**How to Create a Library App with Ruby on Rails**

**Installing Development Tools on Mac**

**# Installfest Step 1: Development Environment Setup**

**## Operating System & Command Line Tools**

If you don't already have an account for the Mac App Store, follow the instructions on Apple Support to [create a Mac App Store account](https://support.apple.com/kb/PH11499?locale=en\_US).

Before class starts, we suggest you upgrade your operating system to OS X Yosemite (10.10), OS X El Capitan (10.11), or macOS Sierra. If you're on an older machine with 4GB or less of memory, please stick to OS X Yosemite. Also, if Apple releases a newer version of macOS while you're in WDI; please don't update until your instructors say it's ok.

To check which operating system you're running:

1. Click the apple icon in the top left of your computer screen.

2. Select "About This Mac" from the dropdown menu.

3. Read the version information from the window that pops up.

If you are not using OS X Yosemite (10.10), OS X El Capitan (10.11), or macOS Sierra, detailed instructions for upgrading your operating system are available through Apple support: [How to upgrade to macOS Sierra](https://support.apple.com/macos)

> Please let an instructor know if you're using an older version of OS X or if your system has less than 2 GB of memory.

**### Install Command Line Tools from the Terminal**

1. Open the Terminal application.

2. In your Terminal, type `xcode-select --install`, and a new window and installer will appear.

3. Follow the instructions in the installer.

### Install Command Line Tools through Xcode (Older Versions of OS X)

\*\*Only follow these steps if you were not able to install Xcode Command Line Tools with the instructions above.\*\* If you must run an older version of OS X, you will need to install Command Line Tools that come from Xcode.

1. Open the Mac App Store and install Xcode.

2. Open Xcode.

3. Inside the Xcode menu, choose Preferences > Downloads > Install The Command Line Tools.

4. Follow the instructions in the installer.

**## Homebrew**

\_\_Note: when copying the code snippets, please exclude the `$` as you paste and run the code into your terminal. The dollar sign `$` is simply an indicator of the logged-in user in examples.\_\_

[Homebrew](http://brew.sh) is a \*package manager\* - it downloads and updates programs on your machine. We'll use it to quickly download and install other tools we need, or to update already installed tools.

1. Open the Terminal application, and run `which brew` to check if you have Homebrew installed already. The `which` Terminal command shows where on your computer a program is installed. If it is installed, the Terminal will output a file path. If it is not installed, the Terminal won't output anything.

2. \*\*Only if you do not have Homebrew installed\*\*, run the command below to install Homebrew. Wait while Homebrew downloads and installs.

```bash

$ ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"

```

If you run into problems, you may need to run `rm -rf /usr/local/Cellar /usr/local/.git` and then retry the command above.

3. Run `brew update` to update Homebrew.

4. Run `brew doctor` in your Terminal to check that Homebrew and any current packages are installed correctly. If there are issues, `brew doctor` will list suggestions for how to fix them. Follow these suggestions one by one. If you're not sure what to do, \*\*ask!\*\*

5. Based on the errors in the step above, you may need to edit your `~/.bash\_profile` to include the path to Homebrew if `brew doctor` shows warnings. If in doubt ask for help here.

```bash

$ echo 'export PATH="/usr/local/bin:/usr/local/sbin:~/bin:$PATH"' >> ~/.bash\_profile

```

6. Let's install our first package with Homebrew, `tree`! This package adds a command to your Terminal that displays files in a tree view (instead of a list view like `ls`). Enter the following command in your Terminal:

```bash

$ tree

```

It will tell you that this command is not found or it just won't do anything. This is because it's not a program (command) that's installed on your machine. Now install it with this command:

```bash

$ brew install tree

```

7. Now run the Terminal command `tree` to see a tree view of all the files inside your current directory!

>If you run `tree` from your root directory, it might be printing files for a LONG time! Remember that you can always use `ctrl + C` in the terminal to stop the currently running process.

8. `trash` is another Homebrew package that easily allows one to move files or folders into the trash bin from the terminal. This is good practice because [other commands](http://docstore.mik.ua/orelly/unix3/upt/ch14\_03.htm) may be more dangerious. Again, the `trash` tool does not permanently delete files or folders, but simply places them in the trash. Let's install it using homebrew with the command:

```bash

$ brew install trash

```

We can test this by creating a file and moving our dummy file into our trash bin. Run these lines in your terminal to see the power of `trash`:

```bash

$ touch new-trash-file.txt

$ ls

```

You should see the newly created file.

```

$ trash new-trash-file.txt

$ ls

```

Now it's moved to the trash bin!

**## Customization**

**### Option as Meta Key**

Let's change a setting in the Terminal that will allow us to move through and delete entire words at a time by holding down the `option` key. This the default behavior of most text editors and will allow us to make more productive edits to our bash commands.

1. Navigate back to `Terminal` -> `Preferences` -> `Profile`.

2. Navigate to the the sub-tab, `Keyboard`

3. Check the input, at the bottom, that displays "Use Option as Meta key".

Try typing something in the Terminal then using the arrows to navigate through the text; now hold the `option` key at the same time to see the cursor jump from one word to the next.

