A Multitenant Devise App

Create our App

First let's create our new app with:

```
rails new devise_app
```

After that let's get the basics of a working app with a single model using scaffolding with:

```
rails g scaffold task name
```

This will give us a single model, view and controller around a Task, along with a single field for each task called ‘name’. Note that we did not need to add the :string option to name, as this is the default if a attribute type is not specified. Now run ‘rails db:migrate’ and visit localhost:3000/tasks and verify you can CRUD tasks.

Install Devise

Now let's add authentication via devise. First, add devise to your gemfile with “gem devise” and run ‘bundle’. Now let's run the devise generator to actually install devises various files, folders and services into our app. We will do this with:

```
rails g devise:install
```

This is actually pretty pedestrian as this only creates our initializer and an english language locales.

Now we will generate a model using devise. This model will represent our users, so it stands to reason we will call the model ‘user’ and run:

```
rails g devise user
```

The important files generated from this command are the user model and the migration. Now run “rails db:migrate” to create the user table.

Now when we visit /users/sign_up we can sign up for an account. This is great, but right now all of our authentication soft lives outside of our tasks, and the whole point is to limit users ability to ‘do stuff’ to our tasks, so for that we will add **before_action :authenticate_user!** to our tasks_controller.

```
class TasksController < ApplicationController
  before_action :set_task, only: [:show, :edit, :update, :destroy]
  before_action :authenticate_user!

  # GET /tasks
  # GET /tasks.json
  def index
    @tasks = Task.all
  end
...
```

Now when we try to visit /tasks we are greeted with a login page. No user, not tasks! Now once we have signed in we are able to manage our tasks.

Before going forward, a couple of quick notes. When testing with devise I like to edit /initializers/devise.rb and allow a shorter password link you can change config.password_length = 6..128 to config.password_length = 4..128. Also you for /sign_out to work, in the same file you may need to   config.sign_out_via = :delete to be   config.sign_out_via = :get

So this is nice. We have a model, and we have authentication tied to that model and all its CRUD behavior. And my wife and my buddies could create users, and manage their own tasks, but what if I want my wife to be able to create tasks in my account, and I want her to be able to do the same. And the wannapreneur in us starts to think, hey let's make this app so other people can use it and create their own little group of people that can manage tasks. Now we begin to dip our toe into the realm of multitenancy.

Finally, be sure and add the following code to your layout file so flash messages from devise are displayed appropriately:

```
<% flash.each do |name, msg| %>
       <%= content_tag :section, msg, :id => "flash_#{name}", :class => "flash" %>
<% end %>
```

```
<!DOCTYPE html>
...
  <body>
        <% flash.each do |name, msg| %>
               <%= content_tag :section, msg, :id => "flash_#{name}", :class => "flash" %>
        <% end %>

    <%= yield %>
  </body>
</html>
```

Create an Organization

So the first thing we would need to make this happen is to create some kind of group that all of our users will fit into. We will call this group at organization. It also might be called a company. So to make this happen let's first create a model for our organization. We will keep it simple for now with:

```
 rails g model organization name
```

This command will create a model called organization along with a migration file, and a single string field called name. Our plan is that each of our users will be associated with an organization, and to do that we will need a foreign key association between our user and organization. To make that happen we will add an organization_id field to our user with

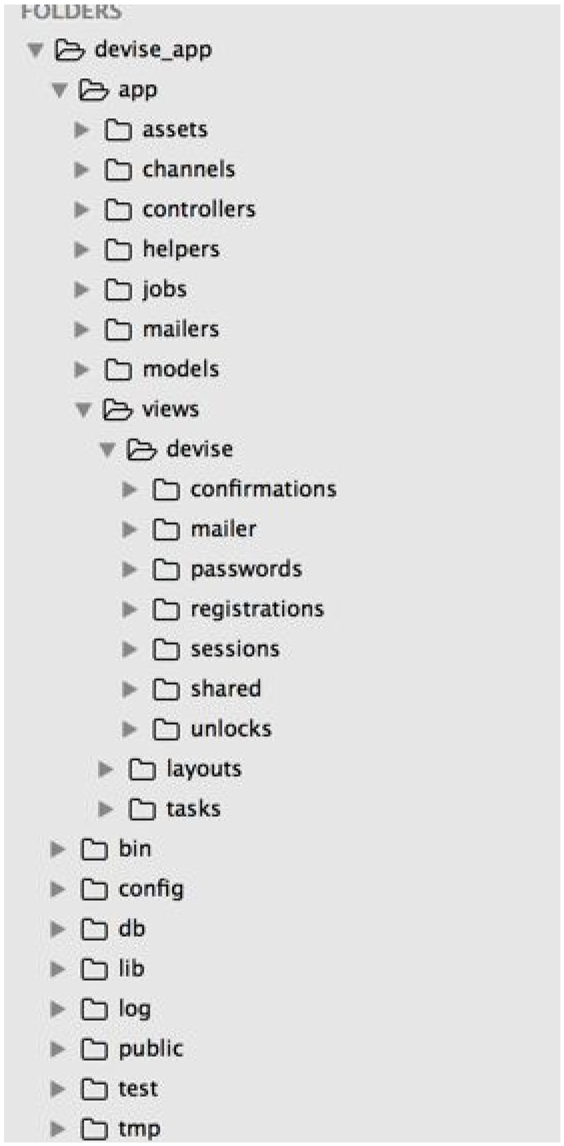
```
rails g migration add_organization_to_users organization_id:integer
```

and now let’s run **rails db:migrate** to migrate the database and add this field.

