is great news for us, because it's less work for us when we create a new schema.

We can see that these tables have actually been created within this new schema by starting up a new `psql` session with `psql subscribem_development` and then setting the search path and asking for a list of tables:

```
SET search_path = 'one';
\dt
```

We'll see the tables are safely nestled inside this schema:

```
List of relations
 Schema |          Name          | Type | Owner
-----+-----+-----+-----
  one   | schema_migrations      | table | ryan
  one   | subscribem_accounts     | table | ryan
  one   | subscribem_accounts_users | table | ryan
  one   | subscribem_users       | table | ryan
  one   | things                 | table | ryan
(5 rows)
```

Let's check what the schema search path is for the current Rails console session by putting this code into the console:

```
ActiveRecord::Base.connection.schema_search_path
# => "public"
```

As stated before, the default schema search path for PostgreSQL is “public”, which is the default schema for PostgreSQL databases and where a Rails application’s tables are typically created if the application is using PostgreSQL.

We’re now using multiple schemas in our application, and with that comes a small caveat: if we attempt to query a model now, we’ll run into difficulty. For instance, if we attempted a simple `find` on the `Subscribem::Account` model:

```
Subscribem::Account.first
```

PostgreSQL would report back that this table doesn’t exist, albeit in slightly more words than that due to the query that Active Record uses to find information about a table:

```
ActiveRecord::StatementInvalid:
PG::Error: ERROR: relation "subscribem_accounts" does not exist
LINE 4:          WHERE a.attrelid = "'subscribem_accounts'":reg...
                        ^
...
```

This error is happening because the `public` schema doesn’t contain the `subscribem_accounts` table; the `one` schema does.

To change the schema search path for Active Record’s benefit, we can use the `Apartment::Tenant.switch` method:

```
Apartment::Tenant.switch("one")
```

Schema switching with straight Active Record

We could’ve also switched the schema search path by calling `ActiveRecord::Base.connection.schema_search_path = 'one'`. The only real reason for using `Apartment::Tenant.switch` is because the method call is shorter, and that the `Apartment` gem offers some other useful tools like being able to exclude models.

Now when we attempt to find the first account, Active Record will return `nil` instead:

```
Subscribem::Account.first
# => nil
```

Active Record is now configured correctly to look up in the correct schema. All subsequent calls in this rails console session to Active Record-backed models will use the “one” schema, and not the “public” schema. The only accounts record that exists within our database, exists only within the `public.accounts` table.

Now that we’ve seen how to use PostgreSQL schemas and the `Apartment` gem, let’s put this stuff into action. The first thing that we’re going to need to do is to create schemas for accounts when a new account is created. By automatically creating the schemas, they will be available when we need to query them for schema-specific resources.

Automatically creating schemas

When an account is created for our engine, its record will live in `public.accounts`, but the associated data for this should live in another schema, so that the data is separate from every other account.

To create this schema, we'll make a call to `Apartment::Tenant.create` in a new method on the `Account` model. It's important that this new method works because if it doesn't, then any attempt to create data in that account's schema would fail dramatically. Therefore, we should write a test to ensure that this works. Let's put this new test inside `spec/models/subscribem/account_spec.rb`.

`spec/models/subscribem/account_spec.rb`

```

1  require 'rails_helper'
2  describe Subscribem::Account do
3    def schema_exists?(account)
4      query = %Q{SELECT nspname FROM pg_namespace
5                WHERE nspname='#{account.subdomain}'}
6      result = ActiveRecord::Base.connection.select_value(query)
7      result.present?
8    end
9    it "creates a schema" do
10     account = Subscribem::Account.create!({
11       :name => "First Account",
12       :subdomain => "first"
13     })
14     account.create_schema
15     failure_message = "Schema #{account.subdomain} does not exist"
16     assert schema_exists?(account), failure_message
17   end
18   ...
19 end

```

The gist of this test is that when an account is created and then the `create_schema` method is called on it, a schema should exist called `#{account.subdomain}`.

We assert that the schema has been created by running a query where we select from a table called `pg_namespace`; a table that we never created ourselves. This table is a PostgreSQL system catalog that comes with the database when it is created. This particular catalog contains a list of all the schema names, along with information about who created them. This is going to be really helpful for figuring out if a schema exists within the database or not.



More about PostgreSQL catalogs

The PostgreSQL manual is quite thorough when it comes to explaining what each catalog does. If you're interested in the minutiae of this feature, the documentation is here: <http://www.postgresql.org/docs/9.2/static/catalogs.html>

We run the query that selects from `pg_namespace` by using `ActiveRecord::Base.connection.select_value`, a method that will return a single value result for a query. Calling `present?` on this result will return `true` if the query did return a value, or `false` if it did not. This is what forms the basis for the `schema_exists?` method.

In the test, creating an account and then calling `create_schema` should result in a schema for that account then existing. We check that this is happening by using `assert` and the `schema_exists?` method we defined first in the test. If the account has no schema, then our test fails and prints out a helpful message indicating exactly that.

When we run this test with `rspec spec/models/subscribem/account_spec.rb` we'll see that the `create_schema` method doesn't exist:

```
Failure/Error: account.create_schema
NoMethodError:
  undefined method `create_schema' for #<Subscribem::Account:...>
```

This method should create the schema for an account when it's called. We can do this using this code placed inside the `Subscribem::Account` class:

```
def create_schema
  Apartment::Tenant.create(subdomain)
end
```

It's not by accident that we'll be using the subdomain as the schema name here. We're doing this on purpose because when we want to easily switch between schemas, we can use a piece of middleware provided by the `apartment` gem to automatically switch between schemas based on the subdomain. More on that later.

When we run the test again, we'll see that it now passes:

```
1 example, 0 failures
```

Good! Now we need to use this `create_schema` method within the `Subscribem::AccountsController` so that when a user signs up for a new account, their schema is automatically created. To test that this method is actually called, we'll write a controller test for it in a new file called `spec/controllers/subscribem/accounts_controller_spec.rb`:

`spec/controllers/subscribem/accounts_controller_spec.rb`

```
1 require "rails_helper"
2 describe Subscribem::AccountsController do
3   context "creates the account's schema" do
4     let!(:account) { Subscribem::Account.new }
5     before do
6       expect(Subscribem::Account).to receive(:create_with_owner).
7         and_return(account)
8       allow(account).to receive(:valid?).and_return(true)
9       allow(controller).to receive(:force_authentication!).and_return(true)
10    end
11    specify do
12      expect(account).to receive(:create_schema)
13      post :create, :account => { :name => "First Account" }
14    end
15  end
16 end
```

We have altered what `create_with_owner` would return here, as we want only to ensure that the controller calls this method, not that it actually follows through with its action. The test for that lies within `spec/models/subscribem/account_spec.rb` already, so there's no point testing it here too. We've stubbed `valid?` to always return `true` for the account object too, because we don't care about validations either.

When we run this spec with `rspec spec/controllers/subscribem/accounts_controller_spec.rb`, we'll see this strange error:

```
ActionController::RoutingError:
  No route matches {:account=>{:name=>"First Account"},
                  :controller=>"subscribem/accounts",
                  :action=>"create"}
```

This error claims that there's no route *at all* for the `Subscribem::AccountsController`'s `create` action, even though we know that's not the case.

This error is happening because the test is attempting to find the route for this controller inside the dummy app, rather than in our engine. Our dummy app simply *doesn't* have a route defined for `Subscribem::AccountsController`; that route is defined within the engine which is mounted in the application. Therefore we need to tell the test to look up the route in the correct location.

To do that, we can change the `post` call inside the `specify` block to use the `:use_route` parameter, like this:

```
post :create, :account => { :name => "First Account" },
     :use_route => :subscribem
```

What this option will do is tell the Rails router that we mean to look up the route within the context of the engine with the routing name of `subscribem`, which just so happens to be our engine's name.

The next time we run the test, we'll see this error:

```
Failure/Error: account.should_receive(:create_schema)
  (#<Subscribem::Account:0x007fc4e534a038>).create_schema(any args)
    expected: 1 time
    received: 0 times
```

The way to fix this is simple: insert a line to create the schema inside the `create` action like this:

```
def create
  @account = Subscribem::Account.create_with_owner(account_params)
  if @account.valid?
    force_authentication!(@account, @account.owner)
    @account.create_schema
  end
end
```

Running the test again will now make it pass:

1 example, 0 failures

Excellent! Now that we have accounts having schemas created automatically in the situations where they're required, the next step is to put these schemas to use.

Switching between schemas

We'll start off with a basic test now to just demonstrate (for testing purposes) that schema switching is working properly. In the next chapter when we start adding in post functionality, we'll be applying this schema scoping on a much bigger scale.

We'll be using `spec/features/accounts/scoping_spec.rb` to test that schema switching is working fine. This test will require some modifications first. The first modification is to ensure that the accounts in the test have schemas created for them which we can do by replacing these two lines:

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

With these lines:

```
let!(:account_a) { FactoryGirl.create(:account_with_schema) }
let!(:account_b) { FactoryGirl.create(:account_with_schema) }
```

Then we can define a new factory inside `spec/support/factories/account_factory.rb`, nested inside the factory `:account` call there:

```
FactoryGirl.define do
  factory :account, :class => Subscribem::Account do
    ...
    factory :account_with_schema do
      after(:create) do |account|
        account.create_schema
      end
    end
  end
end
```

By nesting this new factory inside the old one, we'll still be able to create an account without a schema by calling `FactoryGirl.create(:account)`, but in cases where we do want a schema, like in this test, we'll be able to call `FactoryGirl.create(:account_with_schema)`.

Next up in the test, we've got some lines which are creating Thing objects:

```
before do
  Thing.create(:name => "Account A's Thing", :account => account_a)
  Thing.create(:name => "Account B's Thing", :account => account_b)
end
```

These Thing objects will now need to be created inside each account's schema, which requires a couple more lines, like this:

```
before do
  Apartment::Tenant.switch(account_a.subdomain)
  Thing.create(:name => "Account A's Thing")
  Apartment::Tenant.switch(account_b.subdomain)
  Thing.create(:name => "Account B's Thing")
  Apartment::Tenant.reset
end
```

With the first two lines, we switch into account_a's schema and create a Thing there. On the next two lines, we do the same but for account_b. On the final line, we reset the schema search path using Apartment::Tenant.reset. If we didn't do this, the schema search lookup path would still be set to account_b's schema, meaning that the second test inside this file may pass regardless of whether or not the schema switching was in place.

That final line will also need to be inside a config.after(:each) inside spec/rails_helper.rb to ensure that the database state is reset after each test:

```
config.after(:each) do
  Apartment::Tenant.reset
end
```

Finally, we're going to need to not visit just /things any more, but rather /things within the scope of a subdomain. For the first test, this will need to be account_a's subdomain, like so:

```
scenario "displays account A's records" do
  sign_in_as(:user => account_a.owner, :account => account_a)
  visit main_app.things_url(:subdomain => account_a.subdomain)
  ...
end
```

And for the second test, account_b's:

```
scenario "displays account B's records" do
  sign_in_as(:user => account_b.owner, :account => account_b)
  visit main_app.things_url(:subdomain => account_b.subdomain)
end
```

When we run this test with `rspec spec/features/accounts/scoping_spec.rb`, both tests will fail:


```

1) Account scoping displays account A's records
Failure/Error: page.should have_content("Account A's Thing")
  expected there to be content "Account A's Thing" in "Things"
...
2) Account scoping displays account B's records
Failure/Error: page.should have_content("Account B's Thing")
  expected there to be content "Account B's Thing" in "Things"
...

```

This is understandable, as we’ve changed the setup for these tests. To make these test pass once again, we’re going to be using a feature of the apartment gem called an “Elevator”, which has actually nothing at all to do with elevating anything. The “Elevator” that we’ll be using is a piece of middleware called `Apartment::Elevator::Subdomain`, which will switch between schemas based on the subdomain for a request. So, as you can see, it’s just named an elevator because it’s a neat pun. Better than `Apartment::TenantSwitcher::Subdomain` in my opinion!

This piece of middleware will sit between Rack and the Rails application that the Subscribem engine is mounted in. When a request comes into the application, it will go through a path like this:

When it reaches the `Apartment::Elevator::Subdomain` middleware piece, that piece leaps into action. It will check the subdomain for the request and then switch the schema lookup path to include the schema which has the same name as the subdomain.

We’re going to be using this piece of middleware to modify behaviour within the parent application (to scope things correctly), and so we’ll insert this middleware into a new initializer within `lib/subscribem/engine.rb`, like this:

```

initializer "subscribem.middleware.apartment" do
  Rails.application.config.middleware.use Apartment::Elevators::Subdomain
end

```

We’ll need to require this class at the top of this file:

```
require "apartment/elevators/subdomain"
```

By having this middleware located here, the things should be well on their way to being scoped. Let’s find out if they’re there by running the test again. When we do that, we’ll see the first test fails like this:

```

Failure/Error: visit main_app.things_url(:subdomain => account_b.subdomain)
ActiveRecord::RecordNotFound:
  Couldn't find Subscribem::Account with id=<id>

```

This test is failing because it can’t find an account. That much is pretty obvious from the error message. But why can’t it find an account? For that answer, we must look deeper. This issue is actually caused by the schema switching stuff we have just implemented. The root of this problem is that the schema lookup path no longer contains the schema which contains the accounts: the `public` schema.

To fix that problem, we can exclude some models from Apartment’s scoping. We only need to exclude the `Subscribem::Account` model for now, but later on we’ll need to do the `Subscribem::Member` and `Subscribem::User` too, so let’s exclude both of these models now with these lines in a new file called `config/initializers/apartment.rb`:

```
Apartment.excluded_models = ["Subscribem::Account",
                             "Subscribem::Member",
                             "Subscribem::User"]
```

When we re-run the test again, we'll see it failing in this way:

```
Failure/Error:
  page.should have_content("Account A's Thing")
    expected there to be content "Account A's Thing" in "Things"
```

This failure has to do with how the things are being fetched within `ThingsController`. The way we're doing that is with this method call:

```
@things = current_account.things
```

This worked perfectly fine before when we were scoping by a database field, but now that we're scoping by a schema it won't work. Rather than that line, we can just call `Thing.all`, like this:

```
@things = Thing.all
```

Because we're no longer using the scoping within the `Thing` model, we can get rid of the `scoped_to_account` method within the `Thing` model and the `lib/subscribem/active_record_extensions.rb` file that provides this method. The `require` line within `lib/subscribem/engine.rb` can go away too.

When we run the test again, we'll see now that the first test is passing, but the second one is failing with this error:

```
Failure/Error:
  let!(:account_a) { FactoryGirl.create(:account_with_schema) }
ActiveRecord::StatementInvalid:
  PG::Error: ERROR: current transaction is aborted,
  commands ignored until end of transaction block
  : SET search_path TO public
```

This test is failing because `RSpec` is configured, by default, to run each test inside a transactional block. It does this so it can issue a `ROLLBACK` call at the end of each test to roll the database back to the pristine state it was in before the test started running. These transactions are what are causing our grief. We can disable these transactions by removing this line from our `spec/rails_helper.rb` file:

```
config.use_transactional_fixtures = true
```

Of course, this doesn't come without side-effects. Our database will no longer be cleaned up after each test. In order for it to be cleaned up, we'll be using the `database_cleaner` gem. This gem will truncate all the tables within the `public` schema, and will leave the other schemas untouched. We'll need to clean up those schemas too, but first let's focus on getting `database_cleaner` installed.

We can add `database_cleaner` to our `subscribem.gemspec` as a development dependency with this line:

```
s.add_development_dependency "database_cleaner", "1.3.0"
```

We can then install the gem by running `bundle install` once again. With it installed, we can then use its features by putting this code inside the `RSpec.configure` block of `spec/rails_helper.rb`, adapting it to the `config.after(:each)` block that was already in this file:

```
config.before(:all) do
  DatabaseCleaner.strategy = :truncation,
    { :pre_count => true, :reset_ids => true }
  DatabaseCleaner.clean_with(:truncation)
end
config.before(:each) do
  DatabaseCleaner.start
end
config.after(:each) do
  Apartment::Tenant.reset
  DatabaseCleaner.clean
end
```

Inside the `before(:all)` block, we're setting up the Database Cleaner gem's strategy to be truncation, so that the tables are all truncated at the end of each test, rather than having the tests run inside a transaction. The `pre_count` option will perform counts on the tables database cleaner wants to truncate, and if there's any records in them will truncate them, leaving the empty tables alone. The `reset_ids` option will reset the auto-increment count on each of the tables.

We'll need to also require `database_cleaner` in this file also. Put this line underneath the require for `factory_girl`, near the top of the file:

```
require "database_cleaner"
```

With this code, the `database_cleaner` gem will clean out all the tables in the `public` schema after each test, leaving it at a nice clean slate.

Now, for the schemas, it's going to be a bit more messier than that. Unfortunately, there isn't a query for PostgreSQL to tell it to nuke all the schemas in the database, and so we must improvise. Inside the `config.after(:each)` block in `spec/rails_helper.rb` put this:

```
connection = ActiveRecord::Base.connection.raw_connection
schemas = connection.query(%Q{
  SELECT 'drop schema ' || nsname || ' cascade;'
  from pg_namespaces
  where nsname != 'public'
  AND nsname NOT LIKE 'pg_%'
  AND nsname != 'information_schema';
})
schemas.each do |query|
  connection.query(query.values.first)
end
```

In this code, we go right down to the raw connection of the database and issue a query against the connection. This query selects all the schema names from the `pg_namespace` table where the name is not “public”, “information_schema” or anything beginning with “pg_”. It then wraps these names in `drop schema [name] cascade;` query, and returns a list of those queries for us to execute, which we do. This will clean all the schemas from the database, dropping all the tables inside the schemas first, before dropping the schemas themselves.

When we run the test again, it will pass:

```
2 examples, 0 failures
```

Excellent! Our schema scoping is working perfectly now. Whenever a user visits an account’s subdomain, they’ll only be able to see the data within that account’s schema.

Let’s find out what happens when we run all the tests with `rspec spec` now:

```
26 examples, 9 failures, 10 pending
Failed examples:
rspec ./spec/features/accounts/sign_up_spec.rb:52
rspec ./spec/features/accounts/sign_up_spec.rb:20
rspec ./spec/features/accounts/sign_up_spec.rb:4
rspec ./spec/features/accounts/sign_up_spec.rb:37
rspec ./spec/features/users/sign_up_spec.rb:7
rspec ./spec/features/users/sign_in_spec.rb:42
rspec ./spec/features/users/sign_in_spec.rb:20
rspec ./spec/features/users/sign_in_spec.rb:10
rspec ./spec/features/users/sign_in_spec.rb:31
```

Over half of our tests are failing now, for one reason or another. Let’s walk through fixing up these tests now.

Repairing the broken tests

The first test file that is failing is the one `spec/features/accounts/sign_up_spec.rb`. The errors that the tests in this file are producing are the same:

```
Failure/Error: visit subscribem.root_url
Apartment::SchemaNotFound:
  One of the following schema(s) is invalid: www, "public"
```

This is happening because the default Capybara host is `www.example.com`, and the `Apartment::Elevator::Subdomain` middleware is attempting to find the `www` schema, which doesn’t exist.