**## Git**

You should already have git installed and have an account on GitHub from Fundamentals. If not, sign up for an account on [github.com](http://github.com). We'll be using GitHub to track code changes and collaborate on projects.

**### Confirm Install**

1. To check whether git is installed on your system, run the Terminal command `which git`. The output should be a directory path like `/usr/bin/git`. This is where git is installed on your machine. If you don't see any output, git is not installed on your computer.

2. \*\*Only if you do not have git installed\*\*, run the following command in your Terminal:

```bash

$ brew install git

```

**### Configure Git**

Configuring your git settings will help GitHub track your contributions and to make it easier and smoother to commit changes.

1. Use the following three `git config` commands to configure your git user information and have git "cache" (remember) it. We use the `--global` (or `-g`) option to make the configuration apply to all repositories.

```bash

$ git config --global user.name "YOUR\_GITHUB\_USERNAME"

$ git config --global user.email "YOUR\_GITHUB\_EMAIL\_ADDRESS"

$ git config --global credential.helper cache

```

2. Generate a SSH key for GitHub by [following GitHub's instructions](https://help.github.com/articles/generating-ssh-keys). This will allow you to use GitHub from your Terminal without entering your login information every time you push.

Clarifying notes for GitHub's instructions that might be confusing:

1. When you are "Adding a new SSH key to your GitHub Account" the command:

```bash

$ pbcopy < ~/.ssh/id\_rsa.pub

```

will take your SSH key that was saved to the file id\_rsa.pub and copy it to your clipboard (similar to using Command-C, but with the command line).

**# Installfest Step 1.1: Node.js & Express Stack**

Most of the coding work we do in weeks 3 to 5 will be driven by the back-end web development framework [Express](http://expressjs.com). We'll install Express individually in each project we create. For now, we'll install the other tools we'll use along with Express.

**#### Plan Overview**

1. Install Node.js, a platform for back-end web development with the JavaScript programming language.

1. Install MongoDB, the database we'll use with our Node.js and Express stack.

**## Node.js**

\_\_Note: when copying the code snippets, please exclude the `$` as you paste and run the code into your terminal. The dollar sign `$` is simply an indicator of the logged-in user's terminal prompt in the examples.\_\_

1. Install Node.js with Homebrew by running the following command in the Terminal:

```bash

$ brew install node

```

2. Run the Terminal command `node -v` to check that Node.js was installed. It should print a version number greater than or equal to "v5.8.0". The Terminal command `node` changes your Terminal into a Javascript REPL ("Read Evaluate Print Loop"), like the right-hand side of repl.it. Use the shortcut `ctrl + c`, or type `.exit` to quit out of the REPL and return to the normal Terminal prompt.

2. Run the Terminal command `which npm` to check that NPM is installed. The Node Package Manager, used through various `npm` commands, is a lot like Homebrew, except we'll use it for Node.js-specific tools instead of for general Mac tools. NPM packages are often called "node modules."

**### Nodemon**

Nodemon (short for "node monitor") will make our Node.js workflow more efficient.

1. Install nodemon globally with the following Terminal command:

```bash

$ npm install -g nodemon

```

**## MongoDB**

MonogDB is a database that stores information as easy to read "documents". We'll use it to store data in our Node.js and Express stack.

1. Use Homebrew to update all our brew packages.

```bash

$ brew update

```

2. Run `brew install` for \*\*MongoDB\*\*.

```bash

$ brew install mongodb

```

3. Then we'll need a directory for \*\*MongoDB\*\* to save data. Make sure you create this directory inside your root (`/`) directory.

```bash

$ cd /

$ sudo mkdir -p data/db

```

You will be prompted to enter your system password here. This is the password you use to log into your computer.

\*Sidenote:

When you use this `sudo` command, you are saying "superuser do" which can force your computer to perform the command you entered, even if it's a bad idea. This is why you're prompted to insert a password. Terminal makes sure your computer is secure and that you're sure about the command you're about to execute forcefully.\*

![relevant xkcd](https://cloud.githubusercontent.com/assets/6520345/17527880/f458616c-5e21-11e6-9156-4db012c5efc7.png)

[Source: xkcd](https://xkcd.com/149/)

4. Finally we'll want to make sure we have permission to read and write to this directory.

```bash

$ sudo chown -R $USER data/db

```

Again, you'll need to enter your system password.

5. Run two commands to check whether the install worked. You should see a file path after each command.

```bash

$ which mongod

$ which mongo

```

If this has worked correctly, you will see `/usr/local/bin/mongod` and `/usr/local/bin/mongo` respectively as the outputs from terminal.

**## RoboMongo**

RoboMongo is a GUI (Graphical User Interface) tool to let us see the data in our Mongo databases. Let's install that now.

1. Go to [https://robomongo.org/download](https://robomongo.org/download) and download the free (community) edition.

2. Install it!

**# Installfest Step 1.2: Ruby on Rails Stack**

1. Update your install of the Ruby programming language and install rvm to manage versions of Ruby.

2. Install Rails, a back-end web development framework for the Ruby programming language.

3. Install PostgreSQL, the database we'll use with our Ruby on Rails stack.

**## Do you already have ruby, rails or rvm? If yes, make sure they're up to date.**

If you've already experimented with Ruby and/or Rails before, verify that your versions are correct for this upcoming class.