Now that we have an organization_id feild in our user model, we need to get something in it. Before we do that, take a moment and ask yourself when is an appropriate time to add an organization to a user. What might the workflow or ‘user story’ look like. An obvious time to do might be when a user first creates an account? So let's do that, when a user initially signs up for our app, in addition to the user's email and password, we will also ask them what name they would like for an organization. With all of the database pieces in place, let's modify our signup form in devise so the user can enter an organization when they sign up for an account. If we take a look at the file and folder structure of our app we will see that in apps/views there is not users folder as we might expect. By default devise does not make its views available to you, but not to worry, this is easily done with the following command:

```
rails generate devise:views
```

After running this command now you should have an apps/views/devise folder which will contain all the views associated with Devise.



```
FOLDERS
▼ devise_app
  ▼ app
    ▶ assets
    ▶ channels
    ▶ controllers
    ▶ helpers
    ▶ jobs
    ▶ mailers
    ▶ models
    ▼ views
      ▼ devise
        ▶ confirmations
        ▶ mailer
        ▶ passwords
        ▶ registrations
        ▶ sessions
        ▶ shared
        ▶ unlocks
      ▶ layouts
      ▶ tasks
  ▶ bin
  ▶ config
  ▶ db
  ▶ lib
  ▶ log
  ▶ public
  ▶ test
  ▶ tmp
```

For the purposes of adding a field called Organization to our form we will focus on the app/views/devise/registrations/new.html.erb file.

*might be good to go ahead and add the field and see it fail?

However, before we do this we need to consider this field's relationship to a user. Since this form contains fields associated with a user, we just can't cram a field from another model into it (in this case organization) and expect it to work. Well, we can..but not without a little work. To help us deal with this situation Rails gives use a method called accepts_nested_attributes_for, and it solves the problem we just stated. It allows us to save attributes related to associated records.

So before we modify our signup form let's modify our user and organization models. First, the User model to look like the following:

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

Then the Organization model should look like:

```
class Organization < ApplicationRecord
       has_many :users
       validates_uniqueness_of :name
end
```

We have done two things here. First we set up an association between the User and Organization models. We also don't want more than one Organization with the same name, so we can use `validates_uniqueness_of` validation to ensure the Organization name is unique. We did this with the addition of belongs_to :organization in the User model, and has_many :users in the Organization. Next we added accepts_nested_attributes_for :organization in our User model. This will allow us to save an organization_id field to a user when we enter an organization on our signup page. Speaking of, let's do that now!

Back in app/views/devise/registrations/new.html.erb let's add the following…

```
h2>Sign up</h2>
<% resource.organization ||= Organization.new %>
<%= form_for(resource, as: resource_name, url: registration_path(resource_name)) do |f| %>
  <%= devise_error_messages! %>

  <div class="field">
    <%= f.label :email %><br />
    <%= f.email_field :email, autofocus: true %>
  </div>
  <div class="field">
    <%= f.label :password %>
    <% if @minimum_password_length %>
    <em>(<%= @minimum_password_length %> characters minimum)</em>
    <% end %><br />
    <%= f.password_field :password, autocomplete: "off" %>
  </div>

  <div class="field">
    <%= f.label :password_confirmation %><br />
    <%= f.password_field :password_confirmation, autocomplete: "off" %>
  </div>
  <%= f.fields_for :organization do |org| %>
    <div><%= 'Organization or Company Name' %><br />
    <%= org.text_field :name %></div>
  <% end %>
  <div class="actions">
    <%= f.submit "Sign up" %>
  </div>
<% end %>

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

Note we added two things to the existing view.

```
<% resource.organization ||= Organization.new %>
```

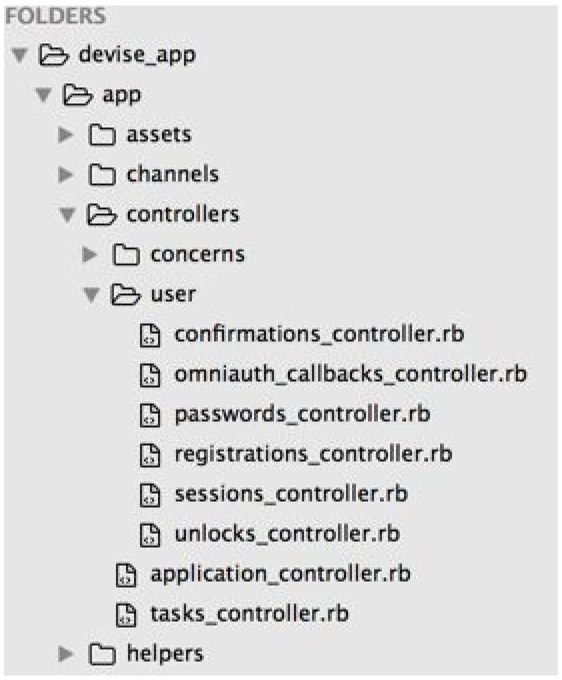
And

```
 <%= f.fields_for :organization do |org| %>
    <div><%= 'Organization or Company Name' %><br />
    <%= org.text_field :name %></div>
  <% end %>
```

The first code snippet creates a new organization in memory when the page loads. The second uses the fields_for method to referencing the organization model rather than the user (registration?). Now if we load our signup form we see the addition of the organization field. However, if we fill in the field and try to submit it we get an error. Not good! So let's fix it. The problem is we also need to modify our registrations controller that is associated with Devise. This is a problem because, like our views earlier in the chapter, these are not immediately available to us, but this is easy enough to fix. Again, similar to what we did for our views in devise, we can also do for controllers. Run the following command:

rails generate devise:controllers user (note user, not users?)

As we would expect this will generate controllers for us in the app/controllers/user folder.


```
FOLDERS
▼ devise_app
  ▼ app
    ▶ assets
    ▶ channels
    ▼ controllers
      ▶ concerns
      ▼ user
          confirmations_controller.rb
          omniauth_callbacks_controller.rb
          passwords_controller.rb
          registrations_controller.rb
          sessions_controller.rb
          unlocks_controller.rb
        application_controller.rb
        tasks_controller.rb
    ▶ helpers
```

If we open the registrations_controller devise gives us a pretty blank slate to work from, however there is quite a bit of commented out code there for reference.

Let's add the following code to the registrations_controller:
NOTE: Add ‘s’ to make class User -> Users?