To get around this problem, we’ll need to tell Capybara that the application’s host should be just a host without a subdomain. We can do this by putting this line at the end of `spec/rails_helper.rb`:

```
Capybara.app_host = "http://example.com"
```

Requests to the root path for the application now will go to a `faux-http://example.com`, rather than `http://www.example.com`. Running this test again, we’ll see that it works:

2 examples, 0 failures

We'll also need to enforce this change within our `spec/support/subdomain_helpers.rb` too, because that is still using the old "www.example.com" route. Change this line in that file:

```
after { Capybara.default_host = "http://www.example.com" }
```

To this:

```
after { Capybara.default_host = "http://example.com" }
```

As a side-effect of this change, the tests in `spec/features/accounts/sign_up_spec.rb` will also start failing. Make a similar change for all the tests in that file and we will stay out of trouble.

The next group of tests that are failing are the ones within `spec/features/accounts/sign_in_spec.rb`. They're failing for a similar reason to the `sign_up_spec.rb` tests:

```
Failure/Error: visit subscribem.root_url(:subdomain => account.subdomain)
  Apartment::SchemaNotFound:
    One of the following schema(s) is invalid: test1
```

This error is happening because the accounts we're creating for this test aren't creating schemas as well. We can fix this very easily by changing this line at the top of this file:

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

To this:

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

When we run these tests again, they will pass:

3 examples, 0 failures

Boy, this is easy work! What's left? Another run of `rspec spec` will tell us:

```
rspec ./spec/features/users/sign_up_spec.rb:6
```

This test is failing for the same reason as the last two:

```
Failure/Error: visit "http://#{account.subdomain}.example.com/sign_up"
  Apartment::SchemaNotFound:
    One of the following schema(s) is invalid: test9
```

To fix this, we'll replace this line at the top of `spec/features/users/sign_up_spec.rb`:

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

With this line:

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

Running the test again will tell us that it's now passing:

```
1 example, 0 failures
```

Too easy! Now we've got the schema scoping working for our application. We're almost done in this chapter with just one more thing to do.

Tidying up the current account helper

That one more thing is the altering of the `current_account` method and related code within the `Subscribem` engine. Rather than using data within `Warden` to determine what our current account is, we should be relying on the request's subdomain instead. If we didn't change this, then future calls to `current_account` after signing in will always yield the same account, even if we're making a request to a completely different subdomain.

The issue is that the `current_account` from the `ApplicationController` extender (`app/extendes/controller/application_controller_extender.rb`) looks like this:

```
def current_account
  if user_signed_in?
    @current_account ||= begin
      account_id = env["warden"].user(:scope => :account)
      Subscribem::Account.find(account_id)
    end
  end
end

helper_method :current_account
```

We were using this method to do the database scoping for our things in `ThingsController`, but this method isn't needed any more. Yet, it will still be useful to keep it around so that applications using `Subscribem` have an easy way to access the current subdomain's account.

Therefore, we should change this method to this:

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

This way, the method will return the correct `Subscribem::Account` object based on the subdomain, rather than Warden session data which is set when a user signs in.

Speaking of which, now that we're not using that data any more, we should remove the code which is setting it. This code is within `app/controllers/subscribem/application_controller_extender.rb`, and is the final line within the `force_authentication!` method:

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

Let's remove this line now and only take one argument for this method:

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

We'll also need to change where this method is being called to reflect these latest changes. The first place is within `app/controllers/subscribem/accounts_controller.rb`, changing this line:

```
force_authentication!(@account, @account.owner)
```

Into this:

```
force_authentication!(@account.owner)
```

The second place is within `app/controllers/subscribem/account/users_controller.rb`, and is this line:

```
force_authentication!(account, user)
```

Which should now be changed to this:

```
force_authentication!(user)
```

With this light refactoring now complete, let's see what our tests think of it with another easy run of `rspec spec`.

```
12 examples, 0 failures
```

Easy done. We've now changed the `current_account` method to reflect the current account properly, based on the subdomain.

3.3 The PostgreSQL Caveat

In this chapter, we've covered two methods of scoping data in our database: foreign key scoping and schema scoping. While schema scoping is a slightly cleaner implementation than foreign key scoping, there can be problems caused by a large number of schemas, as outlined in the Heroku Devcenter article already mentioned several times within this chapter. ⁶ **This book recommends against using schema scoping for a large SaaS implementation, and recommends instead to use foreign key scoping.**

Feel free to continue using PostgreSQL schemas

For non-large implementations of a multi-tenanted application, continuing to use PostgreSQL schemas with the Apartment gem is fine. You won't have any issues. The caveat listed above is to catch those people who may wish to use Apartment + PostgreSQL as a "cheap" fix for multi-tenancy in a large setting. On large databases, PostgreSQL *may* encounter issues where backups take a very long time to complete. In those cases, scoping by a database field is a much better idea.

In the next section, we'll look at how we can switch from using Apartment to foreign key scoping.

3.4 Scoping by a database field: redux

At this point of our engine development, we have discovered that using PostgreSQL schemas is not as effective as we had hoped. If our PostgreSQL database has a large number of tables in it, that will cause backups to be incredibly slow. A daily backup, for instance, might take longer than a day to complete and that's pretty bad. Database backups are extremely important!

Sometimes when we're developing applications ⁷, things don't always go to plan and that's OK. Luckily, we have developed a robust codebase that will allow us to make the required changes relatively easily.

Within this section, we're going to work on switching away from the schema-based setup we developed at the end of the last chapter, and instead work on building out the foreign key scoping we had at the beginning of that chapter.

By using this foreign-key scoping, we're not going to run into the problem of too many tables causing our backups to take too long. In addition to that, this scoping method will work on all databases and not just PostgreSQL. We're going to stick with the PostgreSQL configuration that we have in `config/database.yml`, since there's no point changing it because there's not going to be any difference.

Cleaning up after ourselves

We've got a lot of code still within our engine that's relying on the schema functionality that we're about to rip out. Therefore, we're going to need to remove this code too. To begin with, the apartment gem won't be necessary any more since we're not going to be using its particular brand of scoping, which is either full-database switching or schema scoping.

⁶<https://devcenter.heroku.com/articles/heroku-postgresql#multiple-schemas>

⁷Or writing books.

Let's start with removing the apartment gem from `subscribem.gemspec` and seeing what breaks when we run our tests using `bundle exec rspec spec:`

```
cannot load such file -- apartment (LoadError)
```

This is happening because we're still requiring the gem within `lib/subscribem/engine.rb`. We need to remove these lines from there:

```
require "apartment"
require "apartment/elevators/subdomain"
```

If we look further down in this file, we'll see that we also have some code referencing the `Apartment` constant:

```
initializer 'subscribem.middleware.apartment' do
  Rails.application.config.middleware.use Apartment::Elevators::Subdomain
  Apartment.excluded_models = ["Subscribem::Account",
                              "Subscribem::Member",
                              "Subscribem::User",
                              "Subscribem::Plan"]
end
```

Remove this code and run the tests again, then we'll be told this:

```
config/initializers/apartment.rb:1:in `<top (required)>':
  uninitialized constant Apartment (NameError)
```

As we can tell from this error message, the `Apartment` constant is still being referenced inside some code for `config/initializers/apartment.rb`. Let's delete this file now, since we are not going to need it, and re-run our tests to see what the next issue may be.

At this point, our tests will actually run, and some of them will fail. The first test that fails is this one:

```
1) User signup under an account
   Failure/Error: let!(:account) { FactoryGirl.create(:account_with_schema) }
   NameError:
     uninitialized constant Subscribem::Account::Apartment
     # ./spec/features/users/sign_up_spec.rb:5
```

Within `spec/features/users/sign_up_spec.rb`, on line 5, we have this line:

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

This method tries to create an account using the `account_with_schema` factory. We will no longer need this factory because we're going to be switching to not using schemas. This factory is defined within `spec/support/factories/account_factory.rb`. Let's remove this code from there now:

```
factory :account_with_schema do
  after(:create) do |account|
    account.create_schema
  end
end
```

Let's change the factory reference in `spec/features/users/sign_up_spec.rb` from this:

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

To this:

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

If we attempt to run this test again, using `bundle exec rspec spec/features/users/sign_up_spec.rb`, we'll see this error now:

```
1) User signup under an account
Failure/Error: Unable to find matching line from backtrace
NameError:
  uninitialized constant Apartment
# /subscriptions/spec/rails_helper.rb:36...
```

The line number may be different in your example, but the file location will be the same. Inside `spec/rails_helper.rb`, there is some more code that references the `Apartment` constant inside a `config.after(:each)` block:

```
config.after(:each) do
  Apartment::Tenant.reset
  DatabaseCleaner.clean
  connection = ActiveRecord::Base.connection.raw_connection
  schemas = connection.query(%Q{
    SELECT 'drop schema ' || nsname || ' cascade;'
    from pg_namespace
    where nsname != 'public'
    AND nsname NOT LIKE 'pg_%'
    AND nsname != 'information_schema';
  })
  schemas.each do |query|
    connection.query(query.values.first)
  end
end
```

This block of code is doing a couple of different things. For starters, it resets `Apartment` to the default schema, cleans the database using the `database_cleaner` gem, and then drops all the schemas that were created during the test using some funky SQL query.

We will need to keep the database cleaning code, but not the apartment resetting or schema dropping code. This means that we can change this code to simply this:

```
config.after(:each) do
  DatabaseCleaner.clean
end
```

Let's re-run this test again. This time it will work:

```
1 examples, 0 failures
```

Sweet, wonderful progress! One of the things we used to get this test working was to change the factory use in the test from `:account_with_schema` to just `:account`. If we run a program such as `ack` to find where the other occurrences of this are in our engine, we'll see them used here:

```
spec/features/accounts/scoping_spec.rb
4: let!(:account_a) { FactoryGirl.create(:account_with_schema) }
5: let!(:account_b) { FactoryGirl.create(:account_with_schema) }
spec/features/users/sign_in_spec.rb
6: let!(:account) { FactoryGirl.create(:account_with_schema) }
```

Let's rename these occurrences to also use `FactoryGirl.create(:account)` instead.

The `account_with_schema` factory called the method `create_schema` on the `Subscribem::Account` model instance. We're not going to need this method any longer either, so let's remove this method from the `Subscribem::Account` model:

```
def create_schema
  Apartment::Tenant.create(subdomain)
end
```

If we use `ack` again, we can see that this method is being used in more than just the factory:

```
app/controllers/subscribem/accounts_controller.rb
14: @account.create_schema
spec/controllers/subscribem/accounts_controller_spec.rb
16: account.should_receive(:create_schema)
spec/models/subscribem/account_spec.rb
17: account.create_schema
```

Let's remove these examples also.

In `app/controllers/subscribem/accounts_controller.rb`, remove this line from inside the `create` action:

```
@account.create_schema
```

For `spec/controllers/subscribem/accounts_controller_spec.rb`, all this test is doing is checking that the `create_schema` method gets called when a new account is created. This means then that we can delete this file completely.

For `spec/models/subscribem/account_spec.rb`, we can remove this test:

```

it "creates a schema" do
  account = Subscribem::Account.create!({
    :name => "First Account",
    :subdomain => "first"
  })
  account.create_schema
  failure_message = "Schema #{account.subdomain} does not exist"
  assert schema_exists?(account), failure_message
end

```

When we re-run all our tests again, we should now only be down to two failing tests:

```

rspec ./spec/features/accounts/scoping_spec.rb:17
rspec ./spec/features/accounts/scoping_spec.rb:24

```

These are the two tests that we want failing right now, as they're the ones testing that our engine's scoping is working correctly. These tests are both failing with the same error message:

```

Failure/Error: Apartment::Tenant.switch(account_a.subdomain)
NameError:
  uninitialized constant Apartment

```

We need to remove all the lines here that are referencing the Apartment constant, since we're no longer going to be using that. With the lines referencing Apartment removed, we are no longer scoping within this test. This means that the two examples in this file will fail when we run it with `bundle exec rspec spec/features/accounts/scoping_spec.rb`:

```

1) Account scoping displays only account A's records
   Failure/Error: page.should_not have_content("Account B's Thing")
     expected not to find text "Account B's Thing"
       in "Things Account A's Thing Account B's Thing"
   # ./spec/features/accounts/scoping_spec.rb:21
2) Account scoping displays only account B's records
   Failure/Error: page.should_not have_content("Account A's Thing")
     expected not to find text "Account A's Thing"
       in "Things Account A's Thing Account B's Thing"
   # ./spec/features/accounts/scoping_spec.rb:28

```

Rather than scoping by a database schema, we'll be scoping this resource using a foreign key in the things table, which will link that table with the subscribem_accounts table. Luckily for us, we still have the Thing model within our dummy app, and the things table still has an account_id and so this will not be too difficult at all.

Using the Houser gem

There's a gem out there that will make our lives a lot easier, and that gem is called Houser.⁸ This gem provides a Rack middleware which extracts the subdomains from the host and attempts to find a record matching that subdomain, which we can then use in our application to scope resources however we feel.

⁸<https://devcenter.heroku.com/articles/heroku-postgresql#multiple-schemas>

While this is slightly more work than the automatic scoping with the Apartment gem and PostgreSQL schemas, with this method we will not run into the problems described at the beginning of this chapter.

Let's install the Houser gem by putting this line into `subscriber.gemspec` now:

```
s.add_dependency "houser", "1.0.2"
```

Then running `bundle install`.

Once we've got the gem installed, we need to set up the middleware inside `lib/subscriber/engine.rb`, which we can do with these lines placed inside the `Subscriber::Engine` class:

```
initializer 'subscriber.middleware.houser' do
  Rails.application.config.middleware.use Houser::Middleware,
    :class_name => 'Subscriber::Account'
end
```

To access that `Houser::Middleware` constant, we're going to need to require Houser, which we can do with this line underneath the require for Warden at the top of this file:

```
require "houser"
```

This new initializer configures the Houser gem's middleware to run during every request to the application that the Subscriber engine is embedded into. We pass it the `:class_name` configuration option, which tells Houser which class we want to scope by. Once a request to a subdomain has passed through Houser, it will return us the relevant `Subscriber::Account` object for that subdomain within the `X-Houser-Object` request header, but only if a `Subscriber::Account` object with that subdomain exists.

With the middleware now set up, we can use it to scope the `Thing` class within Subscriber's dummy application so that the `scoping_spec.rb` tests pass again. Before we can do the scoping, we need to be able to easily access the object that Houser says that we're scoping by. To do that, let's change this code within `app/extendes/controllers/application_controller_extender.rb`:

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

To this:

```
def current_account
  @current_account ||= env['X-Houser-Object']
end
```

Much neater! Houser is doing the finding for us, so we just need to grab what it's found and use that instead of doing the finding ourselves. Now with the real `current_account` object available, we can implement this scoping in our `ThingsController` within the dummy app:

spec/dummy/app/controllers/things_controller.rb

```
1 class ThingsController < ApplicationController
2   def index
3     @things = Thing.scoped_to(current_account)
4   end
5 end
```

In all controllers from this point forward we are going to be doing explicit scoping, rather than the *implicit* scoping that was done using the Apartment gem. By making it explicit at the fetching level, it's clearer when scoping is or is not occurring, which is a key benefit of doing the scoping this way.

The `scoped_to` method on `Thing` isn't yet defined, so we should probably define that if we want any chance of our tests passing. Let's open the `Thing` model and add this method now:

spec/dummy/app/models/thing.rb

```
1 class Thing < ActiveRecord::Base
2   def self.scoped_to(account)
3     where(:account_id => account.id)
4   end
5 end
```

This method is extremely basic, and that's all it needs to be. It returns a scope which limits the records returned to just those with the same account id as the account that the Houser gem found.

Back in our test in `spec/features/scoping_spec.rb`, we're creating new `Thing` objects, but not scoping them by an account:

```
Thing.create(:name => "Account A's Thing")
Thing.create(:name => "Account B's Thing")
```

With no scoping here, the `Thing` objects will be lost in the void and will not be shown at all when a request is made to either Account A's or Account B's subdomains. We can see that in practice if we run the tests at this point:

```
1) Account scoping displays only account A's records
   Failure/Error: page.should have_content("Account A's Thing")
     expected to find text "Account A's Thing" in "Things"
   # ./spec/features/accounts/scoping_spec.rb:20
2) Account scoping displays only account B's records
   Failure/Error: page.should have_content("Account B's Thing")
     expected to find text "Account B's Thing" in "Things"
   # ./spec/features/accounts/scoping_spec.rb:27
```

To fix this, we simply need to create the `Thing` objects within their correct scopes. To do this, we can change the `Thing.create` lines to this:

```
Thing.scoped_to(account_a).create(:name => "Account A's Thing")
Thing.scoped_to(account_b).create(:name => "Account B's Thing")
```

With this `scoped_to` call now in all the right places, along with the Houser gem, we will see our scoping tests passing once again:

```
2 examples, 0 failures
```

That's great! We've been able to pretty painlessly move over from the Apartment gem to the Houser gem; using foreign key scoping in our engine in order to retrieve only the records for a specific account.

3.5 Summary

In this chapter, we showed the benefits of foreign key scoping as well as schema scoping. Both have their pros and cons.

With PostgreSQL scoping, the scoping happens *implicitly* – by the Apartment gem in conjunction with PostgreSQL's schema search path – and so there is no need for putting scoping calls everywhere in the application. However, the downside to this is that it will only work on small applications with a small number of tables. Applications with a large amount of total tables may have problems running backups.

With foreign key scoping, you need to make explicit calls to scope your application's resources wherever that is happening. Missing even a single place can lead to your application not behaving as intended. We will be looking more at this explicit scoping and how to deal with it in the next chapter.

4. The blog application

It's time to link together the work that we've done in the last couple of chapters and the Blorgh application. Blorgh is a very simple blogging application, available on GitHub at <https://github.com/radar/blorgh>. All this application has in it is the ability to perform the usual CRUD actions on posts, as well as the ability for users to leave any comment they wish on the posts.

To keep things simple for this chapter, we'll be using this application. It's going to be easy to add multitenancy to this application because of how small it is. After the end of this chapter, we can apply the learnings to bigger apps.

Let's get stuck into making Blorgh a proper multitenanted application.

4.1 Setting up Blorgh

To begin, we'll clone the Blorgh app down into the same directory where `subscribem` is based:

```
git clone git@github.com:radar/blorgh
```

What we have here is just a basic Rails application. To add multitenancy features to this app, we'll need to add in the `Subscribem` engine. We can do this by adding the engine to this application's `Gemfile`:

```
gem 'subscribem', path: '../subscribem'
```

We'll then need to mount the `Subscribem` engine within our application to make its features available. We can do this by putting this line at the bottom of the routes definition within `config/routes.rb`:

```
mount Subscribem::Engine, :at => '/'
```

This will make things such as account sign up available at the root of the application. We're putting this at the end so that the routes for our application are matched first, then the routes for `Subscribem`. All application routes should take precedence over any engine routes within a Rails application.

In order to make Blorgh properly multitenanted, we're going to place all the other routes underneath the subdomain constraint. This way, we'll have a clear separation of posts between different accounts. For instance, Account A will have some set of posts, and Account B will have other posts.