1. In your Terminal, run `ruby --version`. The output will include the version number of Ruby you have installed. \*\*Verify you are running version 2.2+ of Ruby and that it does not say `universal.x86-64`\*\*. The `universal` build is the one that comes with OS X, we want to use a ruby version supplied by RVM.

2. In your Terminal, run `rails --version`. \*\*Verify you are running version 5 of Rails\*\*.

If you are using an earlier version of RVM, Ruby or Rails, continue with the instructions to get your environment set up. Run these version commands in the Terminal again after you're done to ensure everything is working properly.

**## RVM and Ruby**

\_\_Note: when copying the code snippets, please exclude the `$` as you paste and run the code into your terminal. The dollar sign `$` is simply an indicator of the logged-in user's terminal prompt in examples.\_\_

[RVM](https://rvm.io) is a Ruby Version Manager. It lets you easily switch between versions of the Ruby programming language for different projects.

1. Run the following command in your Terminal to install both RVM and the latest version of Ruby.

<!--

\*\*Note for August 2017 cohort ONLY:\*\* Due to a current bug in the RVM installer, you should NOT run the command below, and should instead run:

```

\curl -sSL https://raw.githubusercontent.com/wayneeseguin/rvm/stable/binscripts/rvm-installer | bash -s stable --ruby

```

-->

```bash

$ curl -L https://get.rvm.io | bash -s stable --ruby --auto-dotfiles

```

This installation could take a few minutes, so don't panic if it's slow.

2. Close your terminal and open a new one now.

3. Run

```bash

$ gem install bundler

```

**## Rails 5**

Ruby packages are referred to as "gems".

1. Use the following Terminal command to install Rails 5.

```bash

$ gem install rails

```

This might take a few minutes.

If this causes errors on your machine, ask for help.

**## PostgreSQL**

**### Postgres**

1. Install postgres via Homebrew

```bash

$ brew install postgres

```

2. Configure postgres to start when the system does.

```bash

$ brew services start postgresql

```

3. Check that your install worked

```bash

$ which psql

```

If it worked, you should see ```/usr/local/bin/psql``` as the terminal output.

**# Installfest Step 2: Mac Development Tools & Applications**

**## Text Editor**

Our class will be using [Atom](editor-atom.md) as our preferred text editor. <!--However, feel free to use [Sublime Text 3](editor-sublime-text-3.md) as it is very similar. -->

**## Browser**

We recommend you install the following Chrome Extensions:

\* [JSON View](https://chrome.google.com/webstore/detail/jsonview/chklaanhfefbnpoihckbnefhakgolnmc)

\* [Postman](http://www.getpostman.com/)

**## Window Manager**

While programming it's pretty common to need to juggle the placement of multiple windows. To speed up this process, we'll install a program that does this for us (and gives us a bunch of convenient keyboard shortcuts).

1. Download https://raw.github.com/onsi/ShiftIt/master/ShiftIt.zip

\* Next, extract it and drag the application icon into your `Applications` directory.

\* Finally, launch the app by right-clicking on the ShiftIt icon and click open, follow the instructions to enable the accessibility options.

2. We recommend you familiarze yourself with ShiftIt commands and keyboard shortcuts, like:

\* Move window to the left-half: Command+Option+Control+LeftArrow

\* Move window to the right-half: Command+Option+Control+RightArrow

\* \*\*M\*\*aximize a window: Command+Option+Control+\*\*M\*\*

**Part 1: Users & Auth**

**Setup**

Create a new rails app without tests and with a PostgreSQL database:

rails new lib-app -T -d postgresql

cd lib-app

Create the databases:

rails db:create

**Routeside-in Development**

Rails recommends "routeside-in" development - starting with a RESTful route and an idea of what that route should do, then working through everything else required to enable that route!

Because Rails errors are famously helpful, you can also think of this as "error driven development" to some extent. When you're setting up a simple Rails app, errors will guide you to writing everything you need.

**How to Fix Server “No Password Supplied” Issue**

- Open Sublime Text > Click Open > select the rails-auth-library-app folder > the folder will open up in the side view > click the config folder > click the database.yml file > edit the file so that it looks like this:

# PostgreSQL. Versions 9.1 and up are supported.

#

# Install the pg driver:

# gem install pg

# On OS X with Homebrew:

# gem install pg -- --with-pg-config=/usr/local/bin/pg\_config

# On OS X with MacPorts:

# gem install pg -- --with-pg-config=/opt/local/lib/postgresql84/bin/pg\_config

# On Windows:

# gem install pg

# Choose the win32 build.

# Install PostgreSQL and put its /bin directory on your path.

#

# Configure Using Gemfile

# gem 'pg'

#

default: &default

adapter: postgresql

encoding: unicode

# For details on connection pooling, see Rails configuration guide

# http://guides.rubyonrails.org/configuring.html#database-pooling

pool: <%= ENV.fetch("RAILS\_MAX\_THREADS") { 5 } %>

development:

adapter: postgresql

encoding: unicode

database: lib-app\_development

pool: 5

username: postgres

password: Bonjour19

host: localhost

# The specified database role being used to connect to postgres.

# To create additional roles in postgres see `$ createuser --help`.

# When left blank, postgres will use the default role. This is

# the same name as the operating system user that initialized the database.

#username: lib-app

# The password associated with the postgres role (username).

#password:

# Connect on a TCP socket. Omitted by default since the client uses a

# domain socket that doesn't need configuration. Windows does not have

# domain sockets, so uncomment these lines.

#host: localhost

# The TCP port the server listens on. Defaults to 5432.

# If your server runs on a different port number, change accordingly.

#port: 5432

# Schema search path. The server defaults to $user,public

#schema\_search\_path: myapp,sharedapp,public

# Minimum log levels, in increasing order:

# debug5, debug4, debug3, debug2, debug1,

# log, notice, warning, error, fatal, and panic

# Defaults to warning.

#min\_messages: notice

# Warning: The database defined as "test" will be erased and

# re-generated from your development database when you run "rake".

# Do not set this db to the same as development or production.

test:

adapter: postgresql

encoding: unicode

database: lib-app\_test

pool: 5

username: postgres

password: Bonjour19

host: localhost

port: 5432

# As with config/secrets.yml, you never want to store sensitive information,

# like your database password, in your source code. If your source code is

# ever seen by anyone, they now have access to your database.

#

# Instead, provide the password as a unix environment variable when you boot

# the app. Read http://guides.rubyonrails.org/configuring.html#configuring-a-database

# for a full rundown on how to provide these environment variables in a

# production deployment.

#

# On Heroku and other platform providers, you may have a full connection URL

# available as an environment variable. For example:

#

# DATABASE\_URL="postgres://myuser:mypass@localhost/somedatabase"

#

# You can use this database configuration with:

#

# production:

# url: <%= ENV['DATABASE\_URL'] %>

#

production:

<<: \*default

database: lib-app\_production

username: lib-app

password: <%= ENV['LIB-APP\_DATABASE\_PASSWORD'] %>

**The homepage ('/') should show how many users are signed up.**

Let's start with a route for root, which is just a helper method in Rails for /.

config/routes.rb

Rails.application.routes.draw do

root to: 'users#index'

end

We can look at how these routes are interpreted by Rails.

rails routes

Which should give us the following routes:

Prefix Verb URI Pattern Controller#Action

root GET / users#index

If we're working routeside-in, the question now is **what to do next?** There are a few ways to figure this out. We could go to our new home page in the browser and see a helpful error message. Just looking at the output we have so far, you may notice we don't have a users#index. We don't even have a UsersController.

Let's practice using our rails generate skills to create a users controller and associated files.

rails g controller users

This does something like the following:

\*\*\* create app/controllers/users\_controller.rb

invoke erb

\*\*\* create app/views/users

invoke helper

\*\* create app/helpers/users\_helper.rb

invoke assets

invoke coffee

\*\* create app/assets/javascripts/users.coffee

invoke scss

\*\* create app/assets/stylesheets/users.scss

Note the special create statements here. The \*\*\* ones are the most important. It creates the users\_controller.rbfile and the views/users directory.

Now that we have a users\_controller.rb we should add our users#index method.

class UsersController < ApplicationController

def index

end

end

Now, if you visit the site in your browser, you may see an error about a missing template! We need to actually create a users/index.html.erb view template for this route to render:

touch app/views/users/index.html.erb

Then we can go ahead and add some actual content to our index - the count of users:

<h1>Welcome to Users Index.</h1>

<div>

There are currently <%= @users.length %> user(s) signed up!

</div>

Check that root route in the browser again. Uh-oh! If you have an error, pop back over to the controller and make sure we have the right data available for the view:

class UsersController < ApplicationController

def index

@users = User.all

end

end

But wait! If you go to localhost:3000 after this step (like you should be!), we have a problem. No User model.

Generate a User model with email, first\_name, last\_name, and password\_digest strings.

rails g model user email:string first\_name:string last\_name:string password\_digest:string

Then go ahead and verify that the migration looks correct:

sample db/migrate/201575943834\_create\_users.rb:

db/migrate/201675943834\_create\_users.rb

class CreateUsers < ActiveRecord::Migration[5.0]

def change

create\_table :users do |t|

t.string :email

t.string :first\_name

t.string :last\_name

t.string :password\_digest

t.timestamps

end

end

end

We're ready to migrate!

rails db:migrate

Ok, now we should see 0 users signed up on the home page.

That makes sense because there's no way to sign up yet. **YET.**

Optionally, you can add a new GET /users route that also uses the users#index controller action.

**The ' GET /users/new' route should show a signup form.**

Create this route. Remember to set up the proper prefix.

Rails.application.routes.draw do

root to: 'users#index'

get '/users/new', to: 'users#new', as: 'new\_user'

end

Check that you get the following output from rails routes:

Prefix Verb URI Pattern Controller#Action

root GET / users#index

new\_user GET /users/new(.:format) users#new

We don't have a users#new controller action, so let's create one. If you'd like, you can go ahead and fill it in so it gets the data for the form view.

class UsersController < ApplicationController

#...

def new

# we make a new user

# to pass to the form view later

@user = User.new

end

Then we can continue on to creating a new.html.erb view for the sign up form:

touch app/views/users/new.html.erb

In that file, use form\_for @user to create a sign up form.