```
class User::RegistrationsController < Devise::RegistrationsController
# before_action :configure_sign_up_params, only: [:create]
# before_action :configure_account_update_params, only: [:update]
before_action :update_sanitized_params, if: :devise_controller?

  # GET /resource/sign_up
  def new
    #   super
    @user = User.new
  end

  # POST /resource
  def create
    #   super
  @user = User.new sign_up_params
    if @user.save
      sign_up resource_name, @user #sign in user after sign_up
      redirect_to tasks_path
    else
      render action: 'new'
    end
  end


  ...

  protected

  def update_sanitized_params
    devise_parameter_sanitizer.permit(:sign_up, keys: [:organization_id, :role,
organization_attributes: [:name]])
  end

...
end
```

Now, one more thing, to get Devise to use this new generated controller we must change our routes.rb from this:

```
  devise_for :users
```

To this:

```
  devise_for :users, controllers: { :registrations => "user/registrations" }
```

Now when we sign up for a new account and include an organization we this in the console:

```
Processing by Users::RegistrationsController#create as HTML
  Parameters: {"utf8"=>"✓",
"authenticity_token"=>"UqCIiDjcI0qHMKV6drdbNpnyte2DveQQSCEGLwJW1GVWO/uQjSKT81qvgzc5Z2EoMtQfo4/
cWxvO8Mt9NGlI4Q==", "user"=>{"email"=>"mark@wintas.com", "password"=>"[FILTERED]",
"password_confirmation"=>"[FILTERED]", "organization_attributes"=>{"name"=>"wintas"}},
"commit"=>"Sign up"}
"parms are {\"email\"=>\"mark@wintas.com\", \"password\"=>\"test\",
\"password_confirmation\"=>\"test\", \"organization_attributes\"=>{\"name\"=>\"wintas\"}}"
   (0.2ms)  begin transaction
  User Exists (0.1ms)  SELECT  1 AS one FROM "users" WHERE "users"."email" = ? LIMIT ?
[["email", "mark@wintas.com"], ["LIMIT", 1]]
  SQL (0.3ms)  INSERT INTO "organizations" ("name", "created_at", "updated_at") VALUES (?, ?,
?)  [["name", "wintas"], ["created_at", 2016-08-10 15:15:19 UTC], ["updated_at", 2016-08-10
15:15:19 UTC]]
  SQL (0.1ms)  INSERT INTO "users" ("email", "encrypted_password", "created_at", "updated_at",
"organization_id") VALUES (?, ?, ?, ?, ?)  [["email", "mark@wintas.com"],
["encrypted_password", "$2a$11$DGTSlmNc2xxrvEbHat6kve76sRbamaI5Cw6RZmSp5LAdWWN3LfQG6"],
["created_at", 2016-08-10 15:15:19 UTC], ["updated_at", 2016-08-10 15:15:19 UTC],
["organization_id", 1]]
   (1.5ms)  commit transaction
```

We can see that our user is inserted into the DB with an appropriate organizaton_id, and also that corresponding Organization is also inserted into the Organization table. As additional sanity check we can go to the console and do something like…

```
2.3.0 :001 > User.last
  User Load (0.2ms)  SELECT  "users".* FROM "users" ORDER BY "users"."id" DESC LIMIT ?
[["LIMIT", 1]]
 => #<User id: 1, email: "mark@wintas.com", created_at: "2016-08-10 15:15:19", updated_at:
"2016-08-10 15:15:19", organization_id: 1>
2.3.0 :002 > Organization.last
  Organization Load (0.1ms)  SELECT  "organizations".* FROM "organizations" ORDER BY
"organizations"."id" DESC LIMIT ?  [["LIMIT", 1]]
 => #<Organization id: 1, name: "wintas", created_at: "2016-08-10 15:15:19", updated_at:
"2016-08-10 15:15:19">
```

...and we see both our user and org with the correct attributes associated with each. Ok, this is good. Each time a user signs up, they can specify an Organization. It might stand to reason that the first person who signs up for a particular organization will be the one who is going to manage it, so let's add an admin flag to this user one initial signup. We can take care of this in the database through a migration. At the console run:

```
rails g migration add_admin_to_users admin:boolean
```

We would not want this flag to be set to true unless we explicitly do so, so as a precautionary measure let's set the default to false by adding :boolean, default: false to the migration file. The entire migration should look like this:

```
class AddAdminToUsers < ActiveRecord::Migration[5.0]
  def change
    add_column :users, :admin, :boolean, default: false
  end
end
```

So now by default new users admin status will be set to false, however, we want new users who use the signup page to be admins for their new organization, so to make that happen will will add the following line to our registration controller's create action:

```
...
def create
    #   super
  @user = User.new sign_up_params
    @user.admin = true #first user to signup is admin
    if @user.save
      flash[:notice] = "Successfully created User."
      redirect_to tasks_path
    else
      render :action => 'new'
    end
  end
...
```

Pretty simple, no? It simply sets the admin flag on the @user object to true. Now when users sign up, the will automatically be admins.

Before we go any farther let's do a few UI housekeeping items to clean up our User Interface to make it easier to navigate the app.

First let's do the following to our application layout file:

```
<!DOCTYPE html>
<html>
  <head>
    <title>DeviseApp</title>
    <%= csrf_meta_tags %>

    <%= stylesheet_link_tag    'application', media: 'all', 'data-turbolinks-track': 'reload'
%>
    <%= javascript_include_tag 'application', 'data-turbolinks-track': 'reload' %>
  </head>

 <body>
 <% flash.each do |name, msg| %>
    <%= content_tag :section, msg, :id => "flash_#{name}", :class => "flash" %>
  <% end %>