To enforce this separation, let's wrap all the routes – but not the mount definition – in our `SubdomainRequired` constraint:


```
constraints(Subscribem::Constraints::SubdomainRequired) do
  root :to => "posts#index"
  ...
end
```

To make this `Subscribem::Constraints::SubdomainRequired` constant available, we'll need to require this at the very top of `config/routes.rb`:

```
require 'subscribem/constraints/subdomain_required'
```

At the bottom of these routes are some routes for authentication. We're not going to use Blorgh's authentication, because we already have that feature provided by the Subscribem engine. Therefore we'll remove these routes from `config/routes.rb` now:

```
get '/sign_in' => 'sessions#new'
post '/sign_in' => 'sessions#create'
delete '/sign_out' => 'sessions#destroy'
```

We can also remove the related controller, `app/controllers/sessions_controller.rb`, as well as the related views `app/views/sessions`. There's also some code that displays the "Sign in" link and friends in `layouts/application.html.erb` which needs removing too:

```
<% if signed_in? %>
  You are signed in as <%= current_user.email %> &middot;
  <%= link_to 'Sign out', sign_out_path, method: 'delete' %>
<% else %>
  <%= link_to "Sign in", sign_in_path %>
<% end %>
```

The Blorgh app also contains some Warden configuration which would override the configuration that's within the Subscribem engine. This configuration is within `config/application.rb`:

```
config.middleware.use Warden::Manager do |manager|
  manager.default_strategies :password
  manager.serialize_into_session do |user|
    user.id
  end
  manager.serialize_from_session do |id|
    User.find(id)
  end
end
```

If this configuration was still there, Blorgh would use its own `User` model to attempt to do the authentication, rather than Subscribem's `Subscribem::User` model. Let's remove this entire block of code now.

Finally, we'll need to install the migrations from the Subscribem engine. Without these, any attempts to use the Subscribem engine's models will fail dramatically within the application. We can install these using these commands:

```
bundle exec rake railties:install:migrations
bundle exec rake db:migrate
```

Our next step is to test Subscribem’s integration and make sure that it works with our application.

4.2 Testing Subscribem Integration

Within the Blorgh application, we want to first make sure that people can sign up for accounts. Without this ability, people will not be able to do much else with our application! After that, we’ll want to make sure that once they have created an account they can sign into it.

To test these things, we’ll write some feature specs that walk through the flows of creating an account, as well as signing in to that account. Before we can write those, we’ll need to install the `capybara` and `rspec-rails` gems into this application. We can do that by adding these lines to the `Gemfile`:

```
group :development, :test do
  gem 'rspec-rails', '3.0.1'
  gem 'capybara', '2.3.0'
end
```

Then we need to run `bundle install` to make sure that we have these gems installed, just in case we don’t already from earlier. Next, we’ll need to set up the spec directory:

```
rails g rspec:install
```

With RSpec setup, we can begin writing some tests for our app.

Testing account sign up

Now we can get to writing a feature to test that account sign up works for our Blorgh app. Let’s create a new file within `spec/features/accounts/sign_up_spec.rb`. We’re going to get a lot of “inspiration” from the feature by the same name from the Subscribem engine, and by “inspiration” I mean we’re going to blatantly copy and paste the code from there to here:

`spec/features/accounts/sign_up_spec.rb`

```
1 require "rails_helper"
2 feature "Accounts" do
3   scenario "creating an account" do
4     visit "/"
5     click_link "Account Sign Up"
6     fill_in "Name", :with => "Test"
7     fill_in "Subdomain", :with => "test"
8     fill_in "Email", :with => "subscribem@example.com"
9     password_field_id = "account_owner_attributes_password"
10    fill_in password_field_id, :with => "password"
11    fill_in "Password confirmation", :with => "password"
12    click_button "Create Account"
```

```

13     success_message = "Your account has been successfully created."
14     expect(page).to have_content(success_message)
15     expect(page).to have_content("Signed in as subscribem@example.com")
16     expect(page.current_url).to eq("http://test.example.com/")
17   end
18 end

```

We've made one modification to this test: the `visit` call right at the top. Instead of visiting Subscribem's root, we want to visit the actual root of our application. It's from here that users will go to sign up for an account.

When we run this test using `bundle exec rspec spec/features/accounts/sign_up_spec.rb`, it will fail like this:

```

Failure/Error: page.should have_content(success_message)
  expected to find text "Your account has been successfully created."
    in "My blog Read all about it!"

```

We saw this error way back in Chapter 2, and the cause of it was the `session_store` of Subscribem's dummy application not being setup to correctly handle the case where we want to share sessions between a domain and its subdomains. We need to apply the same fixes here. We can do that by opening `config/initializers/session_store.rb` within the Blorgh app, and changing this line:

```

Blorgh::Application.config.session_store :cookie_store,
  key: '_blorgh_session'

```

To these lines:

```

if Rails.env.test?
  Blorgh::Application.config.session_store :cookie_store,
    key: '_blorgh_session',
    domain: 'example.com'
else
  Blorgh::Application.config.session_store :cookie_store,
    key: '_blorgh_session',
    domain: 'blorghapp.com'
end

```

Let's run this test again and see if that does indeed fix the problem.

```

Failure/Error: click_button "Sign in"
ActionView::Template::Error:
  undefined method `admin?' for ...
  # ./app/controllers/application_controller.rb:17:in `admin?'
  # ./app/views/posts/index.html.erb:1: ...

```

One more dastardly error! Blorgh seemingly has a method called `admin?` which is being called from `posts/index.html.erb`:

```

1  <% if admin? %>
2    <%= link_to 'New Post', new_post_path, class: 'btn btn-primary' %>
3  <% end %>

```

Blorgh calls the `admin?` method here, and as we can see from the stacktrace, this method is defined within Blorgh's `ApplicationController`:

```

def admin?
  current_user && current_user.admin?
end
helper_method :admin?

```

This code was suitable when Blorgh was its own standalone application. Now that we have `Subscribem` in the mix, it's interfering with this particular part of Blorgh's code. Blorgh's own `User` model has an attribute called `admin`, but `Subscribem::User` does not have this attribute, and so we're getting this error.

We can get around this by making one clear assumption: a user is an admin for Blorgh if they are the owner of the current account within `Subscribem`. Let's change the `admin?` method to do this check now:

```

def admin?
  current_user && current_account.owner == current_user
end
helper_method :admin?

```

We're checking here that if there is a `current_user` and if the current account's owner is that user, then that user is an admin. This should make our test happier now.

```

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

```

Happier? Yes. Complete? No. This test is unable to find the "Signed in as..." text on the page, and with very good reason. Over in the `subscribem` engine, this text is located within `app/views/layouts/subscribem/application.html.erb`. This layout is only used by the `Subscribem` engine. Our application uses its own layout at `app/views/layouts/application.html.erb`. What we need to do is to create a way to share this information between the two templates and the best way to do that would be to create a partial.

Let's create a new partial now within the `Subscribem` engine:

`app/views/subscribem/shared/_login.html.erb`

```

1  <% if user_signed_in? %>
2    Signed in as <%= current_user.email %>
3  <% end %>

```

We can use this partial within `app/views/layouts/subscribem/application.html.erb` in place of that content:

```
<%= render "subscribem/shared/login" %>
```

And we can do the same thing in our application, replacing the sign in information we ripped out at the beginning of adding Subscribem to Blorgh:

```
<div class='row'>
  <div class='pull-right'>
    <%= render "subscribem/shared/login" %>
  </div>
</div>
```

When we run our test again, this time it will be completely happy:

```
1 example, 0 failures
```

One down! Now let's test that sign in works for our accounts.

4.3 Testing user sign in

For users to be able to create posts for their account, they're going to need to be able to sign in. The way a user will sign in is that they will visit their account's subdomain, such as myblog.blorghapp.com, and then click 'Sign in'. Therefore, let's make sure that this works by creating a new spec within spec/features/users/sign_in_spec.rb:

spec/features/users/sign_in_spec.rb

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

This spec is also a direct copy of one within the Subscribem engine. With this spec, we're making sure that when we go to the root page, we should be able to click a link to begin the process of signing in, actually sign in and then have that all work.

Let's try running this spec now with `bundle exec rspec spec/features/users/sign_in_spec.rb`. When we do, we'll see this error:

```
...spec/features/users/sign_in_spec.rb:4:in `block in <top (required)>':
  uninitialized constant SubdomainHelpers (NameError)
```

This `SubdomainHelpers` constant is currently only available within the `Subscribem` engine. The constant is defined within `spec/support/subdomain_helpers.rb`, and due to it being defined in that location, it is completely unavailable from the `Blorgh` app. The only place that `Blorgh` can load code from `Subscribem` is the `lib` directory, because that is the pre-defined load path of the gem. Therefore, if we want to use `SubdomainHelpers`, we will need to move it from `spec/support` to within the `lib` directory for `subscribem`.

Sharing SubdomainHelpers

The place where we'll move this file to is `lib/subscribem/testing_support/subdomain_helpers.rb`. Because we've now got this file under that path and also because we don't want pollute the global namespace, we'll change the code in this file to this:

```
module Subscribem
  module TestingSupport
    module SubdomainHelpers
      def within_account_subdomain(&block)
        context "within a subdomain" do
          let(:subdomain_url) { "http://#{account.subdomain}.example.com" }
          before { Capybara.default_host = subdomain_url }
          after { Capybara.default_host = "http://example.com" }
          yield
        end
      end
    end
  end
end
```

In order for our tests over in the `Blorgh` app to pass, we're going to need to make some changes. The first is that we're going to need to require this file from `Subscribem`, which we can do by adding this line to the top of the `spec/features/users/sign_in_spec.rb`:

```
require 'subscribem/testing_support/subdomain_helpers'
```

Because of the code changes the `subdomain_helpers.rb` file, we'll need to change the constant that we're referencing. Let's change this line:

```
extend SubdomainHelpers
```

To this:

```
extend Subscribem::TestingSupport::SubdomainHelpers
```

We'll also need to make this change within `Subscribem's spec/features/users/sign_in_spec.rb` so that it doesn't break when it attempts to reference the old `SubdomainHelpers` constant.

When we run this test within `Blorgh` again with `bundle exec rspec spec/features/users/sign_in_spec.rb`, we'll see that it's gotten past all the errors to do with `SubdomainHelpers` and is now complaining about another constant:

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

The `FactoryGirl` constant is not defined yet, and that's because our tests for Blorgh do not load the `factory_girl` gem at all.

Sharing Factories

In order for us to be able to load the `factory_girl` gem, we'll need to add `factory_girl` as a dependency in Blorgh's Gemfile:

```
group :development, :test do
  gem "rspec-rails", "2.14.2"
  gem "capybara", "2.3.0"
  gem "factory_girl", "4.4.0"
end
```

We'll need to require `factory_girl` within the `rails_helper` file for Blorgh too:

```
require "factory_girl"
```

This will now be enough to get rid of the complaints about the `FactoryGirl` constant not being available, and so let's run our test again to see how it's progressing. We'll now see this:

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

This spec, elegantly borrowed from the `Subscribem` engine, is referring to a factory called `account`, but that factory is not available within Blorgh. It is available within `Subscribem`, however, as it is defined within `spec/support/factories/account_factory.rb`. As before, we're going to need to move these factories to a shareable location. Since we already have the `lib/subscribem/testing_support` directory within the `Subscribem` engine, this would be a perfectly logical place to move the factories to.

Let's move `spec/support/factories/` to `lib/subscribem/testing_support/factories`. We won't need to change the code in these files, as they're using the DSL that the `factory_girl` gem provides. What we will need to do, however, is to require the `account` factory so that we can use it within our test. We can do that with another line placed at the top of the test.

```
require "subscribem/testing_support/factories/account_factory"
```

This is another change that we'll need to make in `Subscribem` as well. At the top of `spec/features/user-sign_in_spec.rb`, put that line as well. Given that the `account` factory depends on the `user` factory, by way of these lines:

```
association :owner, :factory => :user
after(:create) do |account|
  account.users << account.owner
end
```

We should probably require the user factory at the top of the `lib/subscribem/testing_support/factories/account_factory.rb` file:

```
require "subscribem/testing_support/factories/user_factory"
```

If we didn't do that, then we would need to require both files whenever we wanted to create accounts.

This change should now be enough to get the test happy about the `FactoryGirl` constant and the account factory. Let's run our test in Blorgh again and see:

```
Failure/Error: click_link 'Sign in'
Capybara::ElementNotFound:
  Unable to find link "Sign in"
```

The test is still not completely happy. This time it's complaining because it can't find the "Sign in" link when it visits the page. Why is that? Well, we can get a hint of why when we look in `log/test.log` and see the request that's been made:

```
Started GET "/" for 127.0.0.1 at <timestamp>
...
Processing by PostsController#index as HTML
  Post Load (0.1ms)  SELECT "posts".* FROM "posts"
  Rendered posts/index.html.erb within layouts/application (16.6ms)
Completed 200 OK in 77ms (Views: 75.6ms | ActiveRecord: 0.1ms)
```

The request here is coming in and hitting the `PostsController`, which belongs to the Blorgh app. The "sign in" link that *should* be appearing belongs to Subscribem. In order for this link to appear, we're going to need to add it to the Blorgh app. A perfectly good place for this would be within the `app/views/subscribem/shared/_login.html.erb` file, which we created just before within the Subscribem engine. Let's add this content now:

```
<% if user_signed_in? %>
  Signed in as <%= current_user.email %>
<% else %>
  <%= link_to 'Sign in', subscribem.sign_in_path %>
<% end %>
```

We're linking to an action within the Subscribem engine here by referencing the `subscribem` routing proxy, which is available in the controllers and views of both the Blorgh application and the Subscribem engine. The `user_sign_in_path` takes us to `/sessions/new` within the Subscribem engine and should bring up the sign in page.

All of that means that our test should now be happy. Let's find out with another run of `bundle exec rspec spec/features/users/sign_in_spec.rb`:

1 example, 0 failures

It's made it very happy indeed!

We started off by copying over the sign in spec from Subscribem, but from there on out it wasn't smooth sailing. We needed to make some modifications to make the SubdomainHelpers module from Subscribem available to the Blorgh engine. Then we needed to make some similar modifications to get the factories from Subscribem available in Blorgh. Finally, we ran into an issue where Blorgh was still assuming that it was operating with its own User model, but we fixed that by changing the admin? method definition.

We should also make sure that users can sign up.

4.4 Testing user sign up

Just like what we did for the sign in tests, we can copy over the sign up test from Subscribem into Blorgh, with some minor changes:

spec/features/users/sign_up_spec.rb

```

1  require "rails_helper"
2  require "subscribem/testing_support/factories/account_factory"
3  feature "User signup" do
4    let!(:account) { FactoryGirl.create(:account) }
5    let(:root_url) { "http://#{account.subdomain}.example.com/" }
6    scenario "under an account" do
7      visit root_url
8      click_link "Sign in"
9      click_link "New User?"
10     fill_in "Email", :with => "user@example.com"
11     fill_in "Password", :with => "password"
12     fill_in "Password confirmation", :with => "password"
13     click_button "Sign up"
14     expect(page).to have_content("You have signed up successfully.")
15     expect(page.current_url).to eq(root_url)
16     expect(page).to have_content("Signed in as user@example.com")
17   end
18 end

```

Instead of having users automatically redirected to the sign in part of our application, we're allowing them access to the root. This is because we want anonymous users to browse the posts in this application¹. When they click "Sign in", that's when they're provided the ability to sign up.

When they go through that process, they should be redirected back to the root of the application and signed in.

When we run this test using `bundle exec rspec spec/features/users/sign_up_spec.rb`, we'll see this:

¹This gives them clean access to leave insightful comments that move society forward, as internet commentators are renowned for.

1 example, 0 failures

This test passes right off the bat because we have got everything we need setup. This one is free.

4.5 Testing Blorgh integration

Now we get to the main part of this chapter: making sure that the scoping features provided by Subscribem are actively working on our Blorgh application. We're going to apply the same techniques that we saw at the end of Chapter 3 to make this work here.

To start with, we'll copy over the scoping spec from Subscribem and adjust it to use the posts in Blorgh, rather than the "things" from Subscribem's dummy app. We'll also remove the authentication parts of this test, since we want users to be able to view our posts without having to first authenticate.² Let's do this now and place it into `spec/features/accounts/scoping_spec.rb`:

```
require 'rails_helper'
feature "Account scoping" do
  let!(:account_a) { FactoryGirl.create(:account) }
  let!(:account_b) { FactoryGirl.create(:account) }
  before do
    Post.scoped_to(account_a).create(:title => "Account A's Post")
    Post.scoped_to(account_b).create(:title => "Account B's Post")
  end
  scenario "displays only account A's records" do
    visit posts_url(:subdomain => account_a.subdomain)
    expect(page).to have_content("Account A's Post")
    expect(page).to_not have_content("Account B's Post")
  end
  scenario "displays only account B's records" do
    visit posts_url(:subdomain => account_b.subdomain)
    expect(page).to have_content("Account B's Post")
    expect(page).to_not have_content("Account A's Post")
  end
end
```

When we attempt to run this spec using `bundle exec rspec spec/features/accounts/scoping_spec.rb`, we'll see this error:

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

This error is happening because we're not requiring the file that defines the account factory yet. This account factory is used within our spec to create some accounts that will be the scope for the posts within the test. Let's now add the `require` call for the factory at the top of this spec:

²It's just not *nice* to make people sign in before having to view anything. Here's looking at you, Quora and friends.

```
require 'subscribem/testing_support/factories/account_factory'
```

We should also put this same line at the top of `spec/features/accounts/scoping_spec.rb` within `Subscribem`.

Let's run this test again. This time we'll see this error:

```
Failure/Error: Post.scoped_to(account_a).create(:name => "Account A's Post")
NoMethodError:
  undefined method `scoped_to' for #<Class:...>
```

Our `Post` model is missing the method that can scope its results to a specific account. Rather than just copying this method over from `Subscribem`'s dummy app, let's be smarter about this and move the method into a module:

`lib/subscribem/scoped_to.rb`

```
1 module Subscribem
2   module ScopedTo
3     def scoped_to(account)
4       where(:account_id => account.id)
5     end
6   end
7 end
```

To test this module out, we'll go to `Subscribem` and replace these lines in `spec/dummy/app/models/thing.rb`:

```
def self.scoped_to(account)
  where(:account_id => account.id)
end
```

With this line:

```
extend Subscribem::ScopedTo
```

Since we want `Subscribem::ScopedTo` generally available, we'll put a `require` call for it in `lib/subscribem.rb`:

```
require "subscribem/scoped_to"
```

Then, we can go ahead and run `bundle exec rspec spec/features/accounts/scoping_spec.rb` within `Subscribem`. If we've set up everything correctly, this test should still pass; just like the last time we ran it:

```
2 examples, 0 failures
```

Great! Now let's switch back to `Blorgh` and get the test there passing. We need to define the `scoped_to` method within the `Post` model within `Blorgh`, so let's extend that class with this new `Subscribem::ScopedTo` module:

```
class Post < ActiveRecord::Base
  extend Subscribers::ScopedTo
  has_many :comments
end
```

When we run our test one more time, we'll see it's now failing like this:

```
Failure/Error: page.should_not have_content("Account B's Post")
  expected not to find text "Account B's Post" in
  "... Account A's Post Account B's Post"
```

This is a great indicator that although the test is setting up scoped objects, the real application code is not scoping the posts to the correct accounts. This scoping should take place within the area which is collating all the posts, and that would be the index action within PostsController, which currently looks like this:

```
def index
  @posts = Post.all
end
```

We'll need to change this line within the index method to use the `scoped_to` method:

```
def index
  @posts = Post.scoped_to(current_account)
end
```

When we run our test again, we'll see this error:

```
Failure/Error: visit posts_url(:subdomain => account_a.subdomain)
ActionView::Template::Error:
  SQLite3::SQLException: no such column: posts.account_id:
  SELECT "posts".* FROM "posts" WHERE "posts"."account_id" = 1
```

It would seem that we're missing a column in the database. More specifically, we're missing the `account_id` column in the posts table. Let's add this in now with a new migration:

```
rails g migration add_account_id_to_posts account_id:integer
```

This command will create a new migration which will add the `account_id` column to the posts table. We'll need to add an index as well if we want to make our lookups fast. Let's open this new migration and add some code to tell the migration to create an index:

```
class AddAccountIdToPosts < ActiveRecord::Migration
  def change
    add_column :posts, :account_id, :integer
    add_index :posts, :account_id
  end
end
```

To run this migration on the database, we'll run the usual command:

```
rake db:migrate
```

Running our test now will result in happiness:

```
2 examples, 0 failures
```

Our scoping test is now working for the index action in PostsController. We should look around our application and see if anything else needs scoping. If we look in the PostsController underneath the index action, we can see that the show action is unscoped, which could potentially lead to posts “leaking” across accounts. In other places, the Admin::PostsController also operates unscoped, and so we will need to fix that too.

