*Let's Build*

# INSTAGRAM FROM SCRATCH

with Ruby on Rails

BEN WALKER | devwalks.com

# Introduction

Hey there!

So you're done playing in the treehouse or you've completed your codeschool. You've been high-fiving Odin or your coding academy has kicked you out. What now?

Now it's time that you build lots of awesome things and get great at what matters (and therefore what employers want). The Next Step Ruby on Rails guides are all about that awkward *next step* after you've completed a few tutorials. It's at this point, you're asking:

- What now?
- What should I build next?
- What other skills do I need to develop?
- What do I need to learn to get a job?

In the Next Step Ruby on Rails tutorials & books, I do my very best to introduce you to more advanced concepts around Ruby on Rails and also drive home the core learnings that create the foundation for any Ruby on Rails developer.

In this free guide, you'll learn the following skills that are often left out of the majority of other tutorials:

- Customising Devise for your requirements.
- AJAX & streamlining user experience.
- Image previews before uploading.
- JQuery & Rails. How the two combine surprisingly well.
- Custom notifications feature for any web application.
- Adding 'likes' to a post or other component with Rails.
- Creating a follower / following relationship between your users, as seen on social networks such as Twitter or Instagram.
- Profile pages for individual users.

I sincerely hope this book helps you just as much as is has the readers of my blog.

If you'd like updates on my future projects, Next Step Ruby on Rails article series or actionable educational posts, sign up here and I'll send you a quick email as soon as I release a new post / video / lesson. I'm intending on re-writing this entire book using BDD, git workflows and going into much greater detail within

each of the particular Rails features we're dealing with. If that sounds like something that would float your boat, sign up above and I'll keep you posted (full disclosure: that one will be a paid product).

Also, my primary way of feeding my children is with Rails or Meteor consulting, so if you have a project you'd like to work on together, feel free to send me an email at ben@devwalks.com.

All the very best,

Ben Walker - devwalks.com

**ps.** This book WILL have a paid version that will be completely re-written for complete test-driven development, the number ONE requirement if you want a web development job / career. The premium version will also:

- Go into much deeper descriptions of all of the Rails features and quirks.
- Have answers for all of the homework assigned in this book and the blog series.
- Have complete code samples for each step of the journey.
- Be updated for Rails 5.0 upon release.

You can pre-order the book right here and I'll have a glorious guide in your pretty hands before you know it.

# How this book works

In the Next Step Ruby on Rails series, I constantly encourage you to do your own thing.  The absolute BEST way for you to learn any new skill is to struggle.

I'm sorry.

Yes, it sucks, no, it's not fun, but the unfortunate reality is that the struggle really does lead to results.

So please, embrace the struggle.

Every so often, you're going to see this goat:


And for that, I apologise (it seemed like a good idea at the time).  The point of this goat should be clear.  You do your **absolute** best to complete the task I've set you.

Of course, you could go ahead and read my instructions immediately but you'd only be cheating yourself.

These guides are written for those who've already completed a few tutorials, meaning there'll be much more value in building the application yourself wherever possible.  Google for answers before you reveal them here, read the docs, explore and discover!  Your learning experience will be that much better for your struggle.

**There's one other tip I'll give you...**

Rename everything and make it your own.

Where I call my application Photogram, call yours something else (it really doesn't matter what you choose).  Where I name each individual posted image a

'Post' (booorrriinnngg), you should call it something else.

You want this app to BELONG to you, don't just follow blindly.

Even if you think the name I use is the absolute best possible name in the universe, go ahead and change it. This is generally bad practice in progamming (naming things well is important) but for the sake of your education, we're going to pooh pooh that rule.

This guide also assumes that you have Rails installed on your computer of choice.