Sign Up

<%= form\_for @user do |f| %>

<div>

<%= f.text\_field :first\_name, placeholder: "First Name" %>

</div>

<div>

<%= f.text\_field :last\_name, placeholder: "Last Name" %>

</div>

<div>

<%= f.text\_field :email, placeholder: "Email" %>

</div>

<div>

<%= f.password\_field :password, placeholder: "Password" %>

</div>

<%= f.submit "Sign Up" %>

<% end %>

When we try to visit /users/new we'll get an error because rails, in its infinite wisdom, has discovered a problem! Where will we post the form?! There's no route for that. So let's go ahead and appease rails by adding our form submission route:

post '/users', to: 'users#create'

Visit /users/new in your browser. You should see a form. Inspect it to see that the form\_for helper renders a form like the following (note the authenticity token):

Sign Up

<form class="new\_user" id="new\_user" action="/users" accept-charset="UTF-8" method="post"><input name="utf8" type="hidden" value="✓">

<input type="hidden" name="authenticity\_token" value="6EhU37Nz3gcbL8UsJL6Q868ZOI0a/UWJ2GLMOdu4fuiMEb8y8AthB6t032Of4seUgI0NMv8wOPPNPZ/eFOkNfw==">

<div>

<input placeholder="First Name" type="text" name="user[first\_name]" id="user\_first\_name">

</div>

<div>

<input placeholder="Last Name" type="text" name="user[last\_name]" id="user\_last\_name">

</div>

<div>

<input placeholder="Email" type="text" name="user[email]" id="user\_email">

</div>

<div>

<input placeholder="Password" type="password" name="user[password]" id="user\_password">

</div>

<input type="submit" name="commit" value="Sign Up" data-disable-with="Sign Up">

</form>

Note here the correlation between the symbol we put into f.text\_field and name="..." in the generated form.

Note what kind of request this form will make and the path it's going to.

<form class="new\_user" id="new\_user" action="/users" accept-charset="UTF-8" method="post">

What error do you see if you try to submit this form in the browser now?

It looks like this form is sending POST /USERS. Do we have a route for that?

**Having doubts? "Rails" your routes!**

**A POST to /users should create a new user in the database.**

Make sure your routes looks like this:

Rails.application.routes.draw do

root to: 'users#index'

get '/users/new', to: 'users#new', as: 'new\_user'

post '/users', to: 'users#create'

end

Running rails routes should now show:

Prefix Verb URI Pattern Controller#Action

root GET / users#index

new\_user GET /users/new(.:format) users#new

users POST /users(.:format) users#create

Then, add the create action in the users controller. Remember to use strong parameters. (Bonus for separating these into a private method.)

class UsersController < ApplicationController

# ...

def create

@user = User.create(user\_params)

redirect\_to root\_path

end

private

def user\_params

params.require(:user).permit(:first\_name, :last\_name, :email, :password)

end

end

Now when you submit the form, you probably get the following error:

ActiveRecord::Unknown

AttributeError in UsersController#create

unknown attribute 'password' for User.

Oops! Don't we need users to submit passwords to sign up??

No! We only have a password\_digest in our user model, and we'll only store a password\_digest in our database. This password\_digest will be a salted and hashed (obfuscated) version of the user's acutal password.

We actually haven't set up any authentication logic yet -- part of this logic will be turning the plain password the user enters into a password digest that is safe to store in our database.

Uncomment your bcrypt in your Gemfile:

Gemfile

...

# Use ActiveModel has\_secure\_password

gem 'bcrypt'

...

Once bcrypt is added, we can use the has\_secure\_password method in our user model:

class User < ApplicationRecord

has\_secure\_password

end

Reload the page to make sure this hasn't caused any errors.

We changed the Gemfile by bringing in bcrypt - did you remember to bundle install? You may also need to restart your server.

Now when we post the form for the user, you'll see the user being created. The difference now is that the password is being properly hashed and salted into a password\_digest, so it's safe to store. Thanks for has\_secure\_password, Rails!

**The GET /users/:id route should show a view with all of the information about the user with id :id.**

Now we'll add a route to GET /users/:id.

Rails.application.routes.draw do

root to: 'users#index'

get '/users/new', to: 'users#new', as: 'new\_user'

post '/users', to: 'users#create'

get '/users/:id', to: 'users#show', as: 'user'

end

Guess what fun error you'll get later if you put the /users/:id route before the /users/new route!

Rails routes!

Prefix Verb URI Pattern Controller#Action

root GET / users#index

new\_user GET /users/new(.:format) users#new

users POST /users(.:format) users#create

user GET /users/:id(.:format) users#show

Also add a users#show controller action.

class UsersController < ApplicationController

#...

def show

@user = User.find\_by\_id(params[:id])

end

end

Try visiting /users/1 in the browser. We need a users/show.html.erb to display the user's information.

touch app/views/users/show.html.erb

<div>

Welcome, <%= @user.first\_name %>!