Scoping the show action

The show action within PostsController clearly doesn't use our explicit style of scoping. This is a great example of a very simple bug within our multi-tenant application. By not having the posts explicitly scoped to the current account, *all* accounts have access to *all* posts. In order to fix this problem, we will first write a spec that duplicates the bug.

Let's add new specs to the spec/features/accounts/scoping_spec.rb group:

```
scenario "Account A's post is visible on Account A's subdomain" do
  account_a_post = Post.scoped_to(account_a).first
  visit post_url(account_a_post, :subdomain => account_a.subdomain)
  expect(page).to have_content("Account A's Post")
end

scenario "Account A's post is invisible on Account B's subdomain" do
  account_a_post = Post.scoped_to(account_a).first
  expect do
    visit post_url(account_a_post, :subdomain => account_b.subdomain)
  end.to raise_error(ActiveRecord::RecordNotFound)
end
```

In the first scenario, we're grabbing the only post for Account A and validating that when we visit the URL for that post under Account A's subdomain, that the post is visible. In the second scenario, we're doing the same thing but for Account B's subdomain. That little difference should result in something completely different – an ActiveRecord::RecordNotFound exception being thrown.

When we run our tests using `bundle exec rspec spec/features/accounts/scoping_spec.rb`, we'll see this:

Failures:

```
1) Account scoping Account A's post is invisible on Account B's subdomain
   Failure/Error: expect do
     expected ActiveRecord::RecordNotFound but nothing was raised
   # ./spec/features/accounts/scoping_spec.rb:35
```

This is great, as it's showing us the bug and it means that we now have a failing test that we can correct. Let's fix this test now by changing this code in `PostsController`:

```
def show
  @post = Post.find(params[:id])
end
```

To this:

```
def show
  @post = Post.scoped_to(current_account).find(params[:id])
end
```

When we run the test again, we'll see that it passes:

```
4 examples, 0 failures
```

Good! Now this entire controller is scoped correctly.

4.6 Summary

While we could go through the rest of the Blorgh application and scope each controller action individually, the method is the same for each action: write a test to prove a fault and then implement the proper scoping. With a great testing framework in place for our application that ensures that things are scoped correctly, nothing should go wrong. Implementing the rest of the scoping for the other actions within Blorgh is an exercise best left to the reader.

In this chapter we took the Blorgh app and added Subscribem to it in order to give Blorgh multi-tenanted features. We saw these in action when we limited the `PostsController` to only showing posts from the current account, rather than all posts within the subdomain.

In the next chapter, we're going to be adding those subscriptions we talked about at the very beginning of the book.

5. Subscriptions

We're now going to be adding Subscriptions to the subscribem engine so that we can actually *subscribem them*. "Them" being accounts. For a certain fee every month, an account can be subscribed to our application. This subscription will give them access to additional features in the application, like being able to create more forums.

GitHub does something similar to this as well: allowing a user to only create a certain number of private repositories for each level of plan. Public repositories are free. As an example of this, for \$7 a month, you can create 5 private repositories on GitHub. This is part of GitHub's business model.¹

In Subscribem, users will live on the "free" plan. If they want additional features, then they will need to upgrade to one of the paid plans, which they'll do by editing their account information. By default, we will have two "named" plans – the "Starter" plan which allows 5 forums and the "Extreme" plan allowing for unlimited forums – as well as the option to not select any plan, which means the account is on the "Free" plan.

To manage the heavy lifting of managing payments for these subscriptions, we're going to be using Braintree's wonderful Ruby API². By letting Braintree deal with the subscriptions handling, we can get on with developing other feature of our application. Braintree will handle all the logic to do with subscriptions such as charging subscribers per month, so that we don't have to.

The plans for our application will come from our Braintree account (for reasons explained later), and so they'll need to be created there first. Once they exist on Braintree, we'll need a way to update our application with that data, so that the plans can be displayed to a user when they're signing up for a new account. To store these plans, we'll create a `Subscribem::Plan` model within the subscribem engine.

So, in summary, this first part involves creating plans which can be subscribed to by accounts. First, we'll need to sign up to Braintree. Then, we'll need to create some plans on Braintree. Finally, we'll need to create a model locally and then a Rake task to store the plan data locally, so it can be displayed to the users when they sign up.

Once we've done those things, we'll then work on building the interface so that an account owner can select their plan and subscribe to our engine. We're *finally* getting to the subscriptions part of our engine. Exciting!

Let's get to it.

5.1 Formulating plans

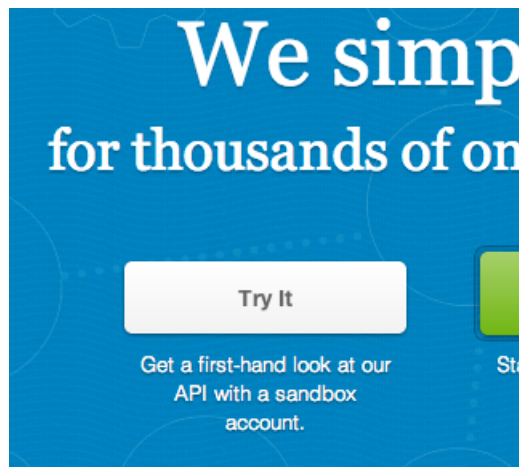
In this section, we'll be signing up to Braintree's sandbox and then we'll create a plan on the sandbox. Once we're done with that, we will write a Rake task to fetch all the plans and store them in our local database. In the next section, we'll take those plans and show them to the user so that they can select a plan for their account.

¹Another part of GitHub's business model is their enterprise version: <https://enterprise.github.com/>

²<https://www.braintreepayments.com/docs/ruby>

Braintree Sign Up

Our first port of call is to sign up to Braintree's sandbox. Go <http://braintreepayments.com> and hit the "Try it" button.



Try it

This will ask us for login information:

Test everything Braintree

Our sandbox environment lets you try our API

Have an account? [Sign in](#)

Name

Company Name

E-mail

Login information

Fill out this form and then hit the "Go!" button. This will send off an email asking you to activate your Braintree Sandbox account.

Thanks!

Thanks for trying Braintree. Next step: check your inbox.

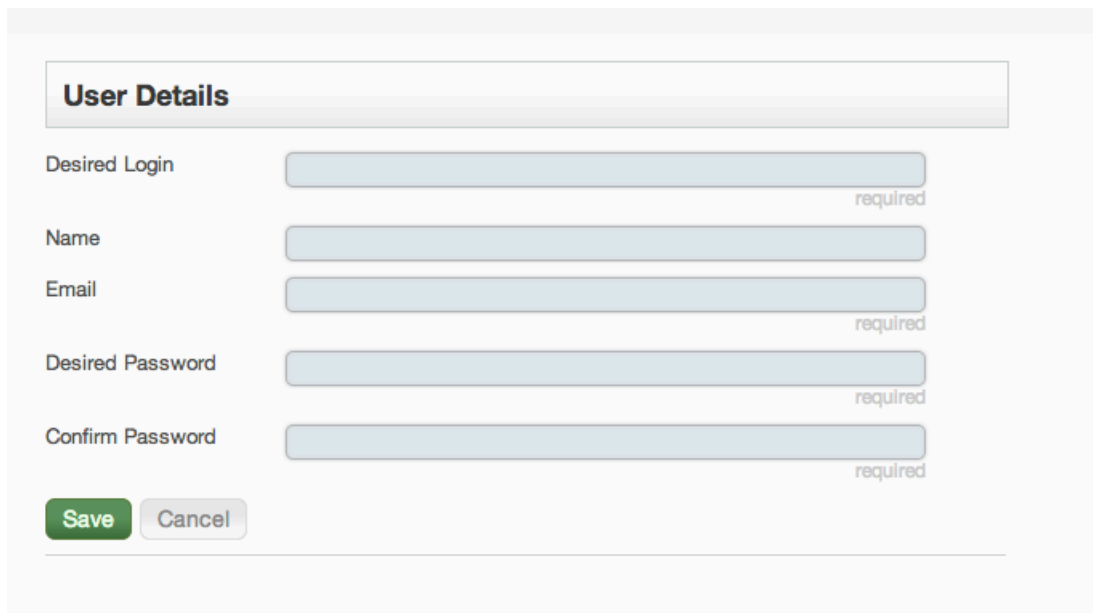
We sent you an email that includes a link to activate your login

You might also be interested in reviewing our [API docs](#).

If you have any questions about logging-in or working with our support@braintreepayments.com

Activation notification

Go to your inbox and click the link that Braintree has sent you. Once that's done, Braintree will ask for some more information:

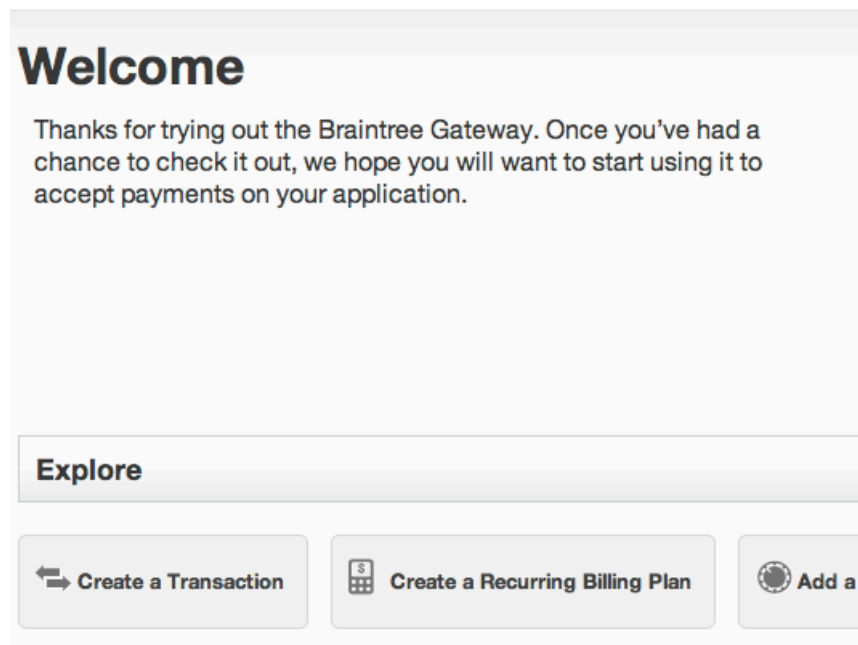


The form is titled "User Details" and contains five input fields, each with a "required" label to its right. The fields are: "Desired Login", "Name", "Email", "Desired Password", and "Confirm Password". At the bottom of the form are two buttons: a green "Save" button and a grey "Cancel" button.

User Details	
Desired Login	<input type="text"/>
Name	<input type="text"/>
Email	<input type="text"/>
Desired Password	<input type="password"/>
Confirm Password	<input type="password"/>
<input type="button" value="Save"/> <input type="button" value="Cancel"/>	

User Signup

Fill in this form too, and remember the login and password used here. Now you'll be signed in to the sandbox, where you can manage the different aspects of your Braintree sandbox, including plans.



Plans

Click the “Create a Recurring Billing Plan” button to begin to create a new plan. The form is in two parts. The first is a form that asks you for the plan ID, a name, a description and a price.

New Plan

Plan Details

Plan ID

A random ID will be generated for you if you leave this blank.

Plan Name

required

Description

Price

required

Currency

USD

Plan form

For the plan ID, leave it blank. For the name, enter “Starter”. Leave the description blank for now. For the price, enter \$9.95.

The second half of the form is to do with billing information for this plan.

Billing Details

Billing Cycle Every

1

 Month(s) required

First Bill Date

☒ Immediately

☐ Specific Day

1st Day of the Month

End Date

☒ Never

☐ After billing cycles

Billing details

The billing information for the plan will tell Braintree how often we want people who are subscribed to this plan to be charged. Since we want them billed every month, just enter “1” into the “Billing Cycle Every” field. Scroll down to the bottom of the page and hit “Create” to create the new plan.

When the plan is created, we'll see some information about it:

Plan Details	
Plan ID	ryn6
Name	bronze
Description	
Price	\$9.95 USD
Trial Period	
Duration	none
Billing Details	
Billing Cycle	Every 1 Month(s)
Billing Start	Immediately
Number of Billing Cycles	Never Expires

Plan details

Braintree has automatically generated an ID for this plan (in the above example it's "ryn6"), which we'll use later as a unique identifier in our application for this plan.

Now that we have a plan within Braintree, the next step is to devise a way to pull these plans from Braintree and store them locally.

Storing plans locally

Before we can store the plans locally, we're going to need to create a model to do that. Within the subscribem engine, run this command:

```
1 rails g model plan name:string price:float braintree_id:string
```

We only care about the name and price for now because that's all we're going to be showing to the user on the plan selection screen. The `braintree_id` field is the unique identifier that is automatically generated by Braintree.

We can run the migration for this model by running this command:

```
1 rake db:migrate
```

Remember: we need to run this within the test environment because otherwise our engine's test database will not be migrated and so the table will not be present when our tests run.

Now we need a way to fetch plans from Braintree and store them in the database. Some people may choose to put the logic for doing this within the `Subscribem::Plan` class itself, but it can lead to crowding of that class. It's best to put this fetching logic outside of the `Subscribem::Plan`, in a separate class, and just call `Subscribem::Plan` methods when we need to.

This new class will be called `BraintreePlanFetcher`, and will live in `lib/subscribem`, because it's a non-application class. This class's job will be to fetch plans from Braintree³, and then store them in our local database.

Before we write the class, we'll write a test for it within `spec/integration/braintree_plan_fetcher_spec.rb`.

`spec/integration/braintree_plan_fetcher_spec.rb`

```

1 require "rails_helper"
2 require "subscribem/braintree_plan_fetcher"
3 describe Subscribem::BraintreePlanFetcher do
4   let(:faux_plan) do
5     double("Plan",
6       :id => "faux1",
7       :name => "Starter",
8       :price => "9.95")
9   end
10  it "fetches and stores plans" do
11    expect(Braintree::Plan).to receive(:all).and_return([faux_plan])
12    expect(Subscribem::Plan).to receive(:create).with({
13      :braintree_id => "faux1",
14      :name => "Starter",
15      :price => "9.95"
16    })
17    Subscribem::BraintreePlanFetcher.store_locally
18  end
19 end

```

This test is simply asserting that when `BraintreePlanFetcher.store_locally` is called, it calls the `Braintree::Plan#all` method. This method comes from the Braintree Rubygem which we will install shortly. This method needs to return an array of plans, which will then be iterated over by `BraintreePlanFetcher.store_locally`, and for each of the plans, `Subscribem::Plan.create` will be called.

At the top of this test it requires a new file at `subscribem/braintree_plan_fetcher`, which doesn't exist right now. Let's create this file at `lib/subscribem/braintree_plan_fetcher.rb` and leave it blank.

If we happen to run the test right now with `bin/rspec spec/integration/braintree_plan_fetcher_spec.rb`, we'll see that it's missing the `BraintreePlanFetcher` constant:

```
... uninitialized constant BraintreePlanFetcher (NameError)
```

Let's define this constant within `lib/subscribem/braintree_plan_fetcher.rb` now. It'll be a class.

³You might have guessed this already from its name.

lib/subscribem/braintree_plan_fetcher.rb

```
1 module Subscribem
2   class BraintreePlanFetcher
3   end
4 end
```

Running the test again will result in the Braintree constant now being the one that's missing.

```
Failure/Error: Braintree::Plan.should_receive(:all).and_return([faux_plan])
NameError:
  uninitialized constant Braintree
```

This constant is going to come from the braintree gem. Let's add this as a dependency to the subscribem engine now by adding this line to subscribem.gemspec:

```
s.add_dependency "braintree", "2.35.0"
```

When we run `bundle install` now, that'll install this gem. Like the warden gem, we will need to require this one into lib/subscribem/engine.rb so that it is loaded when we need it. Let's do that with this line:

```
require "braintree"
```

Running the test again will result in it now not finding the store_locally method:

```
1) Subscribem::BraintreePlanFetcher fetches and stores plans
Failure/Error: Subscribem::BraintreePlanFetcher.store_locally
NoMethodError:
  undefined method `store_locally' for Subscribem::BraintreePlanFetcher:Class
```

This test which is testing the fetching behaviour for the BraintreePlanFetcher class is failing because the star of the show is absent. Without the store_locally method, there will be no fetching.

We can define this method inside lib/subscribem/braintree_plan_fetcher.rb, like this:

```
module Subscribem
  class BraintreePlanFetcher
    def self.store_locally
    end
  end
end
```

The method doesn't need to have any content right now, because all the test is complaining about is that the method is absent. The whole point of TDD/BDD is to let the tests *drive* the design, and that's what we're doing here. The test will say this the next time it's run:

```
Failure/Error: expect(Braintree::Plan).to receive(:all).and_return([faux_plan])
  (<Braintree::Plan (class)>).all(any args)
    expected: 1 time with any arguments
    received: 0 times with any arguments
```

This time, the `Braintree::Plan` class isn't receiving the `all` method, like we're saying it should. This is relatively easy to fix: we just call that method inside our `store_locally` method:

```
class BraintreePlanFetcher
  def self.store_locally
    Braintree::Plan.all
  end
end
```

Again, just some relatively simple code to make the test happier. The test said that it wasn't receiving `Braintree::Plan.all`, and now it will be. Let's see what happens when we run the test again. We'll see this:

```
Failure/Error: Subscribem::Plan.should_receive(:create).with({
  (<Subscribem::Plan(...) (class)>).create({
    :braintree_id=>"faux1",
    :name=>"Starter",
    :price=>"9.95"
  })
  expected: 1 time
  received: 0 times
Failure/Error: expect(Subscribem::Plan).to receive(:create).with({
  (<Subscribem::Plan(...) (class)>).create({
    :braintree_id=>"faux1",
    :name=>"Starter"
    :price=>"9.95"
  })
  expected: 1 time with arguments: ({
    :braintree_id=>"faux1",
    :name=>"Starter",
    :price=>"9.95"
  })
  received: 0 times
```

To fix the test this time, we're going to make the `store_locally` method do what it's supposed to: fetch all the plans and store them. We can do that with this code:

```

module Subscribem
  class BraintreePlanFetcher
    def self.store_locally
      Braintree::Plan.all.each do |plan|
        Subscribem::Plan.create({
          :name => plan.name,
          :price => plan.price,
          :braintree_id => plan.id
        })
      end
    end
  end
end
end
end

```

The next time that we run the test, we'll see that it passes:

```
1 example, 0 failures
```

In the test, we're asserting that the `Braintree::Plan` class should receive the `all` method, and it does. We also assert that the `Subscribem::Plan` method receives a call to `create` with the `name`, `price` and `braintree_id` keys. Because the tests passes, we can be pretty certain that the `BraintreePlanFetcher`'s `store_locally` method is doing what it should be.