     <div id="container">
              <span style="text-align: right">
             <div id="user_nav">
               <% if user_signed_in? %>
                 Signed in as <%= current_user.email %>(<%= current_user.organization.name %>)
                 <%= link_to "Sign out", destroy_user_session_path %>
               <% else %>
                 <%= link_to "Sign up", new_user_registration_path %> or <%= link_to "sign
in", new_user_session_path %>
               <% end %>
             </div>
           </span>
         </div>
    <%= yield %>
  </body>
</html>
```

Now login to the app and you should see a menu item in the upper right hand corner of the app that displays the logged in user as well as the organization they are associated with.

Filtering Tasks

Now that we have some our User and Organization relationship firmly in place, let's begin looking at Tasks, and how we want to associate them to Users and Organizations. Take a moment and think about that. Here are a few questions you might ask yourself:

- Who should "own" a task? A User or an Organization?
- What are the pros and cons of either of these entities being associated with a Task?
- What is easiest to implement (write code for)?
- What is more efficient? How quickly can if find records if there are a 1,000 Tasks? What about 1,000,000 Tasks?

I am not necessarily going to address each of the questions above in this book, but these should be some of the questions that come to mind at this point in the development process. That being said, it is important not to get too bogged down with these kinds of questions either. There is danger of "paralysis by analysis", and while we should certainly do our due diligence there is such a thing as re-factoring. If we find that a particular query to architecture doesn't work down the road due high demand, then it can always be changed. Besides, is the South we call high demand is what I call a High Class problem!

First let's address the questions of who should "own" a task. Our first inclination might be to have an Organization own a task, and in this case we would add an organization_id attribute to our task model. At first take this might make sense. However, I am going to propose that we instead have a User own a task, that is we will add a user_id field to a task when it is created. If we do this, in addition to associating a user to a task, we also get the added benefit of being able to get the Organization's relationship to a task "through" the user. So we might say that a user association has greater flexibility and benefit over an Organization association. As an example, let's say we add users to our organization, our users should be able to view tasks that are associated with that organization. However, let's think more granularly. What if we want the admin users in our organization to see all tasks, but non-admin users to see only tasks they created. If that is a requirement, and in this case, I am saying it is, then only have an organization_id as part of a task won't cut it. OK, hopefully I have convinced you that adding a user_id field to a task is the way to go, so let's write to code to make it happen!

First, let's take care of the database. Let's create the following migration to add a user_id field to our tasks:

```
rails g migration add_user_to_tasks user_id:integer
```

This creates a migration file for us named add_user_to_tasks and adds a user_id field of type integer to our tasks table. Don't forget to run rails db:migrate afterwards.

Generally we would set up the associations in our models before going to the controller, but rather than doing that let's jump straight to the controller so I can show a sort of manual way of setting up this association, then we will go back and make the code a bit cleaner.

In the create action of your tasks_controller add the following code:

```
  def create
    @task = Task.new(task_params)
    @task.user_id = current_user.id
    respond_to do |format|
      if @task.save
        format.html { redirect_to @task, notice: 'Task was successfully created.' }
        format.json { render :show, status: :created, location: @task }
      else
        format.html { render :new }
        format.json { render json: @task.errors, status: :unprocessable_entity }
      end
    end
  end
```

Note we added the line **`@task.user_id = current_user.id`**. This does just what it says. Just after we create a new Task with the task_params, and before we save the Tasks, but set the user_id field to the current_user id. In fact, if we create a new task in our app, we see this work perfectly fine as evidenced by our console output:

```
 SQL (0.7ms)  INSERT INTO "tasks" ("name", "created_at", "updated_at", "user_id") VALUES (?,
?, ?, ?)  [["name", "Register Students"], ["created_at", 2016-08-11 15:56:50 UTC],
["updated_at", 2016-08-11 15:56:50 UTC], ["user_id", 3]]
```

Yep, our task has a user_id, 3 in this case. While this works, its not really the 'rails way' and it feels a bit hackish, but I wanted to do it just to take some of the Rails magic out of it. OK, now let's do it the right way, or at least a way that's a bit cleaner.

First, configure the association between a task and a user. We will add the following to our task and user models:

*maybe investigate using accepts_nested_attributes_for to get the user_id association

```
class Task < ApplicationRecord
       belongs_to :user
end
```

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

We will add a few database related goodies to this association later, but right now this is very basic has_many belongs_to association in rails. Now, lets go back to our controller and use some of the methods and dot notation this now available to us base on this new relationship we have added to our models. In the controller modify it to look like this…

```
def create
    @task = Task.new(task_params)
    @task.user = current_user
    respond_to do |format|
      if @task.save
        format.html { redirect_to @task, notice: 'Task was successfully created.' }
        format.json { render :show, status: :created, location: @task }
      else
        format.html { render :new }
        format.json { render json: @task.errors, status: :unprocessable_entity }
      end
    end
  end
```

So this is not terrible different. In fact all we did was change `@task.user_id = current_user.id` to `@task.user = current_user`. Not at all earth shattering, but if we would have tried to use @task.user with out the model associations (go ahead and try it) we would have gotten an error that looked something like:

undefined method `user=' for #<Task:0x007ff032cad8c8> Did you mean? user_id=

In fact Rails is smart enough to give us the suggestion of using user_id which we were previously using. The point is that a big part of what the model associations do is give you handy methods like @task.user, and later on we'll do something like @task.user.organization. These methods are nice because they allow you to 'walk' though the database and get peices of data along the way. We can also use these in the view. Now that we are associating a taks to a user we might as well display that in the index view of the task. Go to tasks/index.html.erb and add the following code:

```
<p id="notice"><%= notice %></p>

<h1>Tasks</h1>

<table>
  <thead>
    <tr>
      <th>Name</th>
      <th colspan="3"></th>
    </tr>
  </thead>

  <tbody>
    <% @tasks.each do |task| %>
      <tr>
        <td><%= task.name %>(<%= task.user.email %>)</td>
        <td><%= link_to 'Show', task %></td>
        <td><%= link_to 'Edit', edit_task_path(task) %></td>
        <td><%= link_to 'Destroy', task, method: :delete, data: { confirm: 'Are you sure?' }
%></td>
      </tr>
    <% end %>
  </tbody>
</table>