</div>

Let's test what we've got so far by going to the show page of an existing user in the browser.

**Managing Sessions for Existing Users**

Now that we can create a user, we need to let existing users log in and out.

**The GET /login route should show a form to log in.**

Logging in and logging out out is a concern of a new controller, the sessions controller.

Rails.application.routes.draw do

#...

get '/login', to: 'sessions#new'

end

**Self-check for understanding**: Justify the choice to use sessions#new for the log in form.

**Self-check for understanding**: Which sessions controller action will we use to actually set the session and log the user in?

**Self-check for understanding**: Which sessions controller action will we use to log out?

Let's generate the new controller.

rails g controller sessions --no-assets

This will create sessions\_controller.rb, sessions\_helper.rb and a views/sessions/ directory, but it will skip adding app/assets/javascripts/sessions.coffee and app/assets/stylesheets/sessions.scss.

Now we need to add the sessions#new action. This will just show a log in form that takes a user's password and email.

class SessionsController < ApplicationController

def new

@user = User.new

end

end

Then we need to add a view for the sessions/new.html.erb:

touch app/views/sessions/new.html.erb

This login form can look very similar to the form for sign in, but we'll need to be a little more specific about the form\_forline, but it only needs email and password fields:

Login

<%= form\_for @user, url: "/sessions", method: "post" do |f| %>

<div>

<%= f.text\_field :email, placeholder: "Email" %>

</div>

<div>

<%= f.password\_field :password, placeholder: "Password" %>

</div>

<%= f.submit "Log In" %>

<% end %>

Try using an email\_field for some built-in client side validataions and a password\_field to obscure the password as the user types.

Verify that you can see the log in form. What error do you get when you submit?

**POSTing to /sessions should log a user in if the email/password combination was correct.**

Note that the form is getting submited to POST /sessions. We don't have a sessions#create however or a route to handle the post.

Rails.application.routes.draw do

# ...

get '/login', to: 'sessions#new'

post '/sessions', to: 'sessions#create'

end

Verify your routes:

root GET / users#index

new\_user GET /users/new(.:format) users#new

users POST /users(.:format) users#create

user GET /users/:id(.:format) users#show

login GET /login(.:format) sessions#new

sessions POST /sessions(.:format) sessions#create

Note that the POST /sessions route has automatically been assigned the prefix sessions. This may seem odd, but its url or path (sessions\_path, "/sessions") is actually the same as the sessions#index route where we'd usually expect to see this prefix. That said, we won't use the prefix in our code.

Now let's add the sessions#create controller action. It should log the user in by saving their id into the session.

This will involve:

• creating a User.confirm method in the user model

• creating login and current\_user helper methods in the sessions helpers file

• including the session helper methods in the application controller

class SessionsController < ApplicationController

def create

user\_params = params.require(:user).permit(:email, :password)

# confirm that email/password combination is correct

@user = User.confirm(user\_params)

if @user

login(@user)

redirect\_to @user

else

redirect\_to login\_path

end

end

end

HOLD THE HORSES! What is User.confirm? The comment claims to tell us what it's doing, but where did it come from? Is this a built-in model method? Does it come with has\_secure\_password??

Nope, it's something we suggest you add to your User model as a custom model method. This will make your code more modular.

Before we go forward let's go ahead and drop in a very key piece of confirmation logic into our User model. Create a confirm class method that checks whether the email and password from the parameters are a matching pair. Use authenticate from has\_secure\_password.

class User < ApplicationRecord

has\_secure\_password

def self.confirm(params)

@user = User.find\_by({email: params[:email]})

@user ? @user.authenticate(params[:password]) : false

end

end

HOLD THOSE HORSES! What's up with ? :? Don't worry, it's just your friendly neighborhood ternary operator. It's keeping us from trying to call @user.authenticate when the @user is nil.

BUT WOOAH, HORSES! What is @user.authenticate?!? Where does that come from? What is it doing? Hint: has\_secure\_password.

You can test this is working by trying it out in your rails console:

rails c

> reload! # use this if you already had the console open

> User.confirm({email: "test@test.com", password: "123"})

> User.confirm({email: "test@test.com", password: "WRONG"})

Try with an existing correct email/password combination and with an incorrect password/email combination.

Okay, back to the sessions#create action. If we confirm this is an authentic user, we should log them in. Write code that logs in confirmed users and sends them to their show page, and write code that redirects non-confirmed visitors to the login path. (You may want to create and use a login session helper method).

class SessionsController < ApplicationController

def create

user\_params = params.require(:user).permit(:email, :password)

# confirm that email/password combination is correct

@user = User.confirm(user\_params)

if @user

login(@user)

redirect\_to @user

else

redirect\_to login\_path

end

end

end

HORSE, STOP! What the heck is this login method? It's a method we suggest you add to the helper methods for this controller.

Find the sessions helper file and add:

module SessionsHelper

def login(user)

session[:user\_id] = user.id

@current\_user = user

end

def current\_user

@current\_user ||= User.find\_by\_id(session[:user\_id])

end

end

Note that we've also snuck in a @current\_user instance variable and current\_user method. This is so we can look up the logged in user from the session later, like in views.