The next step is to make sure that this is actually doing something in the real world.

Retrieving the plans

To make sure that `BraintreePlanFetcher` actually really retrieves plans from Braintree, we're going to create a Rake task which uses the `BraintreePlanFetcher` class to fetch the plans from our Braintree account and stores them in the database.

We can define this rake task within the Subscribem engine like this:

`subscribem/lib/tasks/subscribem_tasks.rake`

```

1 require "subscribem/braintree_plan_fetcher"
2 namespace :subscribem do
3   desc "Import plans from Braintree"
4   task :import_plans => :environment do
5     Subscribem::BraintreePlanFetcher.store_locally
6   end
7 end

```

We can then try out this task by going into the blorgh application and running the task with this command:

```
rake subscribem:import_plans
```

When we run this command, it will error out, unlike in the test:


```
rake aborted!
Braintree::Configuration.environment needs to be set
.../subscribem/lib/subscribem/braintree_plan_fetcher.rb:4:in `store_locally'
```



The line it's pointing to is the `Braintree::Plan` line. In our test, we're only asserting that the method is called by stubbing it using `should_receive`. The method isn't actually being called in the test, but it is being called in the Rake task.

We need to configure the Braintree gem to connect to our Braintree sandbox in order to make this Rake task work. It needs the following information:

```
Braintree::Configuration.environment = :sandbox
Braintree::Configuration.merchant_id = "your_merchant_id"
Braintree::Configuration.public_key = "your_public_key"
Braintree::Configuration.private_key = "your_private_key"
```

This information is on the first page you'll see after signing in to the sandbox.

Sandbox API Keys

Merchant ID:	<input type="text" value="rjyzvq7k5"/>	
Public Key:	<input type="text" value="4dhtq8cmjv"/>	
Private Key:	<input type="text" value="d76f2378fa72b6f3a7af797c"/>	

API Sandbox Keys

Copy these keys and create a new initializer at `config/initializers/braintree.rb` and use those API keys:

```
Braintree::Configuration.environment = :sandbox
Braintree::Configuration.merchant_id = "..."
Braintree::Configuration.public_key = "..."
Braintree::Configuration.private_key = "..."
```

Now with this configured, it should be able to connect to Braintree just fine and then attempt to import our plans. However, there's one more thing we've forgotten to do. Running `subscribem:import_plans` again will result in it telling us what that is:

```
[Braintree] [<timestamp>] GET /plans 200
rake aborted!
ActiveRecord::StatementInvalid: Could not find table 'subscribem_plans'
```

At the top of the output for this Rake task, we can now see that it's making a request to `/plans` at Braintree, and that's responding with HTTP status 200, which is good.

The rest of the output shows that we're missing the `subscribem_plans` table, and that's because we've not copied over the migrations from the engine into the application, and then run them. We can do that with two simple commands:

```
bin/rake railties:install:migrations
bin/rake db:migrate
```

The next time we run `rake subscribem:import_plans`, it'll run through correctly:

```
$ rake subscribem:import_plans
I, [timestamp] INFO -- : [Braintree] [timestamp] GET /plans 200
```

If we open up `rails console` now, we can see that the plan has been imported correctly:

```
rails console
Subscribem::Plan.first
=> #<Subscribem::Plan id: 1, name: "Starter" ...>
```

Great. So now we're correctly importing plans from Braintree. There's a problem though. If we re-run this script, it will create another plan with the same name.

```
bin/rake subscribem:import_plan
rails console
Subscribem::Plan.all
=> [#<Subscribem::Plan id: 1, name: "Starter" ...>
    #<Subscribem::Plan id: 2, name: "Starter" ...>]
```

This is because we're not checking for plans with the `braintree_id` before creating them. We need to check for plans that already exist and update them with the information from Braintree if they already exist. For plans that don't exist, we want them to still be created.

Let's update our `BraintreePlanFetcher` class to do this now.

Updating plans

The first thing we want to do is make sure that plans are being checked for existence before they're being created. If they exist, then they should be updated.

Let's put a new test for this inside `spec/integration/braintree_plan_fetcher_spec.rb`:

```
it "checks and updates plans" do
  expect(Braintree::Plan).to receive(:all).and_return([faux_plan])
  expect(Subscribem::Plan).to receive(:find_by).
    with(braintree_id: faux_plan.id).
    and_return(plan = double)
  expect(plan).to receive(:update_attributes).with({
    :name => "Starter",
    :price => "9.95"
  })
  expect(Subscribem::Plan).to_not receive(:create)
  Subscribem::BraintreePlanFetcher.store_locally
end
```

This test makes sure that when `BraintreePlanFetcher.store_locally`, it still calls `Braintree::Plan.all`, but rather than blindly creating the plans it actually checks for them. If `find_by_braintree_id` returns an object, then the code should update the plan's name and pricing information from Braintree. The test also asserts that the We'll test for the opposite – where a plan doesn't already exist – right after this.

When we run this test with `bin/rspec spec/integration/braintree_plan_fetcher.rb`, we'll see it error with this:

```
Failure/Error: Subscribem::BraintreePlanFetcher.store_locally
(⟨Subscribem::Plan(...) (class)⟩).create({
  :name=>"Starter",
  :price=>"9.95",
  :braintree_id=>"faux1"
})
expected: 0 times with any arguments
received: 1 time with arguments: ({
  :name=>"Starter",
  :price=>"9.95",
  :braintree_id=>"faux1"
})
```

This is indicating that `BraintreePlanFetcher.store_locally` isn't checking for a plan first. Let's fix this part of the test by changing the code for `BraintreePlanFetcher` to this:

`lib/subscribem/braintree_plan_fetcher.rb`

```
1 module Subscribem
2   class BraintreePlanFetcher
3     def self.store_locally
4       Braintree::Plan.all.each do |plan|
5         Subscribem::Plan.find_by(braintree_id: plan.id)
6         Subscribem::Plan.create({
7           :name => plan.name,
8           :price => plan.price,
9           :braintree_id => plan.id
10        })
11      end
12    end
13  end
14 end
```

We only need to call the `find_by` method (with the correct arguments) in order to make the test work. When we run the test again, it'll say this:

```
Failure/Error: plan.should_receive(:update_attributes).with({
(Stub).update_attributes({:name=>"Starter", :price=>"9.95"})
  expected: 1 time
  received: 0 times
```

Now we need to move on to fixing the code to do the updating.

lib/subscribem/braintree_plan_fetcher.rb

```
1 module Subscribem
2   class BraintreePlanFetcher
3     def self.store_locally
4       Braintree::Plan.all.each do |plan|
5         if local_plan = Subscribem::Plan.find_by_braintree_id(plan.id)
6           local_plan.update_attributes({
7             :name => plan.name,
8             :price => plan.price
9           })
10        else
11          Subscribem::Plan.create({
12            :name => plan.name,
13            :price => plan.price,
14            :braintree_id => plan.id
15          })
16        end
17      end
18    end
19  end
20 end
```

This code will now attempt to find a plan in the local system, and if it exists then it will update it. If it doesn't, then it will create one. Running the test file again will show that both situations are working:

```
2 examples, 0 failures
```

Now we can make sure that this is working in the real world too. Let's delete all plans within the hosted forums application by running this command:

```
rails runner "Subscribem::Plan.delete_all"
```

To actually verify that this is updating information, let's change the price for the plan to \$9 in Braintree. Log in and do that now.

Run the importer again using the rake task:

```
rake subscribem:import_plans
```

When we go into the console again and find the plan, it'll be the right price:

```
rails c
Subscribem::Plan.all
=> [#<Subscribem::Plan id: 1, name: "Starter", price: 9.00...>
```

Then if we add a new plan to Braintree, say a "Silver" plan that's \$15 and a monthly subscription too, and run the importer again using `rake subscribem:import_plans`, then open the console and find all the plans, we'll now see this new "Silver" plan:

```
rails c
Subscribem::Plan.all
=> [#<Subscribem::Plan id: 1, name: "Starter", price: 9.00...>
    #<Subscribem::Plan id: 2, name: "Silver", price: 15.00...>]
```

Great! This means that the plan importing from Braintree is working. This means that we can now display the plans to a user so that they can select a plan for their account after they've signed up.

5.2 Switching plans

Account owners will be able to choose what plan their account is on by visiting the account edit screen, which will look like this:

TODO: Add plan select screen

The account owner will navigate to this screen from a 'Edit Account' link next to the "Signed in as ..." message that appears at the top of the screen when they log in. When they click this link, they should be able to update their account name or change their plan.

Let's get the basic 'Edit Account' form down first, then we'll add plan selection right after.

Editing account information

The first thing we're going to do is write a feature for updating an account. Let's put this new feature in a new file at spec/features/accounts/updating_spec.rb:

spec/features/accounts/updating_spec.rb

```
1 require "rails_helper"
2 require "subscribem/testing_support/factories/account_factory"
3 require "subscribem/testing_support/authentication_helpers"
4 feature "Accounts" do
5   include Subscribem::TestingSupport::AuthenticationHelpers
6   let(:account) { FactoryGirl.create(:account) }
7   let(:root_url) { "http://#{account.subdomain}.example.com/" }
8   context "as the account owner" do
9     before do
10       sign_in_as(:user => account.owner, :account => account)
11     end
12     scenario "updating an account" do
13       visit root_url
14       click_link "Edit Account"
15       fill_in "Name", :with => "A new name"
16       click_button "Update Account"
17       expect(page).to have_content("Account updated successfully.")
18       expect(account.reload.name).to eq("A new name")
19     end
20   end
21 end
```

Nothing too fancy in this new feature: we sign in as an account owner, click the ‘Edit Account’, set the account’s name to something new, click the update button and then verify that it was reportedly successful and that the name within the database has been updated correctly.

Running this test with `bin/rspec spec/features/accounts/updating_spec.rb` will fail when it attempts to click on the “Edit Account” link:

```
Failure/Error: click_link 'Edit Account'
Capybara::ElementNotFound:
  Unable to find link "Edit Account"
```

Let’s add this link right after the “Signed in as” in Subscribem’s `app/views/subscribem/shared/_login.html.erb`:

```
<% if user_signed_in? %>
  Signed in as <%= current_user.email %>
  <% if current_user == current_account.owner %>
    &middot;
    <%= link_to "Edit Account", account_path %>
  <% end %>
<% end %>
```

In this new code, we will only be displaying the “Edit Account” link to users who are the account’s owner. We don’t want *every* user to be able to edit the account’s details!

We could probably move this logic to the `Subscribem::Account` model so that the check looks neater. Let’s define a new method inside `subscribem/app/models/subscribem/account.rb` called `owner?`, like this:

`subscribem/app/models/subscribem/account.rb`

```
1 def owner?(user)
2   owner == user
3 end
```

Back in the view, we can change this line:

```
<% if current_user == current_account.owner %>
```

To this:

```
<% if current_account.owner?(current_user) %>
```

We will even go one step further, moving this logic into a new helper method inside our `Application-Controller` extender (`app/extenders/controllers/application_controller_extender.rb`), defining it like this:

```
def owner?
  current_account.owner?(current_user)
end
helper_method :owner?
```

This way, the view code will become simply this:

```
<% if owner? %>
```

Additionally, we will be able to use the `owner?` method within our controllers to check to see if the user is an owner, if the need for that ever arises in the future.

Going back to the test at hand now, and the `edit_account_path` helper method isn't defined currently. If we were to run our test again, it would fail with an error like this:

```
Failure/Error: visit root_url
ActionView::Template::Error:
  undefined local variable or method `account_path'
```

We can define this inside `config/routes.rb` by putting this code at the bottom of the scope `:module => "account"` block, which is nested inside the `SubdomainRequired` constraint:

```
constraints(Subscribem::Constraints::SubdomainRequired) do
  scope :module => "account" do
    ...
    get "/account", :to => "accounts#edit", :as => :edit_account
  end
end
```

The reason why we're putting it inside this scope is because the route should only be accessible at an account's subdomain, and since this action is unique to the account scope, it will live under the `Account` namespace, along with other actions like user sign up and user sign in.

Now that we've done everything we know to do, let's find out what the test is telling us again by running it one more time. When we do it, we'll see that the controller is missing:

```
Failure/Error: click_link 'Edit Account'
ActionController::RoutingError:
  uninitialized constant Subscribem::Account::AccountsController
```

Let's create this controller by running this command within the `Subscribem` directory:

```
rails g controller account/accounts
```

With the controller now existing, our test should tell us something different next time we run it.

```
Failure/Error: click_link 'Edit Account'
AbstractController::ActionNotFound:
  The action 'edit' could not be found for
  Subscribem::Account::AccountsController
```

Typically within the `edit` action of a controller, we would create an instance variable which contains the object for the form. We don't need to do this in this action because we already have the object available through the `current_account` helper within `app/extenders/controller/application_controller.rb`.

Therefore, all we need to do for this action is to create the template. Let's do that now in a new file located at `app/views/subscribem/account/accounts/edit.html.erb`. All this form needs to do is provide a name field so that people can update their account's name if they choose, and a submit button that reads "Update Account".

`app/views/subscribem/account/accounts/edit.html.erb`

```
1 <h2>Edit Account</h2>
2 <%= form_for(current_account, :url => account_path) do |account| %>
3   <%= account.error_messages %>
4   <p>
5     <%= account.label :name %>
6     <%= account.text_field :name %>
7   </p>
8   <%= account.submit "Update Account" %>
9 <% end %>
```

We need to specify the `:url` option for the `form_for` here otherwise the URL that will be generated will look like `/accounts/1`, which is unnecessary since the user will only be able to edit their own account. Therefore, the URL will just be `/account` for this form.

Based on that information, the form will attempt to submit a PUT request to `/account`. We don't currently have a route defined for this, so let's define a new one underneath the `get` route that we just defined in `subscribem/config/routes.rb`:

```
patch "/account", :to => "accounts#update"
```

Now when we run the test, it should find the `edit` action, be able to fill in the "Name" field and click the "Update Account" button. If it's able to do all that, then we'll see this output:

```
Failure/Error: click_button "Update Account"
AbstractController::ActionNotFound:
  The action 'update' could not be found
  for Subscribem::Account::AccountsController
```

Alright, now we're getting to the meat of the test: the actual updating of information for the account! All this update action needs to do is to take the form parameters and update the `current_account` object with it. Oh, and return a flash message indicating that the updating was a success. Let's do this with some new code:

subscribem/app/controllers/subscribem/account/accounts_controller.rb

```
1 require_dependency "subscribem/application_controller"
2 module Subscribem
3   class Account::AccountsController < ApplicationController
4     def update
5       if current_account.update_attributes(account_params)
6         flash[:success] = "Account updated successfully."
7         redirect_to root_path
8       end
9     end
10    private
11    def account_params
12      params.require(:account).permit(:name)
13    end
14  end
15 end
```

So far, all the user can update is the name for their account. Therefore this is all we're permitting to be passed through in the `account_params` method definition. We'll permit an attribute for the plan select field a little later on.

We're only accounting for the happy path here so far, since that's all our test is doing. With this new code, our test should pass when we run `bin/rspec spec/features/accounts/updating_spec.rb`:

```
1 example, 0 failures
```

Great! Now we've got a good foundation for the account updating that we will build our plan selection on to. Before we do that, we should account for two "unhappy" paths. The first is where a non-account owner attempts to visit `/account`, they should be redirected away. The second is if an account owner enters a blank name, they should be shown a validation error message.

Then, after that, we'll do plan selection. Promise!

Restricting account editing to owners only

To block normal users of an account from being able to update that account's information, we will need to add in some authorization checking to the `AccountsController`. Before we do that, though, let's add a test to ensure that when they attempt to visit the `edit` link, they'll be redirected away.

code/subscribem/spec/features/accounts/updating_spec.rb

```

1 context "as a user" do
2   before do
3     user = FactoryGirl.create(:user)
4     sign_in_as(:user => user, :account => account)
5   end
6   scenario "cannot edit an account's information" do
7     visit subscribem.edit_account_url(:subdomain => account.subdomain)
8     page.should have_content("You are not allowed to do that.")
9   end
10 end

```

With this test, we sign in as a user and attempt to navigate to the /account page for a subdomain. When we do this *as a user* rather than as an account owner, we should see the message “You are not allowed to do that.”

Running this test with `bin/rspec spec/features/accounts/updating_spec.rb` will result in it still navigating to the edit account form:

```

Failure/Error: page.should have_content("You are not allowed to do that.")
expected there to be text "You are not allowed to do that."
in "Signed in as test2@example.com Edit Account Name"

```

That is evidenced by the “Edit Account” title showing up in the text from the page.

To fix the test, we’ll add a new method to `Subscribem::ApplicationController` called `authorize_owner` which will set the denial message and redirect any non-owner back to the root path:

subscribem/app/controllers/subscribem/application_controller.rb

```

1 def authorize_owner
2   unless owner?
3     flash[:error] = "You are not allowed to do that."
4     redirect_to root_path
5   end
6 end

```

The reason why this is added to `Subscribem::ApplicationController` is the same reason the `owner?` method lives there: so we can access it anywhere in our controllers. We’re actually making great use of the `owner?` method inside the `authorize_owner` method.

To use this method, we’ll add it as a `before_filter` within `Subscribem::Account::AccountsController`, right underneath the class definition:

```

class Account::AccountsController < ApplicationController
  before_filter :authorize_owner, only: [:edit, :update]

```

We also want only authenticated users visiting this controller, so let’s put this line before that before filter too:

```
before_filter :authenticate_user!
```

With these `before_filters` in place, the test should now work. Let's find out with one more run of `bin/rspec spec/features/accounts/updating_spec.rb`:

```
2 examples, 0 failures
```

That wasn't too hard! The final thing that we need to account for is the situation where an account owner might enter invalid details into the form. That's quick and easy to do, so let's do that now.