<br>

<%= link_to 'New Task', new_task_path %>
```

Note, we added task.user.email next to our list of tasks in the index view. Very handy!

*Note, you may need to reset your database if you have users that do not of a user associated with them. A quick way to do this is to run ‘rails db:reset’ at the command prompt. However, be advised, you will loose any data in your development database when you do this.

We have make good progress, but if we create a couple of users and create tasks under each of those users accounts, all tasks are still being displayed regardless of who is logged in. Let’s fix that now!

Filtering tasks is surprisingly easy, in fact we can do it by chaning a single line in our tasks_cotroller. In our index action change:

```
def index
```

```
    @tasks = Task.all
end
```

To

```
def index
    @tasks = current_user.tasks
end
```

Now create a couple of tasks under different users and check out the results. The ability to call current_user.tasks is all based on our associations we set up in our user and task models. Lets look at it in the console:

```
2.3.0 :005 > u=User.first
  User Load (0.3ms)  SELECT  "users".* FROM "users" ORDER BY "users"."id" ASC LIMIT ?
[["LIMIT", 1]]
 => #<User id: 1, email: "mark@wintas.com", created_at: "2016-08-11 16:16:42", updated_at:
"2016-08-12 14:17:23", organization_id: 1, admin: true>
2.3.0 :006 > pp u.tasks
  Task Load (0.1ms)  SELECT "tasks".* FROM "tasks" WHERE "tasks"."user_id" = ?  [["user_id",
1]]
[#<Task:0x007fcf3980add0
  id: 1,
  name: "test",
  created_at: Thu, 11 Aug 2016 16:16:53 UTC +00:00,
  updated_at: Thu, 11 Aug 2016 16:16:53 UTC +00:00,
  user_id: 1>,
 #<Task:0x007fcf3980aba0
  id: 3,
  name: "Buy Grains",
  created_at: Fri, 12 Aug 2016 14:23:01 UTC +00:00,
  updated_at: Fri, 12 Aug 2016 14:23:01 UTC +00:00,
  user_id: 1>]
 => #<ActiveRecord::Associations::CollectionProxy [#<Task id: 1, name: "test", created_at:
"2016-08-11 16:16:53", updated_at: "2016-08-11 16:16:53", user_id: 1>, #<Task id: 3, name:
"Buy Grains", created_at: "2016-08-12 14:23:01", updated_at: "2016-08-12 14:23:01", user_id:
1>]>
```

In the console I am assigning u to a user, but it works the same as current_user in our app. Current_user is a convenience method call that devise gives us to reference the currently logged in user.

This is great, but let's not pat ourselves on the back to quickly. While we are filtering tasks in the index action, we aren’t doing it anywhere else, namely the show action. So, for example if user calvin has task/1 and user daryl has task/2, I can login as user calvin, then go to the address bar in the web browser and type in task/2, and voila, I can view that task. Not good! So we also need do some filtering in the show action. This is also not terribly difficult. Add this to your show action in the tasks_controller:

```
def show
  if @task.user_id == current_user.id
    respond_to do |format|
      format.html # show.html.erb
      format.xml  { render :xml => @patient }
    end
  else
    flash[:notice] = "Unauthorized Page View"
    redirect_to(tasks_url)
  end
end
```

Dead simple, no? A simple if conditional asks if the task's user_id matches the current user's id, and if so, then we display it, if not, then we redirect the user to the task index page (tasks_url) along with a flash message. So now go back and try and change the URL to view another user's task. No cigar!

If you are out and about on the internet give this a try. There are so many restful frameworks around these days, and you can usually spot a restful website when you see one. If you are on a page that looks to be associated with data and records try modifying the URL, you may be surprised at what you find. Be sure and try things like /edit and /new. Speaking of which we have just discovered another bug in our app! A user can do what we just described using /edit to view (and even worse) edit another user's task. Again the fix is pretty easy, in fact it is the same code we used in the show action. But wait, we can see this is getting pretty tedious, modifying each of these tasks this way. Wouldn't it be nice if there was some place we could put this code in our controller and just call it once each time the controller is called? Well, there is, down in the set_task method of the controller. Go ahead and remove the conditional from the show and edit actions, and put it in the set_task method of the task_controller:

```
private
  # Use callbacks to share common setup or constraints between actions.
  def set_task
    @task = Task.find(params[:id])
    if @task.user_id == current_user.id
    respond_to do |format|
      format.html # show.html.erb
      format.xml  { render :xml => @patient }
    end
    else
      flash[:notice] = "Unauthorized Page View"
      redirect_to(tasks_url)
    end
  end
```

Ahh! Nice and DRY (Do Not Repeat Yourself). So now this will apply to all of our tasks. That is, every time the tasks_controller is called, the set_task method is called for the show, edit, update, and destroy actions via the before_action at the top of the controller, and we will ensure that any tasks user_id field will match the current_users id.

Testing

Before going any further, let's add some tests.

*Note: add a comment box here that includes by testing philosophy

First add the rspec and capybara gems to your Gemfile and then run bundle to install them:

```
...
group :development, :test do
  # Call 'byebug' anywhere in the code to stop execution and get a debugger console
  gem 'byebug', platform: :mri
  gem 'rspec-rails'
  gem 'capybara'
end
...
```

Note we added these gems to our development and test group so they are not included in production. Now run rspec's installer:

```
marklocklear$ rails g rspec:install
Running via Spring preloader in process 72696
      create  .rspec
      create  spec
      create  spec/spec_helper.rb
      create  spec/rails_helper.rb
```

Capybara is great because it allows us to do high level testing in a headless browser As a former manual tester, I love this kind of testing because you can have all the unit tests you want, but there is nothing like go ole' fashion user interaction with your app to test all layers of its functionality. In rspec feature specs default to the spec/features folder, so we'll need to create that folder, in fact do that now in either your text editor, or at the console with:

```
marklocklear$ mkdir spec/features
```

Now lets create our first test, or 'spec'. In spec/features created a file called sign_up_spec.rb with the following code:

```
require "rails_helper"
feature "Accounts" do
	scenario "creating an account and sign in" do
		visit new_user_registration_path
		fill_in "Email", with: "marklocklear@gmail.com"
		fill_in "Password", with: "wintas"
		fill_in "Password confirmation", with: "wintas"
		fill_in "user_organization_attributes_name", with: "Locklear Inc."
		click_button "Sign up"
		expect(page).to have_content("You need to sign in or sign up before continuing")
		visit new_user_session_path
		fill_in "Email", with: "marklocklear@gmail.com"
		fill_in "Password", with: "wintas"
		click_button "Log in"
		expect(page).to have_content("Signed in successfully")
	end
end
```

Before we talk more about exactly what is going on here go ahead and run **rake spec** and you should see something like this:

```
ruby-2.3.0 marklocklear:devise_app marklocklear$ rake spec
/Users/marklocklear/.rvm/rubies/ruby-2.3.0/bin/ruby
-I/Users/marklocklear/.rvm/gems/ruby-2.3.0/gems/rspec-core-3.5.2/lib:/Users/marklocklear/.rvm/
gems/ruby-2.3.0/gems/rspec-support-3.5.0/lib
/Users/marklocklear/.rvm/gems/ruby-2.3.0/gems/rspec-core-3.5.2/exe/rspec --pattern
spec/\*\*\{,/\*/\*\*\}/\*_spec.rb
.

Finished in 0.31103 seconds (files took 1.45 seconds to load)
1 example, 0 failures
```

There are a few things we want to look at here, so let's look at them one at a time. First, we require the rails_helper file. That should look familiar from when we ran the installer earlier. If you look at the file in spec/rails_helper.rb you can see it has a ton of configuration settings for rspec. Generally you can leave the defaults in place for a typical Rails app as we will do here, but for more sophisticated apps, this is where you would modify configuration settings along with the spec/spec_helper.rb file.

Next, notice the general structure of this test. This is true for most tests in rspec, they include a feature block and a scenario block. We'll add more scenarios to this feature as we go along in the book. The commands in the scenario block are the meat of our tests. This is where we mimic user interaction with the app, in this case we are going to sign up for a new account. How do we do that? Well, first we need to visit the sign_up page, where is that? If we open our app and like the sign_up like we can see that the URL for that page is /auth/users/sign_up. The way we reference that path is the use the URI pattern that Rails gives us. The best way to find that is to go to the console and type rails routes:

```
marklocklear$ rails routes
                  Prefix Verb   URI Pattern                          Controller#Action
        new_user_session GET    /auth/users/sign_in(.:format)        devise/sessions#new
            user_session POST   /auth/users/sign_in(.:format)        devise/sessions#create
    destroy_user_session GET    /auth/users/sign_out(.:format)       devise/sessions#destroy
           user_password POST   /auth/users/password(.:format)       devise/passwords#create
       new_user_password GET    /auth/users/password/new(.:format)   devise/passwords#new
      edit_user_password GET    /auth/users/password/edit(.:format)  devise/passwords#edit
                         PATCH  /auth/users/password(.:format)       devise/passwords#update
                         PUT    /auth/users/password(.:format)       devise/passwords#update
cancel_user_registration GET    /auth/users/cancel(.:format)         users/registrations#cancel
       user_registration POST   /auth/users(.:format)                users/registrations#create
   new_user_registration GET    /auth/users/sign_up(.:format)        users/registrations#new
  edit_user_registration GET    /auth/users/edit(.:format)           users/registrations#edit
                         PATCH  /auth/users(.:format)                users/registrations#update
                         PUT    /auth/users(.:format)                users/registrations#update
                   tasks GET    /tasks(.:format)                     tasks#index
                         POST   /tasks(.:format)                     tasks#create
                new_task GET    /tasks/new(.:format)                 tasks#new
               edit_task GET    /tasks/:id/edit(.:format)            tasks#edit
                    task GET    /tasks/:id(.:format)                 tasks#show
                         PATCH  /tasks/:id(.:format)                 tasks#update
                         PUT    /tasks/:id(.:format)                 tasks#update
                         DELETE /tasks/:id(.:format)                 tasks#destroy
                   users GET    /users(.:format)                     users#index
                         POST   /users(.:format)                     users#create
                new_user GET    /users/new(.:format)                 users#new
               edit_user GET    /users/:id/edit(.:format)            users#edit
                    user GET    /users/:id(.:format)                 users#show
                         PATCH  /users/:id(.:format)                 users#update
                         PUT    /users/:id(.:format)                 users#update
                         DELETE /users/:id(.:format)                 users#destroy
                    root GET    /                                    tasks#index
```

If we look for that URI above we see the prefix for it is **`new_user_registration`**. To reference that path all we need to do is add _path to the end of identifier and that will allow the test to access and load the page, just like a user would. So when we call `visit new_user_registration_path` our test is loading this pages view. This is followed by fill_in methods that capybara gives us access to. The fill_in method does just what it says. Its fills in a particular field in the view (that is the first argument) with the text that is passed into the second argument. After filling in the email, password, Password confirmation and Organization fields the next thing we want to do is submit the form. To do that we click (click_button) the Signup button. The final thing we want to do is verify that the user successfully signed up. The best way to do that is check for some content on the page that displays on the page after a user signs up. Go ahead and do that now in you app. Sign up for an account and look at the page that loads after a user signs up. What are some things (text?) on the page that might be good candidates for verifying that the user successfully signed up for an account? One thing might be the flash message that is displayed. If you look carefully, after user sign up notice the flash message “Successfully created User” is being displayed. So do do this we can use respecs expect method:

```
expect(page).to have_content("Successfully created User")
```

This is pretty straightforward, and reads nicely! If we were reading this out loud me might say “Expect the page to have the content ‘Successfully created User’”. While we're at let's add one more check for something specific to the information we submitted on the previous page:

```
expect(page).to have_content("Signed in as marklocklear@gmail.com")
```

This line fits that bill by checking for the specific email address that we signed up for an account with.

TODO: Add more tests for creating multiple users with multiple tasks, and ensure users can’t see other users tasks.

User Dashboard

As our app currently stands all our users are admins, so we can’t even begin to address our requirement of filtering tasks in different ways for admins and non admins. So to do that let's go ahead and dive into creating a user dashboard that will allow our admin users to create other users.

Before we go further let's remind ourselves what it is we are trying to accomplish. As the app currently stands, users can create a user account using devise. When they do this they are also creating an organization that is associated with that account. During this process we are also making that user an admin (setting the admin flag to true), in effect making them an admin of the organization that they create when they sign up. There is no use in being the admin of an Organization if you can *run thangs*, so now we are going to give that user the ability to create other users in their Organization. Additionally, we will be able to create both admin and non-admin users. At that point we will look at filtering tasks based on the admin/non-admin user status. Let's get started!

Our Dashboard won’t need a model (no database related CRUD actions) so we generate a controller with a single index view using this command:

```
rails g controller dashboard index
```

Let’s address the controller first:

```
class DashboardController < ApplicationController
```

```
    before_action :authenticate_user!
    def index
       if current_user.admin?
       @users = current_user.organization.users
    else
      flash[:notice] = "Unauthorized Page View"
      redirect_to(tasks_url)
    end
  end
end
```

First, we want to be sure a user is logged in before accessing this controller (and its associated views) so we add the authenticate_user before action. Next, in the index action we first check to see if the user is an admin, we then call organization.users method. This `organization.users` method will retrieve all user in the user's Organization. We can see this in the rails console with the following:

```
2.3.0 :001 > u=User.last
  User Load (0.2ms)  SELECT  "users".* FROM "users" ORDER BY "users"."id" DESC LIMIT ?
[["LIMIT", 1]]
 => #<User id: 3, email: "admin@abtech.edu", created_at: "2016-08-12 17:19:14", updated_at:
"2016-08-12 17:19:45", organization_id: 2, admin: false>
2.3.0 :002 > u.organization
  Organization Load (0.2ms)  SELECT  "organizations".* FROM "organizations" WHERE
"organizations"."id" = ? LIMIT ?  [["id", 2], ["LIMIT", 1]]
 => #<Organization id: 2, name: "Speech N Progress", created_at: "2016-08-12 14:16:03",
updated_at: "2016-08-12 14:16:03">
2.3.0 :003 > u.organization.users
  User Load (0.2ms)  SELECT "users".* FROM "users" WHERE "users"."organization_id" = ?
[["organization_id", 2]]
 => #<ActiveRecord::Associations::CollectionProxy [#<User id: 2, email: "faline@tsc.om",
created_at: "2016-08-12 14:16:03", updated_at: "2016-08-12 17:31:37", organization_id: 2,
admin: true>, #<User id: 3, email: "admin@abtech.edu", created_at: "2016-08-12 17:19:14",
updated_at: "2016-08-12 17:19:45", organization_id: 2, admin: false>]>
```

Now we will add the following to our dashboards index view:

```
<h2>Users</h2>
<% @users.each do |u| %>
  <p><%= u.email %> <%= u.admin ? "(admin)" : "" %>
     <%= link_to "Edit User", edit_user_path(u) %>
      <% unless u.email == current_user.email %>
        <%= link_to "Delete User", u, method: :delete, data: { confirm: 'Are you sure?' }  %>
      <% end %>
<% end %>
</p>
<%= link_to 'New User', new_user_path %><p/>
<%= link_to 'Tasks',root_path %>
```

This view should be pretty similar to other Rails views you have seen. Remember in our Dashboard Controller we set the @users variable to the current users, organizations users (say that five times fast) so not in the view this @users.each block is going to loop through each of those users and display the users admin status (whether they are admin or not), their email address, and a link delete the user (unless its the current user).

Next we will need a controller for our users. Why not use the controller in devise? I have no idea! TODO: Explain why we use a separate controller and not devise.

```
rails g controller users
```

Now let's add the following to our users_controller:

```
class UsersController < ApplicationController
  before_action :set_user, only: [:edit, :update, :destroy]
  before_action :authenticate_user!

       def new
              @user = User.new
       end

       def create
    @user = User.new(user_params)
    @user.organization = current_user.organization
    if @user.save
      flash[:notice] = "Successfully created User."
      redirect_to dashboard_index_path
    else
      render :action => 'new'
    end
  end

       def edit
    #TODO only admins of current_organization should be able to edit users in that
organization
       end

  def update
    params[:user].delete(:password) if params[:user][:password].blank?
    params[:user].delete(:password_confirmation) if params[:user][:password].blank? and
params[:user][:password_confirmation].blank?
    if @user.update_attributes(user_params)
      flash[:notice] = "Successfully updated User."
      redirect_to dashboard_index_path
    else
      render :action => 'edit'
    end
  end

  def destroy
    @user = User.find(params[:id])
    if @user.destroy
      flash[:notice] = "Successfully deleted User."
      redirect_to dashboard_index_path
    end
  end

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

    # Never trust parameters from the scary internet, only allow the white list through.
  def user_params
    params.require(:user).permit(:email, :password, :password_confirmation,
        :organization_id, :admin)
  end

end
```

Now we need to modify our routes with the following:

```
Rails.application.routes.draw do
  devise_for :users, path_prefix: 'auth',
        :controllers => { :registrations => "users/registrations" }
  resources :tasks
  resources :dashboard
  resources :users
  # For details on the DSL available within this file, see
http://guides.rubyonrails.org/routing.html
  root "tasks#index"
end
```

We are doing two important things here. First, we are modifying our devise routes, and adding the prefix /auth to all of the devise routes. We are doing this because we added a users controller, which will not use the /users routes. Without adding the /auth to the devise routes, the two controller paths would collide. The other thing we did was add resources :users.

Finally, lets add views for creating new users in the dashboard. These views will live in app/views/users:

app/views/users/new.html.erb

```
<%= form_for @user, :url => users_path do |f| %>
  <%= render :partial => 'form', :locals => { :f => f } %>
```

```
<% end %>
```

app/views/users/edit.html.erb

```
<%= form_for @user, url: user_path do |f| %>
  <%= render partial: 'form', :locals => { :f => f } %>
<% end %>
```

app/views/users/_form.html.erb

```
<p><%= f.label :email %><br />
<%= f.text_field :email %></p>

<p><%= f.label :password %><br />
<%= f.password_field :password %></p>

<p><%= f.label :password_confirmation %><br />
<%= f.password_field :password_confirmation %></p>

<p><%= f.label :admin %><br />
<%= f.check_box :admin %></p>

<p><%= f.submit "Submit" %></p>
```

So now when we login we can go to /dashboard and we are able to add users to our organization. A couple of things to note here. First notice there is not an organization field. This makes sense, because remember, the whole point is we want to add user to our current organization, and in fact if we create a user from /dashboard we see something like this in the console:

```
Started POST "/users" for ::1 at 2016-08-12 14:16:47 -0400
Processing by UsersController#create as HTML
  Parameters: {"utf8"=>"✓",
"authenticity_token"=>"0senlY/AJPYV7gTCHSzkUEqHdY5lxEC03UYvvH8brYIG/5qLQG3HzlBbQzRkG6l8RjX19f2
srMsdLZK3d3B+ag==", "user"=>{"email"=>"flo@progressive.com", "password"=>"[FILTERED]",
"password_confirmation"=>"[FILTERED]", "admin"=>"1"}, "commit"=>"Submit"}
  User Load (0.2ms)  SELECT  "users".* FROM "users" WHERE "users"."id" = ? ORDER BY
"users"."id" ASC LIMIT ?  [["id", 2], ["LIMIT", 1]]
  Organization Load (0.1ms)  SELECT  "organizations".* FROM "organizations" WHERE
"organizations"."id" = ? LIMIT ?  [["id", 2], ["LIMIT", 1]]
   (0.0ms)  begin transaction
  Organization Exists (0.1ms)  SELECT  1 AS one FROM "organizations" WHERE
"organizations"."name" = ? AND ("organizations"."id" != ?) LIMIT ?  [["name", "Speech N
Progress"], ["id", 2], ["LIMIT", 1]]
  User Exists (0.1ms)  SELECT  1 AS one FROM "users" WHERE "users"."email" = ? LIMIT ?
[["email", "flo@progressive.com"], ["LIMIT", 1]]
```

```
  SQL (0.3ms)  INSERT INTO "users" ("email", "encrypted_password", "created_at", "updated_at",
"organization_id", "admin") VALUES (?, ?, ?, ?, ?, ?)  [["email", "flo@progressive.com"],
["encrypted_password", "$2a$11$lvSpOkn7bobK4N62nAIHceHTbSAv2Nd1.H4XPjPdT6cE65k4PHBbK"],
["created_at", 2016-08-12 18:16:47 UTC], ["updated_at", 2016-08-12 18:16:47 UTC],
["organization_id", 2], ["admin", true]]
   (0.8ms)  commit transaction
Redirected to http://localhost:3000/dashboard
Completed 302 Found in 122ms (ActiveRecord: 1.6ms)
```

All looks well! Our user is being inserted with the appropriate organization_id and also with the appropriate admin flag; in the case above I checked the admin box to ensure Flo is an admin. Also, if we restart our server and go to localhost:3000, when we get to login or signup we see routes that look like /auth/users/sign_in and /auth/users/sign_up as opposed to working with users in our /dashboard console, you will notice the routes are /users; without the /auth prefix.

Filter tasks by admin status

Now, as admins, we have the ability to create other users in our organization, lets complete the final part of our requirement and filter tasks based on whether or not the user has an admin flag. A couple of User Stories for this requirement might look like this:

> User Calvin is an admin user in our application. When he logs into the app, he should be able to see all tasks created by all users within our organization
>
> User Flossie is a non-admin user. When she logs into the app she should only see tasks created by her. She should not see any other tasks.

First, take a moment and ask yourself, where do you think we'll look to in the app to filter our tasks? If you recall, earlier in the chapter we did this in the tasks_controller, so lets go back there and make a couple of changes.

In the index action of our tasks_controller make the following change:

```
def index
    if current_user.admin
      @tasks = current_user.organization.tasks
    else
      @tasks = current_user.tasks
    end
 end
```

OK, this is really straightforward. We do a simple check to see if the current user is an admin, and if so we display all of the tasks in the user's organization. However, if you add this code to the tasks_controller we get an error undefined method `tasks'. This is because the method ‘tasks’ is not available to organization. The reason we get this error is that we have not yet set up the association between a Task and an Organization. Let do that now:

```
class Task < ApplicationRecord
       belongs_to :user, dependent: :destroy
       has_one :organization, through: :user
end

class Organization < ApplicationRecord
       has_many :users
       has_many :tasks, through: :users
end
```

Here in our Task model we added the relationship `has_one :organization, through: :user` and in our Organization model we added `has_many :tasks, through: :users`. Notice we used the ‘through’ keyword here. Since there is not a direct relationship between a Task and a Organization we could not use the standard belongs_to and has_one relationship. However, because there is a relationship between tasks and users we are able to establish a connection to a task and an organization through that relationship.

Now if we login as out Admin user, we should see all tasks in our organization, and if we login as our non-admin user, we can only see tasks that that user created.