Before we can use the methods, though, we have to add these methods to the ApplicationController.

class ApplicationController < ActionController::Base

protect\_from\_forgery with: :exception

include SessionsHelper

end

Now try to log in using the form, both with a correct email/password combination and an incorrect one. Do you see a welcome when you use the correct information? And do you see the log in form again if you enter the wrong one (hm, a flash message would be nice here)? If so, you're ready to continue. Otherwise, you should start debug before moving on.

**Refactor: Signing up should also log a user in.**

After a user is signed up they should be logged in. Thank goodness we have a convenient login helper to keep our code DRY! Refactor the controller action that handles signing up so that it calls this login method for the new user and redirects to the user show page.

class UsersController < ApplicationController

def create

@user = User.create(user\_params)

login(@user) # <-- log the user in

redirect\_to @user # <-- go to show

end

end

Try signing up - you should see the welcome message from the user show page.

**The GET /logout route should log the user out.**

Start with the route! We'll use the sessions#destroy controller action to handle logging out.

Rails.application.routes.draw do

# ...

get '/login', to: 'sessions#new'

get '/logout', to: 'sessions#destroy'

post '/sessions', to: 'sessions#create'

end

Run rails routes!

Prefix Verb URI Pattern Controller#Action

root GET / users#index

new\_user GET /users/new(.:format) users#new

users POST /users(.:format) users#create

user GET /users/:id(.:format) users#show

login GET /login(.:format) sessions#new

sessions POST /sessions(.:format) sessions#create

We decided to use sessions#destroy because all of the information that's keeping a user logged in is in the user's session.

Strictly speaking, it isn't RESTful to do a destroy with the GET method (it should be DELETE). However, it's super convenient to do log out this way so we can add a log out link on each page.

The sessions#destroy controller action needs to clear the user\_id from the session. This is a great time to start thinking about using a logout session helper method, too.

class SessionsController < ApplicationController

...

def destroy

logout # coming soon in SessionsHelper

redirect\_to root\_path

end

end

Let's go ahead and add that logout helper method to correspond to the login we wrote before.

module SessionsHelper

def login(user)

session[:user\_id] = user.id

@current\_user = user

end

def current\_user

@current\_user ||= User.find\_by\_id(session[:user\_id])

end

def logout

@current\_user = session[:user\_id] = nil

end

end

Now we can go directly to the /logout URL to log out (delete the session user\_id), but we should also have a "Log Out" button somewhere.

Even better would be a navbar with all the login/signup/logout options. Let's add a very simple list of navigation links to views/layouts/application.html.erb with some conditional logic, depending on whether we have a current user logged in:

<body>

<ul>

<% if current\_user %>

<li><%= link\_to "Profile", user\_path(current\_user) %></li>

<li><%= link\_to "Log Out", logout\_path %></li>

<% else %>

<li><%= link\_to "Create Account", new\_user\_path %></li>

<li><%= link\_to "Log In", login\_path %></li>

<% end %>

</ul>

<%= yield %>

<!-- ... -->

</body>

Go ahead and check all your links are working (try it both logged in and logged out).

As a final touch, let's add "flash" messages to inform the user that they are "Successfully logged in" and "Successfully logged out". Start by setting up the flash messages in the controller.

class SessionsController < ApplicationController

# ...

def create

user\_params = params.require(:user).permit(:email, :password)

@user = User.confirm(user\_params)

if @user

login(@user)

flash[:notice] = "Successfully logged in." # <--- Add this flash notice

redirect\_to @user

else

flash[:error] = "Incorrect email or password." # <--- Add this flash error

redirect\_to login\_path

end

end

def destroy

session[:user\_id] = nil

flash[:notice] = "Successfully logged out." # <--- Add this flash notice

redirect\_to root\_path

end

end

Update views/layouts/application.html.erb to display the messages.

<body>

<!-- ... -->

<% flash.each do |name, msg| %>

<p>

<small> <%= name.capitalize %>: <%= msg %> </small>

</p>

<% end %>

<!-- ... -->

<%= yield %>

</body>

Try logging in with correct and incorrect information, and try logging out. Ensure that these are working.

Nice work! We're finished with Authentication!

**Part 2: Library Users**

**A Library Model**

Let's add our second model, a Library!

rails g model library name:string floor\_count:integer floor\_area:integer

We want a user to be able to join multiple libraries, but each library can also have multiple members. This means a many-to-many or n:n relationship.

Thus, we need a library\_user model for our join table. It should have foreign keys for both other models.

rails g model library\_user user:belongs\_to library:belongs\_to

In the future we can store other things on the library\_user model that are relevant to someone's membership in a library like join date, membership "level", etc.

We will also need two different controllers for library and library\_user. Let's start by implementing CRUD with libraries in the library controller.

rails g controller libraries

**A Library Index**

Add a route to be able to view all the libraries.

Rails.application.routes.draw do

...

get '/libraries', to: 'libraries#index'

end

Add a libraries#index method to the libraries controller.

class LibrariesController < ApplicationController

def index

@libraries = Library.all

end

end

Add a basic view for all libraries.