Updating with invalid account details

If an account owner attempts to update an account with invalid data, we should tell them that the data is wrong and show the form again. Let's add a test for this within `spec/features/accounts/updating_spec.rb`, inside the "as the account owner" context.

```
context "as the account owner" do
  ...
  scenario "updating an account with invalid attributes fails" do
    visit root_url
    click_link "Edit Account"
    fill_in "Name", :with => ""
    click_button "Update Account"
    expect(page).to have_content("Name can't be blank")
    expect(page).to have_content("Account could not be updated.")
  end
end
```

If we run this test with `bin/rspec spec/features/accounts/updating_spec.rb`, we'll see it fail in this way:

```
Failure/Error: page.should have_content("Name can't be blank")
  expected to find text "Name can't be blank" in "Your account's dashboard."
```

The update action is still telling us that the request is successful, even though it isn't. Let's account for this behaviour within the update action of `Subscribem::Account::AccountsController` now:

```
def update
  if current_account.update_attributes(account_params)
    flash[:success] = "Account updated successfully."
    redirect_to root_path
  else
    flash[:error] = "Account could not be updated."
    render :edit
  end
end
```

This isn't quite enough to get the test to pass, as we'll see when we run it again:

```
Failure/Error: page.should have_content("Account could not be updated.")
expected there to be text "Account could not be updated."
in "Account updated successfully..."
```

This test is failing because it's not showing the content that the test is expecting. The reason for this is because there isn't a validation for an account's name yet. We never added one back when we created the `Subscribem::Account` model! Let's forgive our past selves for making this mistake and fix it now by adding this line underneath the subdomain validation inside `subscribem/app/models/subscribem/account.rb`:

```
validates :name, presence: true
```

Now with the validation, our test should pass when we re-run it:

```
3 examples, 0 failures
```

Good! Now we've got all the paths accounted for with the account updating functionality. Only account owners can update the account information and only then with a valid name. Non-account owners are politely told that they aren't allowed to perform that specific action.

Let's move on to the whole reason why we did all that: so that we can switch plans for an account.

The plan switcher

When an account owner goes to the edit account page, they should be able to see what plan their account is currently on and they should be able to change that plan if they wish. The way they'll do this is with a select box which lists all the different plans, which the user can choose from.

Let's write a new test for this flow now within the "as the account owner" context block within `subscribem/features/accounts/updating_spec.rb`:

```
context "with plans" do
  let!(:starter_plan) do
    Subscribem::Plan.create(
      :name => "Starter",
      :price => 9.95,
      :braintree_id => "starter"
    )
  end
  let!(:extreme_plan) do
    Subscribem::Plan.create(
      :name => "Extreme",
      :price => 19.95,
      :braintree_id => "extreme"
    )
  end
  before do
    account.update_column(:plan_id, starter_plan.id)
  end
  scenario "updating an account's plan" do
```

```

    visit root_url
    click_link "Edit Account"
    select "Extreme", :from => 'Plan'
    click_button "Update Account"
    expect(page).to have_content("Account updated successfully.")
    expect(page).to have_content("You are now on the 'Extreme' plan.")
    expect(account.reload.plan).to eq(extreme_plan)
  end
end

```

At the beginning of this new code, we create two new plans called the ‘Starter’ plan and the ‘Extreme’ plan. By default, the account that we’re editing will have the ‘Starter’ plan and during the flow of the test, it will switch to the ‘Extreme’ plan. Clicking the ‘Update Account’ button will still trigger the “Account updated successfully.” message to appear, but another message informing the user of the plan switch should appear also. On the very last line of this test we reload the account object and ensure that its plan has indeed be changed to the ‘Extreme’ plan.

Running this new test with `bin/rspec spec/features/accounts/updating_spec.rb` will fail because the `plan_id` column doesn’t exist on `subscriber_accounts` yet:

```

Failure/Error: account.update_column(:plan_id, starter_plan.id)
ActiveRecord::StatementInvalid:
PG::Error: ERROR:
  column "plan_id" of relation "subscriber_accounts" does not exist

```

To fix that, all we need to do is add this new column. We can do that by running the migration generator within the `subscriber` directory:

```
rails g migration add_plan_id_to_subscriber_accounts plan_id:integer
```

We can run this migration by running this command:

```
rake db:migrate
```

This will add the `plan_id` field to `subscriber_accounts`. Now that the `plan_id` field exists, our test shouldn’t complain about it being missing when we re-run it. Let’s do that now. Now it will say it can’t find the “Plan” select box:

```

Failure/Error: select 'Starter', :from => 'Plan'
Capybara::ElementNotFound:
  Unable to find select box "Plan"

```

The plan select box needs to display a list of plans that we can select for this account. We can do this by adding this code underneath the “Name” field inside the `edit.html.erb` template:

```
<p>
  <%= account.label :plan_id %>
  <% plans = Subscribem::Plan.all.map { |p| [p.name, p.id] } %>
  <%= account.select :plan_id, plans %>
</p>
```

For this field to be assignable to our model, we'll need to add it to the list of permitted attributes in `Subscribem::Account::AccountsController`:

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

Running the test now should get a little further.

```
Failure/Error: page.should have_content("You are now on the 'Extreme' plan.")
expected there to be text "You are now on the 'Extreme' plan." in ...
```

The test is now failing because it's not seeing the message that notifies it that the plan has changed. We can fix this by altering our controller to check to see if the account's `plan_id` was changed with the `update_attributes` call. The way we do that is through the `previous_changes` method provided by Active Record, like this:

`app/controllers/subscribem/account/accounts_controller.rb`

```
1 def update
2   if current_account.update_attributes(account_params)
3     flash[:success] = "Account updated successfully."
4     if current_account.previous_changes.include?("plan_id")
5       plan = current_account.plan
6       flash[:success] += " You are now on the '#{plan.name}' plan."
7     end
8     redirect_to root_path
9   else
10    flash[:error] = "Account could not be updated."
11    render :edit
12  end
13 end
```

If the previous changes to this model contain changes for the `plan_id` attribute, the `flash[:success]` message will now have some additional content added to it. This additional content *should* make our test happy. Before we can do that though, we've got some code inside the `update` action which is referencing the `plan` association on the `current_account` object. That association doesn't exist yet. We can add it to the `Subscribem::Account` model with this code:

```
belongs_to :plan, :class_name => "Subscribem::Plan"
```

That's all the things we need to make our test happy. When we run it again with `bin/rspec spec/features/accounts/updating_spec.rb` it will all pass:

4 examples, 0 failures

Now we've got account updating working pretty solidly. The next thing we want to do is hook this up to Braintree to actually subscribe the accounts to the plans.

5.3 Subscribing with Braintree

When an account is created within Subscribem, it has no `plan_id` assigned to it. This means that it is effectively on the "Free" plan. It's not until they select a plan from the admin backend that we will bill them.

In order to bill the users, we will need to ask them for their credit card details, but only once. Once Braintree has them, it will be easier to switch between plans. So that means there's two scenarios already we need to care about: the one where an account first switches their plan, and the one where the account has already switched their plan, but is looking to switch again.

Let's focus on the first scenario now.

The payment page

When a user switches their account's plan for the first time, we will create a record for them within Braintree, which will then automatically bill the user for that plan every month. Braintree thinks of objects that are subscribed to a plan as "customers", so you may see that term used often in this section.

We need to get this credit card data from the user and then transmit it directly to Braintree, remembering to not store them locally because that would be a violation of PCI-DSS⁴. Braintree will then process the credit card data and give us back a response indicating the success or failure of the transaction, and then we can go from there.

The first thing we want to do to get this feature to work is to create some code which will redirect the user to a page where they can enter credit card information if they change their plan. Once we're done with this, we'll work on transmitting the data to Braintree using the `braintree` gem.

To make sure that this capturing can happen, we can update that last test we were editing within `spec/features/accounts/updating_spec.rb` to check that the controller is redirecting to a page where we can capture payment information. Let's change the final lines of the "updating an account's plan" scenario to this:

```
expect(page).to have_content("Account updated successfully.")
plan_url = subscribem.plan_account_url(
  plan_id: extreme_plan.id,
  subdomain: account.subdomain)
expect(page.current_url).to eq(plan_url)
```

Now when the plan is altered on an account, it should redirect to the `plan_account_url` location within Subscribem. This path isn't defined yet, but we can fix that by putting this code within `subscribem/config/routes.rb`, inside the account module scope:

⁴http://en.wikipedia.org/wiki/Payment_Card_Industry_Data_Security_Standard

```
get "/account/plan/:plan_id",
  :to => "accounts#plan",
  :as => :plan_account
```

When we run the spec with `bin/rspec spec/features/accounts/updating_spec.rb`, we'll see that it's not redirecting:

```
Failure/Error: page.current_url.should == plan_url
expected: "http://test3.example.com/account/plan/1"
got: "http://test3.example.com/" (using ==)
```

This is failing because the update action within `Subscribem::Account::AccountsController` is still doing its default behaviour of redirecting to the `root_path`. This is not the only thing that is wrong with this action. We don't want it persisting the `plan_id` changes to the database yet because the user hasn't paid for their plan. Therefore, we should remove the `plan_id` parameter from the attributes and pass it around as needed:

```
def update
  plan_id = account_params.delete(:plan_id)
  if current_account.update_attributes(account_params)
    flash[:success] = "Account updated successfully."
    if plan_id != current_account.plan_id
      redirect_to plan_account_url(:plan_id => plan_id)
    else
      redirect_to root_path
    end
  else
    flash[:error] = "Account could not be updated."
    render :edit
  end
end
```

This new code will redirect the user to the `plan_account_url` path, which is where we will be showing them the credit card form. It does this by deleting the `plan_id` from the parameters, which means that it won't be passed to `update_attributes` and that the account's plan won't be changed before the user pays for it. If there is a `plan_id` parameter and it's not the same as the account's current `plan_id`, then we want to change the plan. The way we do that is redirect to `plan_account_url`, passing along the `plan_id`.

Running this test again will show that this new plan action isn't defined within the controller, giving us our next task:

```
Failure/Error: click_button "Update Account"
AbstractController::ActionNotFound:
  The action 'plan' could not be
  found for Subscribem::Account::AccountsController
```

All this new action needs to do in order to make our test pass is just *exist*. Let's create a template for this action at `app/views/subscribem/account/accounts/plan.html.erb`:

Placeholder for plan template.

With the placeholder neatly in place, when we run our test again we'll see that it now passes:

4 examples, 0 failures

Good. Now we have a page on which we can capture client details. Let's actually capture these details.

Capturing credit card details

Braintree has some really great examples of capturing payment information over in their “Braintree Ruby Examples” GitHub project⁵. One of these examples uses Braintree’s Transparent Redirect feature, and contains code that we’re going to “borrow” for this section.

Braintree’s documentation explains their Transparent Redirect feature incredibly well, so it’s recommended that you read up on it now: <https://www.braintreepayments.com/developers/api-overview>. Step 4 is crucial because if for whatever reason the user is *not* redirected back to the site, we don’t want the payment to go through automatically. Once the redirect is complete, the transaction is finalized and a message will be displayed to the user indicating success.

If we follow along with the steps in Braintree’s documentation (<https://www.braintreepayments.com/developers/api-overview>), the first step is to create a form which contains the billing information, necessary: the credit card number, the name on the card, expiration date, and CVV. For this point in time, we only want Braintree to capture the user’s credit card details, but not charge them. Charging these cards involves creating subscriptions on Braintree, and this will come later.

Let’s add these fields to the “updating an account’s plan” test in `spec/features/accounts/updating_spec.rb`, underneath the check for the plan URL:

`spec/features/accounts/updating_spec.rb`

```
1 expect(page.current_url).to eq(plan_url)
2 expect(page).to have_content("You are changing to the 'Extreme' plan")
3 expect(page).to have_content("This plan costs $19.95 per month.")
4 fill_in "Credit card number", with: "4111111111111111"
5 fill_in "Name on card", with: "Dummy user"
6 future_date = "#{Time.now.month + 1}/#{Time.now.year + 1}"
7 fill_in "Expiration date", with: future_date
8 fill_in "CVV", with: "123"
9 click_button "Change plan"
```

We’ve snuck in a check for some further information on the page too which tells the user which plan they’ll be switching to and how much it will cost them.

When we run the test again, it will complain because this information is missing from the page:

⁵https://github.com/braintree/braintree_ruby_examples. The code in particular is the form code. It’s been altered slightly from the original, but it’s important that origin is attributed to Braintree.

Failure/Error:

```
page.should have_content("You are changing to the 'Extreme' plan")
expected there to be text "You are changing to the 'Extreme' plan" in ...
```

Fair enough. We can retrieve this information for our template within a new `plan` action within `Subscribem::Account::AccountsController`:

```
def plan
  @plan = Subscribem::Plan.find(params[:plan_id])
end
```

While we're here, we should add this action to the `authorize_owner` before filter at the top of the controller, changing this line:

```
before_filter :authorize_owner, only: [:edit, :update]
```

To this:

```
before_filter :authorize_owner, only: [:edit, :update, :plan]
```

Otherwise, users who are *not* owners will be able to see this page. We don't want that to happen at all.

Then, within the `plan.html.erb` template we can display this information:

```
<div class='plan-info'>
  <p>
    You are changing to the '<%= @plan.name %>' plan.
  </p>
  <p>
    This plan costs <strong>$<%= @plan.price %></strong> per month.
  </p>
</div>
```

If we got that right, running the test again will get past that point and will now fail because the “Credit card number” field is missing:

```
Failure/Error: fill_in "Credit card number", with: "4111111111111111"
Capybara::ElementNotFound:
  Unable to find field "Credit card number"
```

This means that we will now need to create a form, but not just any form. We need a form that will submit its details to Braintree. Let's write the beginning of the form in `app/views/account/accounts/plan.html.erb` like this:

Beginnings of a Braintree form

```

1 <%= form_for :customer,
2   :url => Braintree::TransparentRedirect.url,
3   :html => {:autocomplete => "off"} do |customer| %>
4 <% end %>

```

Each line in this code has its own little bit of special importance.

We’re building a form for a credit card. Using `form_for :customer` means that all fields inside this form will be nested underneath the `customer` key in our parameters. If we look at Braintree’s documentation about Transparent Redirect ⁶ regarding form fields, it shows us that it expects these fields to be nested as such. Therefore the use of `:customer` on this line serves a purpose. We’re creating a new customer on Braintree and then later on we will subscribe the customer to the plan.

The second line uses the `:url` option to specify a URL for this form rather than letting Rails decide what it feels like would be the right URL for it. By using `Braintree::TransparentRedirect.url`, the `braintree` gem will build a URL for a new transaction to our sandbox account over on Braintree. This means that the form will be sending its data over there, rather than to our application.

The third line in this small piece of code turns off the `autocomplete` option for all elements in the form. This is important because we definitely don’t want critical things such as personally identifiable information and credit card details being autocompleted by the browser.

Now’s a good time to add the field that our test is looking for: the credit card number. Let’s add this inside the `form_for` block we just placed in `plan.html.erb`:

```

<%= customer.fields_for :credit_card do |cc| -%>
  <p>
    <%= cc.label :number, "Credit card number" %><br>
    <%= cc.text_field :number %>
  </p>
<% end %>

```

This is the first field that our test needs. If we run our test again, we’ll see this time it’s missing the “Name on card” field:

```

Failure/Error: fill_in "Name on card", :with => "Dummy user"
Capybara::ElementNotFound:
  Unable to find field "Name on card"

```

Re-running the test every time to see what field we need to add next is going to get really tedious, so let’s go ahead and add the “Name on card”, “Expiration date” and “CVV” fields now because we know we’re going to need them, and the submit button too:

⁶https://www.braintreepayments.com/docs/ruby/customers/create_tr

```

<%= customer.fields_for :credit_card do |cc| -%>
  <p>
    <%= cc.label :number, "Credit card number" %><br>
    <%= cc.text_field :number %>
  </p>
  <p>
    <%= cc.label :cardholder_name, "Name on card" %><br>
    <%= cc.text_field :cardholder_name %>
  </p>
  <p>
    <%= cc.label :expiration_date, "Expiration date" %><br>
    <%= cc.text_field :expiration_date %>
  </p>
  <p>
    <%= cc.label :cvv, "CVV" %><br>
    <%= cc.text_field :cvv %>
  </p>
  <%= customer.submit "Change plan" %>
<% end %>

```

The next time we run the test, it'll get right up to clicking the button and then it will fail like this:

```

Failure/Error: click_button "Change plan"
ActionView::Template::Error:
  Braintree::Configuration.environment needs to be set

```

Oops, seems like another hurdle has popped up before we can figure that one out. We saw this error earlier and it happened then because we didn't have Braintree's configuration set up anywhere inside our blorgh application. This error is happening now because it's not set up in the dummy app for the subscribem engine. Let's copy over the settings from blorgh/config/initializers/braintree.rb and put them into subscribem/spec/dummy/config/initializers/braintree.rcb.

This'll be enough to get our test going, but there's one more field that we need to add to this form. You may have noticed it mentioned on the Braintree documentation earlier over here: https://www.braintreepayments.com/docs/ruby/cards/create_tr. The new field is a field called `tr_data` which contains information about the new customer.

We should add this field inside the `form_for` block now by using this code:

```

<% tr_data = Braintree::TransparentRedirect.create_customer_data(
  :redirect_url => subscribem.subscribe_account_url(:plan_id => params[:plan_id]),
) %>
<%= hidden_field_tag "tr_data", tr_data %>

```

This `tr_data` field contains some information to uniquely identify this transaction to Braintree. Here's an example one (linebreaks added):

```
74b90211302b7921612912ec89ca5d3559127eea|
api_version=3&
kind=create_customer&
public_key=vnf526wrzwyg3acq&
redirect_url=http%3A%2F%2Ftest1.example.com%2Faccount%2Fsubscribe&
time=20130413061914&
```

The first part of this `tr_data` field is a unique hash which Braintree will use to ensure that the attributes for this aren't tampered with when it's been submitted. It's not too difficult for someone to come along and tamper with fields on the page, so this signing is very useful.

The remaining fields include things such as:

- The API version of Braintree in use
- The kind of API request being performed
- The public key for our sandbox, which is matched against the private key to confirm we are who we say we are
- A redirect URL so that braintree can redirect back somewhere after a successful request.
- The time at which this page was loaded, in UTC.

Within the `tr_data` definition, we're using a route that we've not defined yet:

```
:redirect_url => subscribem.subscribe_account_url(:plan_id => params[:plan_id]),
```

We can define this route within `subscribem/config/routes.rb` by using this code:

```
get "/account/subscribe",
  :to => "accounts#subscribe",
  :as => :subscribe_account
```

This `subscribe` action is where Braintree will redirect to after it receives the form parameters. The next time we run the test, we'll see it fail with this strange error:

```
Failure/Error: click_button "Change plan"
ActionController::RoutingError:
  No route matches [POST]
  "/merchants/rjyzvq7k53n45sph/transparent_redirect_requests"
```

This error is happening because Braintree's code is attempting to make a request out to `sandbox.braintreegateway.com`. This request is interpreted by `Rack::Test` (the gem underneath Capybara that's used to connect to our application) to be a request for our local application, and so it tries to serve it with the application it's using for our Capybara tests.

This weirdness is actually mentioned in Capybara's README:⁷ From Caybara's README:

⁷There's a source dive into Capybara and Rack::Test to figure out why this happening. If you're interested take a look here: <http://stackoverflow.com/questions/16013975/capybara-remote-form-request>

Rack::Test is Capybara's default driver. It is written in pure Ruby and does not have any support for executing JavaScript. Since the Rack::Test driver interacts directly with Rack interfaces, it does not require a server to be started. However, this means that if your application is not a Rack application (Rails, Sinatra and most other Ruby frameworks are Rack applications) then you cannot use this driver. **Furthermore, you cannot use the Rack::Test driver to test a remote application, or to access remote URLs (e.g., redirects to external sites, external APIs, or OAuth services) that your application might interact with.**

That's settled then. The requests will always be made to our local server rather than to Braintree's. To handle these requests, we need to have some code in place that acts before our application. This code needs to capture the sandbox request and return a response that Braintree normally would.

Braintree would normally respond with a 303 ("See Other")⁸, which would redirect the browser to the `redirect_url` link within the `tr_data` submitted by the form. At the end of this URL, it will add additional parameters, like this:

```
http://example.com/path?http_status=200
&id=vqgssrhqhxfhgrwz
&hash=0c3c641f1de3ed1c732c54cab367355350603b28
```

What are these parameters for? Well, if we look at the API documentation again over at <https://www.braintreepayments.com/de/overview> and pay attention to Steps 3 and 4, we'll see that these parameters are used to uniquely identify the transaction to Braintree once the user confirms it. We can save these parameters and then submit them to Braintree by making what's called a Server-to-Server request using the Braintree gem.

Now, how are we going to make our application pretend like it's Braintree for only *some* of the requests? Luckily the author⁹ of this book wrote a gem for this called `fake_braintree_redirect`. It's only available on GitHub, and so to add it as a dependency to our engine, we'll need to add it into the `Gemfile` using this line:

```
gem "fake_braintree_redirect", :github => "radar/fake_braintree_redirect"
```

This gem provides a piece of middleware which will intercept Braintree sandbox requests and give the expected response, with the URL and the `http_status`, `id` and `hash` parameters that we will deal with later on. To insert this piece of middleware into the middleware stack for the `Subscribem` engine so that our test no longer breaks, we can put this code underneath all the other middleware pieces in `lib/subscribem/engine.rb`

⁸303 status code: <http://httpstatus.es/303>

⁹Equal parts handsome and humble.

```

initializer "subscribem.middleware.fake_braintree_redirect" do
  if Rails.env.test?
    require "fake_braintree_redirect"
    Rails.application.config.middleware.insert_before \
      Warden::Manager,
      FakeBraintreeRedirect
  end
end

```

Does this all fix our errors though? You betcha:

```

Failure/Error: click_button "Change plan"
AbstractController::ActionNotFound:
  The action 'subscribe' could not be found
  for Subscribem::Account::AccountsController

```

Great! The form is now submitting its request, the FakeBraintree middleware is picking up on that request and then redirecting Capybara to the subscribe action within Subscribem::Account::AccountsController. Within this action we'll confirm the transaction with Braintree by using the Braintree gem.

We don't want it to actually make the request, so we will stub the method at the top of our test:

```

scenario "updating an account's plan" do
  expect(Braintree::TransparentRedirect).to receive(:confirm).
    with("query goes here")

```

This line is important, as it will stop the external request to Braintree's servers, and at the same time not be testing the Braintree gem's behaviour. Both great wins. We don't know what the query string is exactly just yet, so we've got a placeholder value.

Before we proceed, we should add some more assertions to our test to ensure that the user sees a message informing them that their plan is changed and that they're redirected back to somewhere useful. After the clicking of the "Change plan" button, add these lines:

spec/features/accounts/updating_spec.rb

```

1 click_button "Change plan"
2 expect(page).to have_content("You have switched to the 'Extreme' plan.")
3 expect(page.current_url).to eq(root_url)

```

This new code in our test will be testing the subscribe action, making sure that it does what it should.

Let's go ahead now and create the subscribe action within Subscribem::Account::AccountsController to make this transaction confirmation request to Braintree and then redirect the account owner back to the root page.

app/controllers/subscribem/account/accounts_controller.rb

```

1 def subscribe
2   @plan = Subscribem::Plan.find(params[:plan_id])
3   result = Braintree::TransparentRedirect.confirm(request.query_string)
4   if result.success?
5     current_account.update_column(:plan_id, params[:plan_id])
6     flash[:success] = "You have switched to the '#{plan.name}' plan."
7     redirect_to root_path
8   end
9 end

```

In this code, we first send the request to Braintree to confirm the transaction, using `Braintree::TransparentRedirect.confirm` and passing in the query string from the request. This is the method we're stubbing at the beginning of the test. This is the query string which will contain something like this (shown in Hash format for readability):

```

{
  "plan_id"    => "1",
  "http_status" => "200",
  "id"         => "<transaction_id>",
  "kind"       => "create_customer",
  "hash"       => "ff17342a17b89a6380653d92950f0ab3ce545ce8"
}

```

The `plan_id` key here is just for our uses, we use it within the action to find the plan so that we can display a flash message to the user. The hash is used to ensure the query string hasn't been tampered with along the way. Everything else besides `plan_id` Braintree accepts willingly.

The last two lines set the flash message and redirect, nothing special there.

Running `bin/rspec spec/features/accounts/updating_spec.rb` again will now error because we got the query string wrong:

```

Braintree::TransparentRedirect received :confirm with unexpected arguments
expected: ("query goes here")
got: ("plan_id=2
      &http_status=200
      &id=a_fake_id
      &kind=create_customer
      &hash=<hash>")

```

Here we see the query string again. We need to replace the “query goes here” with some real parameters. Rather than build this hash ourselves, we'll get by with a little help from our friends. Let's change the `should_receive` line at the top of our test to read this:


```

query_string = Rack::Utils.build_query(
  :plan_id => extreme_plan.id,
  :http_status => 200,
  :id => "a_fake_id",
  :kind => "create_customer",
  :hash => "<hash>"
)
mock_transparent_redirect_response = double(:success? => true)
Braintree::TransparentRedirect.
  should_receive(:confirm).
    with(query_string).
    and_return(mock_transparent_redirect_response)

```

Note that your hash is going to be different. The `Rack::Utils.build_query` method here will build a query string from a Hash object so that we don't have to. We then pass this String to the `with` method on our `should_receive` assertion. At the end, we use `and_return` to return a stub object which has a method called `success?` which returns `true`, which is what our action needs in order to succeed.

Now what happens when we run our test? Let's find out with `bin/rspec spec/features/accounts/updating_spec.rb`:

```
4 examples, 0 failures
```

Most excellent! Now our users are able to switch their accounts into a new plan and have a customer record on Braintree created for them. Next, we're going to need to create a subscription also. This is relatively easy, so let's look at this now.

Braintree's documentation about creating a subscription ¹⁰ shows that it's pretty straight forward to create a new subscription. All we need to do is call code like this:

```

result = Braintree::Subscription.create(
  :payment_method_token => "the_token",
  :plan_id => "silver_plan"
)

```

The payment method token is the token provided by Braintree that uniquely identifies the credit card that the user has provided, and the plan id is the unique Braintree-specific plan id from our plan. Therefore, in order to create a new subscription in our `Account::AccountsController`'s `subscribe` action, we only need to add this code to the `if result.success? success` clause:

```

subscription_result = Braintree::Subscription.create(
  :payment_method_token => result.customer.credit_cards[0].token,
  :plan_id => @plan.braintree_id
)

```

The `result` object from a `Braintree::TransparentRedirect.confirm` call returns more than just an object with a method called `success?`. It also has a method called `customer` which itself has a method

¹⁰<https://www.braintreepayments.com/docs/ruby/subscriptions/create>

called `credit_cards` which returns an array of credit card objects associated with the user. Rather than having the card number and so on associated with these objects, Braintree returns a unique token for this card.

We will need to stub this whole chain in our test, but first let's stub the `Braintree::Subscription.create` call in our test to return the result we want. We can do that by adding this code to the top of the test:

```
subscription_params = {
  :payment_method_token => "abcdef",
  :plan_id => extreme_plan.braintree_id
}
expect(Braintree::Subscription).to receive(:create).
  with(subscription_params).
  and_return(double(:success? => true))
```

With this new code, we're stubbing the call to `Braintree::Subscription.create` to receive the `payment_method_token` parameter and a `plan_id` parameter. We don't care what this method does – because that's Braintree's responsibility, not ours – but we do care that it's called with the right parameters.

Running the test again will show that we now need to alter the `mock_transparent_redirect_response` stubbing:

```
Failure/Error: click_button "Change plan"
Stub received unexpected message :customer with (no args)
```

Right underneath the `mock_transparent_redirect_response` definition in our test, let's put this code:

```
allow(mock_transparent_redirect_response).
  to(receive_message_chain(:customer, :credit_cards).
    and_return([double(:token => "abcdef")]))
```

By using `stub_chain`, we can stub a series of methods. In this case, we've stubbed the chain of `result.customer.credit_cards` and now have that returning an array which itself contains a double that responds to `token` with `"abcdef"`, which is what our double for `Braintree::Subscription.create` is expecting as the `payment_method_token` parameter.

Running the test one more time will show it working once again:

```
4 examples, 0 failures
```

Excellent! When a user selects a plan, the `update` action within `Account::AccountsController` detects this and redirects to the `plan` action. The `plan` action displays a form containing important transactional information that Braintree requires in order to process a transaction.

The form from this action is supposed to go off to Braintree to be processed, but thanks to a weird feature in Capybara, when we run our tests the form will be submitted back to our local application. We handled the response for that with the `fake_braintree_redirect`¹¹ gem. Once Braintree (or the fake Braintree) has finished processing the transaction, it redirects back to the application to the `subscribe` action which then

¹¹https://github.com/radar/fake_braintree_redirect

confirms the request by passing the query string from the request back to Braintree. We're stubbing the method that does this in our test so that we don't actually make a request to Braintree, which may make our test slow or possibly even make it fail.

Now that we've got some tests covering creating a subscription within Braintree, let's see it working for ourselves.

Playing with the Braintree Sandbox

To test this out for ourselves, we'll need to start the `hosted_forums` application by running the `rails s` command inside that directory. Once that's started, we can open our browser and go to `http://localhost:3000` to view the app.

In order to see the plan switcher for ourselves, we'll need an account. Clicking "Account Sign Up" will let us sign up for this new account, so do that now. Fill in the form with some test information, using "foo" for the subdomain. Once the "Create Account" button is pressed, the form may fail to submit. This is because the application is attempting to route to a subdomain, which the browser doesn't understand at this point in time.

To make the browser understand it, we will need to edit our `/etc/hosts` file ¹². Open this file as a super user and add in these lines at the end:

```
127.0.0.1 blorghapp.local
127.0.0.1 test.blorghapp.local
```

If we attempt to sign up again through `http://foremapp.com:3000` this time and with a subdomain of "test", it will work because our `/etc/hosts` file is now telling our system where it can find `foremapp.local` and `test.foremapp.local`,

Once the account sign up is complete, we will need to sign in. Once we're signed in, we can then begin to see this new feature in action. Clicking "Edit Account" will take us to the feature we've spent a lot of time building, a simple form for the account details containing just a name and the plan select box. The plan select box here has been populated thanks to the Rake task (`subscribem:import_plans`) we ran in the first section of this chapter.

Selecting any plan from this list and then clicking "Update Account" will show us the transaction form that we just built:

¹²Windows users, this is inside your `WINDOWS` folder at `drivers/etc`.

Account updated successfully.
Signed in as me@ryanbigg.com · [Edit Account](#)

You are changing to the 'Silver' plan.

This plan costs **\$15.0** per month.

First name

Last name

Credit card number

Expiration date

CVV

Transaction form

Fill this form out now with your name, a credit card number of “4111 1111 1111 111”, an expiry date formatted as MM/YY in the future, and a CVV of your choosing. Hitting the “Change plan” button on this form will send off a request to Braintree and it will respond. Once that process is complete, we will see the “You have switched to the ‘

Switching over to the terminal where our server is running, we can see that the Braintree gem has made a couple of requests back to Braintree, indicated by these lines:

```
[Braintree] [timestamp] POST /transparent_redirect_requests/.../confirm 201  
[Braintree] [timestamp] POST /subscriptions 201
```

If we sign into the Braintree sandbox now (<https://sandbox.braintreegateway.com>) and go to Subscriptions and click “Search”, we’ll see the one subscription that we have now just created:

Found 1 Subscription

Download Add-on/Discounts

Download Subscriptions

Edit

Search Criteria

Merchant Accounts: rcyyjcnysfpmwbhf

Status: All

Subscription ID	Plan Name	Price	Trial Period	Billing Cycle	Customer Name	Status	Actions
bt86rm	Silver	\$15.00 USD	none	Every 1 Month(s)		Active	<div>EditCancel</div>

A wild subscription appears

We've come a long way so far. We have a form that allows us to submit transaction data to Braintree and to actually charge customers. How good is that feeling of accomplishment?

We've got a couple of things to do in order to wrap this feature up. First of all, if we go back to the "Edit Account" screen, select a plan again and fill in the transaction form with invalid details (i.e. a credit card number of simply "1"), the success message will still appear. Obviously "1" is not a valid credit card number, and so Braintree should fail.

We can check and make sure it is actually failing by adding the pry gem to our application's Gemfile:

```
group :development, :test do
  ...
  gem "pry"
end
```

The Pry gem allows us to pry into code and see what it's doing. It's got other, equally-useful features and there's even a Railscast¹³ detailing it.

We'll need to run `bundle install` to install pry and restart our server for it to be loaded, ready for what we're about to do.

And then inside our `Subscribem::Account::AccountsController`'s `update` action, right after the call to `Braintree::TransparentRedirect.confirm`, we can put this line:

```
binding.pry
```

Let's now edit the account and use invalid data again. This time, the page will hang. This is because `binding.pry` has stopped the execution of the action and has opened up a console in the terminal window for us to use to inspect our code with. In the console, we'll see this output:

¹³<http://railscasts.com/episodes/280-pry-with-rails>

```
[Braintree] [<timestamp>]
POST /transparent_redirect_requests/psks36gw3635fvqb/confirm 422
From:
.../subscribem/app/controllers/subscribem/account/accounts_controller.rb
@ line 25 Subscribem::Account::AccountsController#subscribe:
22: def subscribe
23:   @plan = Subscribem::Plan.find(params[:plan_id])
24:   @result = Braintree::TransparentRedirect.confirm(request.query_string)
=> 25:   binding.pry
26:   flash[:success] = "You have switched to the '#{plan.name}' plan."
27:   redirect_to root_path
28: end
```

The first line shows the Braintree Transparent Redirect request returning a result of 422, which we know means “Unprocessable entity”¹⁴, typically indicating the data is invalid. This is good, in this case, because we *want* it to fail.

For all intents and purposes, this console provided by Pry will act like an IRB session that’s looking right into the very soul of our code. If we ask it to return `result` by typing in that word and pressing enter, it will give us something like this:

```
#<Braintree::ErrorResult params:{...}
errors:<,
  transaction/credit_card:
    [(81703) Credit card type is not accepted by this merchant account.,
     (81710) Expiration date is invalid.,
     (81716) Credit card number must be 12-19 digits.]:>>
```

The `result` variable is a `Braintree::ErrorResult` instance, which is what is returned by the `braintree` gem when things go bad. These errors are returned by Braintree’s gateway, and yet our transaction still “succeeds”; we’re shown the “You have switched...” message when we shouldn’t be.

If we type `exit` into this console and hit enter, application processing will resume and we’ll see that successful switching message again. This is obviously a bad thing, but it’s not too difficult to fix at all.

When transactions go bad

When a user enters invalid information into the form, we want Subscribem to most definitely *not* change their plan, but rather inform them that their details are incorrect.

To test this we’ll need to fill in the credit card form with invalid details. Before we do this, let’s move the query string out of the “updating an account’s plan” scenario and into it a `let` block, underneath the one for `extreme_plan`:

¹⁴<http://httpstatus.es/422>

```

let(:query_string) do
  Rack::Utils.build_query(
    :plan_id => extreme_plan.id,
    :http_status => 200,
    :id => "a_fake_id",
    :kind => "create_customer",
    :hash => "<hash>"
  )
end

```

Doing it this way will allow us to re-use this query string within our new test. The “updating an account’s plan” scenario is now just responsible for stubbing out the call to `Braintree::TransparentRedirect.confirm` to return a fake object that responds truthfully to a `success?` call.

Let’s write a new test in this file, right underneath the last scenario, to account for the situation where `Braintree::TransparentRedirect.confirm` returns an object where `success?` is false.

```

scenario "can't change account's plan with invalid credit card number" do
  expect(Braintree::TransparentRedirect).to receive(:confirm).
    with(query_string).
    and_return(double(:success? => false))
  visit root_url
  click_link "Edit Account"
  select "Extreme", :from => 'Plan'
  click_button "Update Account"
  page.should have_content("Account updated successfully.")
  plan_url = subscribem.plan_account_url(
    :plan_id => extreme_plan.id,
    :subdomain => account.subdomain)
  page.current_url.should == plan_url
  page.should have_content("You are changing to the 'Extreme' plan")
  page.should have_content("This plan costs $19.95 per month.")
  fill_in "Credit card number", :with => "1"
  fill_in "Name on card", :with => "Dummy user"
  future_date = "#{Time.now.month + 1}/#{Time.now.year + 1}"
  fill_in "Expiration date", :with => future_date
  fill_in "CVV", :with => "123"
  click_button "Change plan"
  page.should have_content("Invalid credit card details. Please try again.")
  page.should have_content("Credit card number must be 12-19 digits")
end

```

At the top of this test, `Braintree::TransparentRedirect.confirm` will return an object with `success?` returning false, meaning that the action has failed. In the remainder of the test, we go through the process of updating an account’s plan again and when the “Change plan” button is pressed we’re expecting to see an error message, rather than a successful “You have switched …” message.

Does this happen? Let’s find out by running our test again:

```
Failure/Error: click_button "Change plan"
ActionView::MissingTemplate:
  Missing template subscribem/account/accounts/subscribe ...
```

The test is failing now because when the “Change plan” button is pressed on the form, the subscribe action is called. When `Braintree::TransparentRedirect.confirm` is called in that action, it isn’t successful and therefore never gets into the “true” case of the `if` statement within that action. Due to there being no code to determine what to do in the case of a failure, it falls back to rendering the same template.

We can account for the failure within this action by adding an `else` to the `result.success?` check, which just re-renders the plan template:

`app/controllers/subscribem/account/accounts_controller.rb`

```
1 def subscribe
2   @plan = Subscribem::Plan.find(params[:plan_id])
3   @result = Braintree::TransparentRedirect.confirm(request.query_string)
4   if @result.success?
5     subscription_result = Braintree::Subscription.create(
6       :payment_method_token => @result.customer.credit_cards[0].token,
7       :plan_id => @plan.braintree_id
8     )
9     current_account.update_column(:plan_id, params[:plan_id])
10    flash[:success] = "You have switched to the '#{plan.name}' plan."
11    redirect_to root_path
12  else
13    flash[:error] = "Invalid credit card details. Please try again."
14    render "plan"
15  end
16 end
```

In the `else` here, we set up `flash[:error]` which the test is testing for, simply informing the user that (most likely) their credit card details are invalid and that they should try again. Calling `render "plan"` will render the plan template again and bring the user back to the credit card form so that they can fill it in again.