<% @libraries.each do |library| %>

<div>

<h3><%= library.name %></h3>

</div>

<br>

<% end %>

**A New Library**

To be able to add a new library, we need a GET /libraries/new route to display the form.

Rails.application.routes.draw do

...

get '/libraries/new', to: 'libraries#new', as: 'new\_library'

end

Add a libraries#new controller action.

class LibrariesController < ApplicationController

...

def new

@library = Library.new

end

end

Add a view for the new library form.

<%= form\_for @library do |f| %>

<div>

<%= f.text\_field :name, placeholder: "Name" %>

</div>

<div>

<%= f.number\_field :floor\_count, placeholder: "Floor Count" %>

</div>

<div>

<%= f.number\_field :floor\_area, placeholder: "Floor Area" %>

</div>

<%= f.submit %>

<% end %>

This form has nowhere to go; if we try to submit it we get an error because there is no POST /libraries route. Add one.

Rails.application.routes.draw do

...

post '/libraries', to: 'libraries#create'

end

Then we need a corresponding libraries#create.

class LibrariesController < ApplicationController

def create

@library = Library.create(library\_params)

redirect\_to libraries\_path # very light on the error handling, for now!

end

private

def library\_params

params.require(:library).permit(:name, :floor\_count, :floor\_area)

end

end

**CRUDing Libraries**

We now have the ability to view all libraries and create new libraries.

**Independent Practice**: Implement libraries#show on your own. You will need to create routes, controller actions, and views.

**Associating Users and Libraries**

Before we get start letting users become library members, we need to wire together all of our models to know about these associations. Use the has\_many through pattern to set up the many-to-many association in the models.

class LibraryUser < ApplicationRecord

belongs\_to :user

belongs\_to :library

end

The above will already be in the LibraryUser model! How could that be? Rails guesses that a library-user will be the join table for users and libraries and builds the model accordingly.

class User < ApplicationRecord

has\_many :library\_users, dependent: :destroy

has\_many :libraries, through: :library\_users

# ...

end

class Library < ApplicationRecord

has\_many :library\_users, dependent: :destroy

has\_many :users, through: :library\_users

end

You should now test this out in the console.

> user = User.first

> user.libraries

#=> []

> sfpl = Library.create({name: "SFPL"}) # San Francisco Public Library

> sfpl.users

#=> []

> sfpl.users.push(user)

> sfpl.users

#=> [ <#User ... @id=1> ]

> LibraryUser.count

#=> 1

> reload!

> user.libraries

#=> [ <#Library ... @name="SFPL" @id=1> ]

**library\_users Controller**

In order for us to have users become members libraries, we need to first create a library\_users controller. Generate that now.

rails g controller library\_users

We want to be able to view all user memberships to a library. We need to decide on a route for this. Based on RESTful routing, we could choose /users/:user\_id/libraries or /libraries/:library\_id/users. Either one would be okay, but an application should not have both. We'll choose the first since this app is more centered on users than libraries.

Rails.application.routes.draw do

...

get '/users/:user\_id/libraries', to: 'library\_users#index', as: 'user\_libraries'

end

We also need the corresponding index method in the library\_users controller.

class LibraryUsersController < ApplicationController

def index

@user = User.find(params[:user\_id])

@libraries = @user.libraries # so we type less in the view

end

end

Then we can have the index view list the user's libraries (app/views/library\_users/index.html.erb):

<div><%= @user.first\_name %> is a member of the following libraries</div>

<ul>

<% @libraries.each do |lib| %>

<li><%= lib.name %></li>

<% end %>

</ul>

We can test this by going to localhost:3000/users/1/libraries. If you want, you can test that this is working by launching your rails console and adding a library to a user, then refreshing the page.

**Add A Membership**

We should make a button that allows a user to become a member of a library!

Let's go back to the libraries#index view and add a button to do just that.

<% @libraries.each do |library| %>

<div>

<h3><%= library.name %></h3>

<% if current\_user %>

<%= button\_to "Join", library\_users\_path(library) %>

<% end %>

</div>

<br>

<% end %>

We don't have an endpoint yet that allows a user to join a library, so let's add that now so that our form will work.

Rails.application.routes.draw do

...

post '/libraries/:library\_id/users', to: 'library\_users#create', as: 'library\_users'

end

Note: We've chosen to structure this route so we can easily get the library id from the url, but there are many ways this route could be written. We just have to make sure we can access the correct library and the correct user in the controller so we connect the right entities.

Then, we need to add a create action in LibraryUsersController that adds the user to the library.

class LibraryUsersController < ApplicationController

...

def create

@library = Library.find(params[:library\_id])

@library.users.push(current\_user) # no error handling currently

redirect\_to current\_user

end

end

**Authorization**

Let's say that in order to visit a users#show page, you have to be logged in. Use a special before\_action to check for this. Set up a require\_login session helper to make help keep the controller "skinny."

class UsersController < ApplicationController

before\_action :require\_login, only: [:show]

...

def show

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

render :show

end

end

This before\_action line means there must be a require\_login method somewhere that will be called before the show action is run. Add a require\_login helper method to the sessions helper to ensure there is a current user.