Thankfully, Braintree returns a list of errors in a method called `message` on the `result` object, so we can call out to that in our view to display any messages. Let’s do that now by putting this code above the `form_for` call in `app/views/account/accounts/plan.html.erb`:

```
<% if @result && @result.message %>
  <ul class="gateway_messages">
    <% @result.message.split("\n").each do |message| %>
      <li><%= message %></li>
    <% end %>
  </ul>
<% end %>
```

The next time we run the test, we’ll see it’s failing because it can’t find `message` on our result stub:


```
Failure/Error: click_button "Change plan"
ActionView::Template::Error:
  Stub received unexpected message :message with (no args)
```

To fix this, we'll change the stubbing at the top of our spec to read like this:

```
message = "Credit card number must be 12-19 digits"
result = double(:success? => false, :message => message)
allow(Braintree::TransparentRedirect).to receive(:confirm).
  with(query_string).
  and_return(result)
```

Next time we run the test, it will all pass:

```
5 examples, 0 failures
```

We're now accounting for both possible results for the `Braintree::TransparentRedirect.confirm` call, a successful call where a customer's account is created on Braintree and the account within the application has its plan changed, and the failed response where Braintree returns error messages and no local account plan changes are made.

When an account owner selects a plan and fills in their credit card information and hits that "Update Account" button, Braintree will receive the information and create a new subscription from that.

But what happens when an account owner wants to change their subscription from one plan to another? We will need to account for that situation, which is what the next section will cover.

5.4 Updating Braintree Subscriptions

In the last section, we covered how to create a new subscription with Braintree using their Transparent Redirect feature. In this section, we'll cover the use case of an account owner updating their plan.

When an account owner goes to change their plan, we should first check to see if they have a subscription with Braintree. Rather than hitting Braintree every time we want to make this check, we'll check an attribute on the `Subscription::Account` record which will be the unique Braintree subscription ID. This subscription ID field will be set when an account owner first subscribes to a new plan.

If there is an ID in this field, then we will update the subscription on Braintree that matches that Subscription ID and tell the user instantly that they've switched to the new plan. There'll be no need for them to fill in the transaction form again because Braintree has already captured their details. If there is no ID, then the user needs to create a new subscription, which is what the code that we have in place already covers. This section will be covering what to do in the case where the Subscription ID is already set.

Tracking Subscription ID

The first thing to do here is to add a field to track the Subscription ID and ensure that it's being tracked by testing for it. Let's do this quick and easy now before we move on to updating subscriptions in Braintree. Let's add this check now right at the end of the "happy path" spec in `updating_spec.rb`; the first one we worked on:

```
...
expect(page.current_url).to eq(root_url)
expect(account.reload.braintree_subscription_id).to eq("abc123")
end
```

This attribute doesn't exist yet, so we will need to create it. We can do this by creating a new migration within our engine which will add this field by running this command:

```
rails g migration add_braintree_subscription_id_to_subscribem_accounts \
  braintree_subscription_id:string
```

To run this migration on our test database, we'll run the by-now-familiar command:

```
rake db:migrate
```

When we run the test again, we'll see quite clearly that this column isn't being set to anything at all:

```
Failure/Error: account.reload.braintree_subscription_id.should == "abc123"
  expected: "abc123"
  got: nil (using ==)
```

This is good and what we can expect. We're not setting this attribute right now, and so its value is going to be nil. In order to set it, we will need to modify the subscribe action in `Subscribem::Account::AccountsController` to take the subscription's ID out of the object that `Braintree::Subscription.create` gives back.

`Braintree::Subscription.create`'s object will be a `Braintree::SuccessfulResult` message which will contain all the information about a subscription that we will ever need. Right after the `update_column` call to update the account's `plan_id`, let's put one to update its `braintree_subscription_id` based on what's returned from `subscription_result` too:

```
current_account.update_column(:plan_id, params[:plan_id])
subscription_id = subscription_result.subscription.id
current_account.update_column(:braintree_subscription_id, subscription_id)
```

Now our controller is updating the `braintree_subscription_id` column on the `current_account` object with the subscription ID that is returned from calling `Braintree::Subscription.create`. Is our test happy about this? When we run it again we'll see that it's not quite happy yet:

```
Failure/Error: click_button "Change plan"
  Stub received unexpected message :subscription with (no args)
```

The object that's being returned from our `Braintree::Subscription.create` stub in our test doesn't have a `subscription` method, which is what our controller is expecting. Let's fix this stub to return a stubbed object for subscription which has that fake ID of "abc123" by replacing the whole stub with this code:

```
subscription_result = double(:success? => true,
                             :subscription => double(:id => "abc123"))
Braintree::Subscription.
  should_receive(:create).
    with(subscription_params).
    and_return(subscription_result)
```

This should be enough to get our test to pass. When we run it again, we'll see that this indeed does work:

```
5 examples, 0 failures
```

Great. This means that the subscribe action is now setting an account's `braintree_subscription_id` attribute right after the account has been subscribed to a plan on Braintree. It's this attribute that we will use in the next section to determine the course of events when a user switches their plan on the "Edit Account" screen.

Updating a subscription

When an account has already subscribed to a plan, we don't want the account's owner to have to fill out the credit card details again. This is because Braintree already has those kept within its system. Instead, all we want to do is send a message to tell Braintree to update the subscription to the plan which has been selected.

With that process in mind, let's write a new test for this in `spec/features/accounts/updating_spec.rb`.

```
scenario "changing plan after initial subscription" do
  account.update_column(:braintree_subscription_id, "abc123")
  visit root_url
  click_link "Edit Account"
  select "Extreme", :from => 'Plan'
  click_button "Update Account"
  page.should have_content("You are changing to the 'Extreme' plan.")
  page.should have_content("This plan costs $19.95 per month.")
  click_button "Change plan"
  page.should have_content("You have switched to the 'Extreme' plan.")
  page.current_url.should == root_url
  account.reload.plan.should == extreme_plan
end
```

This scenario starts out simple: we update the `braintree_subscription_id` on the account object. When the account is edited by its account owner to select a new plan and they click the "Update Account" button, they should be shown a confirmation page which tells them the name of the plan and the price. Just like the `plan` action does. Unlike the original version of this action, this new action should just have a "Change plan" button that the account owner can press to change their plan. Once the button is pressed, they'll see the "You have switched..." message which indicates their account is now on a new plan. At the end of the test, we assert that the user is now back to their subdomain's `root_url` and that the plan has been updated correctly.

When we run this test, we'll see that it's failing because Braintree doesn't know anything about this subscription.

```
Failure/Error: click_button "Change plan"
Braintree::NotFoundError:
  Braintree::NotFoundError
```

Alright, so we’ve got our first problem to solve for this test. How are we going to solve it? Well, we’ve already got the code redirecting from the update action to the plan action and displaying the plan’s information which the test is looking for. A logical thing to do would be to put the “Change plan” button within the `plan.html.erb` template, but only if the account already has a `braintree_subscription_id`. So let’s do that.

To do this, we will move all the markup in the template that isn’t the `div.plan-info` element out to a partial. We will do the same thing with the code that we’re about to write too. This is just so that if we look at this code later, we only want to see one “path” through the template at a time.

Let’s move that markup into a new partial called `app/views/subscribe/account/accounts/_new_subscription.html.erb`:

```
<% if @result && @result.message %>
  <ul class="gateway_messages">
    <% @result.message.split("\n").each do |message| %>
      <li><%= message %></li>
    ...
  <%= hidden_field_tag "tr_data", tr_data %>
  <%= customer.submit "Change plan" %>
<% end -%>
```

In this markup’s place back in `plan.html.erb`, we’ll render this new partial:

```
<%= render "new_subscription" %>
```

Now, in order to update an existing subscription we need that “Change plan” button to still appear. For that, we’ll render another partial:

```
<% if current_account.braintree_subscription_id.present? %>
  <%= render "existing_subscription" %>
<% else %>
  <%= render "new_subscription" %>
<% end %>
```

In that new partial (`app/views/subscribe/account/accounts/_existing_subscription.html.erb`), we’ll put the following code to provide the “Change plan” button that the test wants:

```
<%= form_tag confirm_plan_account_path(:plan_id => params[:plan_id]) do %>
  <%= submit_tag "Change plan" %>
<% end %>
```

This `confirm_plan_account_path` route helper isn’t currently defined, but we can easily fix this by putting this code into `subscribe/config/routes.rb`, inside the `SubdomainRequired` constraint:

```
post "/account/confirm_plan",
  :to => "accounts#confirm_plan",
  :as => :confirm_plan_account
```

For this route to work, we will need an action in `Subscribem::Account::AccountsController` called `confirm_plan`:

```
def confirm_plan
  @plan = Subscribem::Plan.find(params[:plan_id])
  subscription_id = current_account.braintree_subscription_id
  subscription_result = Braintree::Subscription.update(subscription_id,
    :plan_id => plan.braintree_id)
  if subscription_result.success?
    current_account.update_column(:plan_id, plan.id)
    flash[:success] = "You have switched to the '#{plan.name}' plan."
    redirect_to root_path
  end
end
```

In this action, we're finding the plan based on the `plan_id` parameter that's passed in from the `form_tag` in the partial. We're also finding the current account's `braintree_subscription_id`. With these two pieces of information, we're able to make a call to `Braintree::Subscription.update` which will update the subscription on Braintree. If this action is successful, we show the successful message to the account owner.

We don't want `Braintree::Subscription.update` making a call to Braintree's API in our test, so we should stub it to return a successful response. We can do this by putting this code at the top of our test:

```
scenario "changing plan after initial subscription" do
  expect(Braintree::Subscription).to receive(:update)
    with("abc123", { :plan_id => extreme_plan.braintree_id }).
    and_return(double(:success? => true))
```

Does this all work? Let's find out with a run of the test: `bin/rspec spec/features/accounts/updating_spec.rb`:

```
6 examples, 0 failures
```

Yes, it does work! This new feature will correctly update an account's plan without the account owner having to re-enter their credit card information. All the heavy lifting has been done by Braintree, which makes managing the subscriptions so much easier.

The final thing we should do for this test is to account for when Braintree returns an unsuccessful response for the `Braintree::Subscription.update` call.

When updating a subscription goes bad

If `Braintree::Subscription.update` fails, we want to display a message to the user indicating this. Right now, we are not accounting for that situation in our `confirm_plan` action. Before we can account for it, we should write a test for this, underneath the test we just wrote.

```

scenario "changing plan after initial subscription fails" do
  expect(Braintree::Subscription).to receive(:update).
    and_return(double(:success? => false))
  account.update_column(:braintree_subscription_id, "abc123")
  visit root_url
  click_link "Edit Account"
  select "Extreme", :from => 'Plan'
  click_button "Update Account"
  expect(page).to have_content("You are changing to the 'Extreme' plan.")
  expect(page).to have_content("This plan costs $19.95 per month.")
  click_button "Change plan"
  expect(page).to have_content("Something went wrong. Please try again.")
end

```

This test is almost a carbon copy of the test we just wrote, with only differences at the beginning and the end. At the beginning of this test, we're stubbing `Braintree::Subscription.update` to return an object where `success?` returns false, rather than true. At the end of the test, we're checking for an error message informing the user that something went wrong.

When we run this test, we'll see it fail like this:

```

Failure/Error: click_button "Change plan"
ActionView::MissingTemplate:
  Missing template subscribem/account/accounts/confirm_plan ...

```

This test is failing in this way because we're not accounting for the false scenario of `result.success?` in the `confirm_plan` action of `Subscribem::Account::AccountsController`. Let's fix that now by changing the `if` statement inside this action to include an `else` as well:

```

if subscription_result.success?
  current_account.update_column(:plan_id, plan.id)
  flash[:success] = "You have switched to the '#{plan.name}' plan."
  redirect_to root_path
else
  flash[:error] = "Something went wrong. Please try again."
  render "plan"
end

```

Now if `subscription_result.success?` returns false, we will be showing a flash message to the user asking them to try it again and showing them the `plan` template again. Does this work? Let's find out by running our test again with `bin/rspec spec/features/accounts/updating_spec.rb`:

```
7 examples, 0 failures
```

Yup, all our account updating tests are working. Are the rest? Find out with `bin/rspec spec`:

```
21 examples, 0 failures
```

Everything's working. Excellent!

5.5 Summary

Our Subscribem engine now supports accounts subscribing to a plan, and changing that plan if they choose to do so.

To make that work, we first built a plan importer Rake task so that we could fetch all the plans in Braintree and update their details. Running this Rake task will update all the existing plans within our system, as well as creating new ones if they don't exist.

Using those plans from the Rake task, we created a feature to allow the user to edit their account and switch their plan. The first time an account selects a plan, we will create a new subscription in Braintree. The next time they change their plan, we will simply update that subscription.

In the next chapter, we continue our work here adding *plan restrictions*. This new feature will restrict certain features about accounts so that people on smaller, cheaper plans get less, and people on dearer plans get more.

6. Account Restrictions

This chapter is going to be rather short and take care of “cleaning up” the application, preparing it for a release.

In the previous chapter, we implemented the subscriptions feature of our engine which provided different plans for accounts to subscribe to. The different plans have different prices, and the more expensive the plan the more resources of the system the accounts should be entitled to.

That’s one of the things that we’re going to be implementing this chapter: restricting the number of forums that any account may use, based on their plan. The way we’re going to do that is by putting a validation on the `Forem::Forum` model which will check the current account’s plan and error out if the account is at its maximum forum limit.

6.1 Limiting forum creation

The code for restricting the number of forums that any account can have is quite specific to our application, and wouldn’t be useful living in either of the Forem or Subscribem engines. Therefore, this code will live in the application which is currently the combination of these two.

The first thing we’re going to do is write a test that will check that when an account is at its forum limit and it attempts to have a new forum created for it, Active Record will not allow it and will provide an error message.

Let’s put this new test inside `spec/models/forem/forum_spec.rb` inside the hosted forums application now:

`spec/models/forem/forum_spec.rb`

```
1 require "spec_helper"
2 require "subscribem/testing_support/factories/account_factory"
3 require "forem/testing_support/factories/categories"
4 require "forem/testing_support/factories/forums"
5 describe Forem::Forum do
6   context "an account" do
7     let(:account) do
8       FactoryGirl.create(:account_with_schema).tap(&:save)
9     end
10    before do
11      Apartment::Tenant.switch(account.subdomain)
12      5.times do |i|
13        FactoryGirl.create(:forum)
14      end
15    end
16    context "on the 'Starter' plan" do
17      before do
18        account.plan = Subscribem::Plan.create(:name => 'Starter')
19        account.save
20      end
21      it "is limited to 5 forums" do
```



```

22     forum = FactoryGirl.build(:forum)
23     forum.save
24     message = "You are not allowed to create more" +
25               " than 5 forums on your current plan."
26     forum.errors[:base].should include(message)
27   end
28 end
29 end
30 end

```

In this test, we are requiring some factories from both of the engines that our application depends on, the subscribem and forem engines. These factories will allow us to create accounts, categories and forums which are necessary for this test.

At the beginning of the test we create an account and associate it with a new plan called “Starter”. By having the account on this plan, it should be limited to only 5 forums. We switch into the account’s schema and create 5 forums, and then the real test begins. In the real test, we build a new forum object using `FactoryGirl.build`, attempt to save it and then verify that the error message we expect occurs.

When we run this test with `bin/rspec spec/models/forem/forum_spec.rb` we’ll get this error:

```

1) Forem::Forum an account on the 'Starter' plan is limited to 5 forums
Failure/Error: account.plan = Subscribem::Plan.create(:name => 'Starter')
ActiveRecord::StatementInvalid:
  PG::Error: ERROR:  relation "public.subscribem_plans" does not exist
LINE 5:          WHERE a.attrelid = '"public"."subscribem_plans"...

```

This error is coming up because we haven’t yet copied the new migrations for plans that are in the engine into the application. Let’s now run these three commands to copy the migrations and migrate the development and test databases:

```

rake railties:install:migrations
rake db:migrate

```

With the migrations copied over and run, when we run the test again we’ll see it failing correctly:

```

1) Forem::Forum an account on the 'Starter' plan is limited to 5 forums
Failure/Error: forum.errors[:base].should include(message)
expected [] to include
"You are not allowed to create more than 5 forums on your current plan."

```

The test is correctly checking for a validation error to occur when we attempt to create a new forum. That error isn’t showing up yet, because we have no code to prevent the 6th forum creation from happening. Let’s add this code in now.

Creating limits for plans

To define the limits for the plans, we're going to have a simple Hash inside the `Subscribem::Plan` class which will be added by a decorator in our hosted forums application. This Hash's keys and values will be the plan names and their forum limits respectively. To make our test work, we'll run a validation before a forum is created to find the current account, then find its plan and look up the plan in the hash. If the plan is in the hash, then it is limited. If not, then we'll assume it's unlimited.

We can accomplish this by defining this validation in to a new extender for the `Forem::Forum` class, like this:

`app/extendes/models/forem/forum.rb`

```

1 Forem::Forum.class_eval do
2   validate :check_plan_limit
3   def check_plan_limit
4     current_schema = Apartment::Tenant.current
5     account = Subscribem::Account.find_by_subdomain!(current_schema)
6     plan_limit = Subscribem::Plan::LIMITS[account.plan.try(:name)]
7     if plan_limit && Forem::Forum.count == plan_limit
8       errors.add("base", "You are not allowed to create more than" +
9         " #{plan_limit} forums on your current plan.")
10    end
11  end
12 end

```

In this method we use the `Apartment::Tenant.current` to get the current 'database' that Apartment is using. Due to the fact that we're using PostgreSQL, this will actually return the current schema's name. Luckily for us, the schema's name matches the subdomain attribute on `Subscribem::Account` objects, so we can use that to find the account using `find_by_subdomain!`. Once we have the account, then it's a short jump to find the account's plan's name, using `account.plan.try(:name)`. The `try` is there just in case an account does not currently have a plan associated with it. We then check the `LIMITS` hash on `Subscribem::Plan` and see if it contains a key matching the plan's name. If it does, then the code will check if the current amount of forums within this schema is equal to the limit. If that's the case, then the code will not allow any more forums to be created.

When we run our test again, we'll see that we're missing the `Subscribem::Plan::LIMITS` constant:

```

1) Forem::Forum an account on the 'Starter' plan is limited to 5 forums
   Failure/Error: FactoryGirl.create(:forum)
   NameError:
     uninitialized constant Subscribem::Plan::LIMITS
     # ./app/models/forem/forum_decorator.rb:7:in `check_plan_limit'

```

This output indicates that our validator is being run, which is assuring. Let's create this new constant in a new file:

app/extendes/models/subscribem/plan.rb

```
1 Subscribem::Plan::LIMITS = { "Starter" => 5 }
```

When we run our test once more, we'll see our validation is now running correctly:

1 example, 0 failures

Great! Now accounts that are on a limited plan are *actually limited*, we can be sure that accounts that want more forums will need to upgrade to a more expensive plan with a higher limit.

6.2 Summary

That's all for this book for now. If there's anything else you would like to see covered within its pages, please contact me@ryanbigg.com and we'll talk about it.

Thank you for reading!