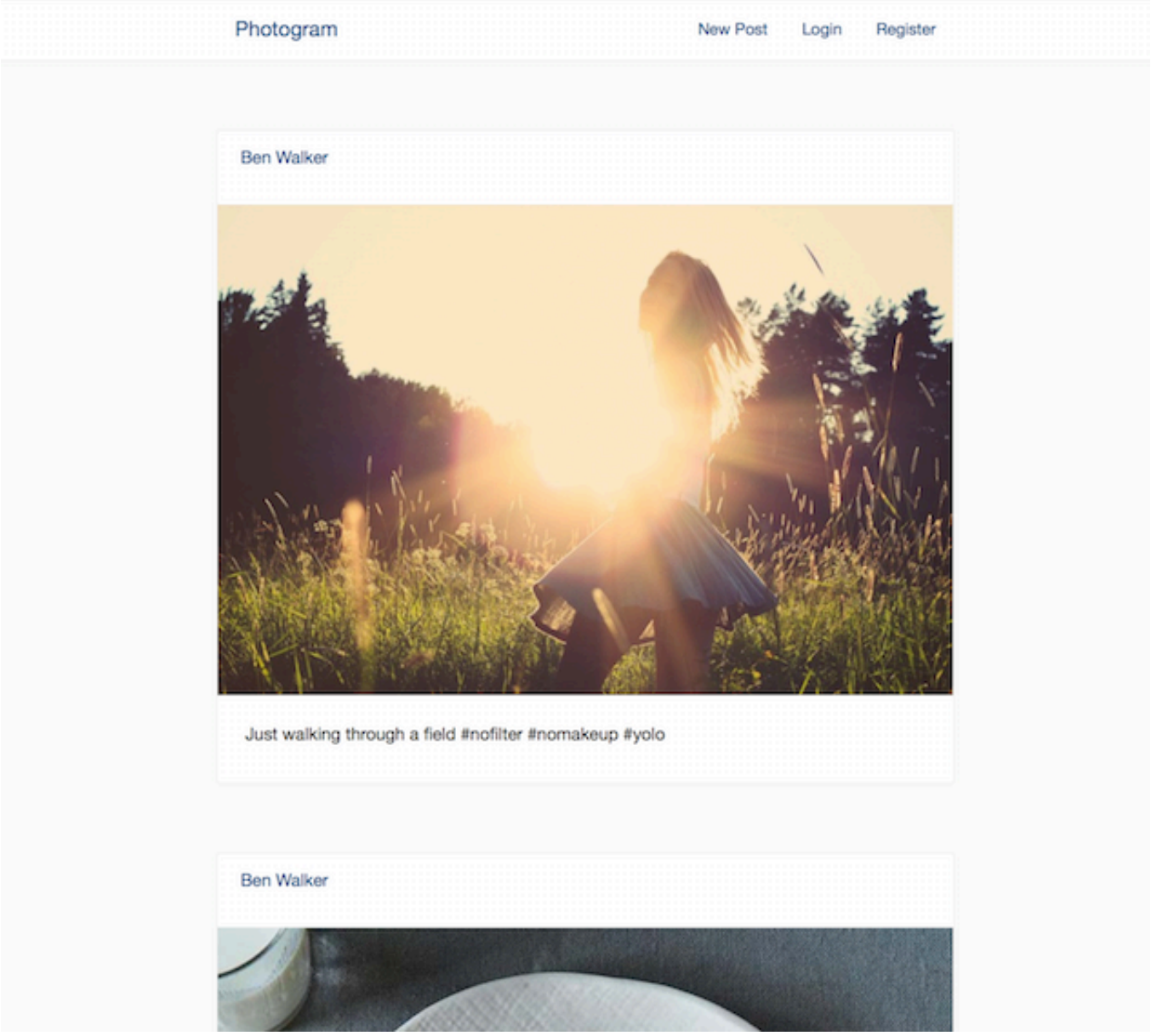
Let's get building!

# Chapter One - Just CRUD Things

In every beginner Ruby on Rails tutorial, you do some very similar activities. Generally you create some form of object that you can:

- Create
- Read (show)
- Update
- Delete

And in this guide we'll be doing the same to begin. Unfortunately, this is the reality of how a large part of web applications work. Keep your spirits high though, by the end of this chapter, our application is looking pretty great!



Let's build it!

**You are a Creator**

Let's create our project. I'm going to call mine Photogram but feel free to be creative and call it something else. I want this project to belong to *you*.

Jump in your terminal and cd to where you want to create your project.

Now use the rails command for creating a new project and we can begin with style.


---

Can't remember how to create a project in the terminal?

```
rails new Photogram
```

---

Now cd to your new directory in the termianl and let's run the server to see if we've got rails working properly.


---

Can't remember how to run a rails server?

```
bin/rails s
```

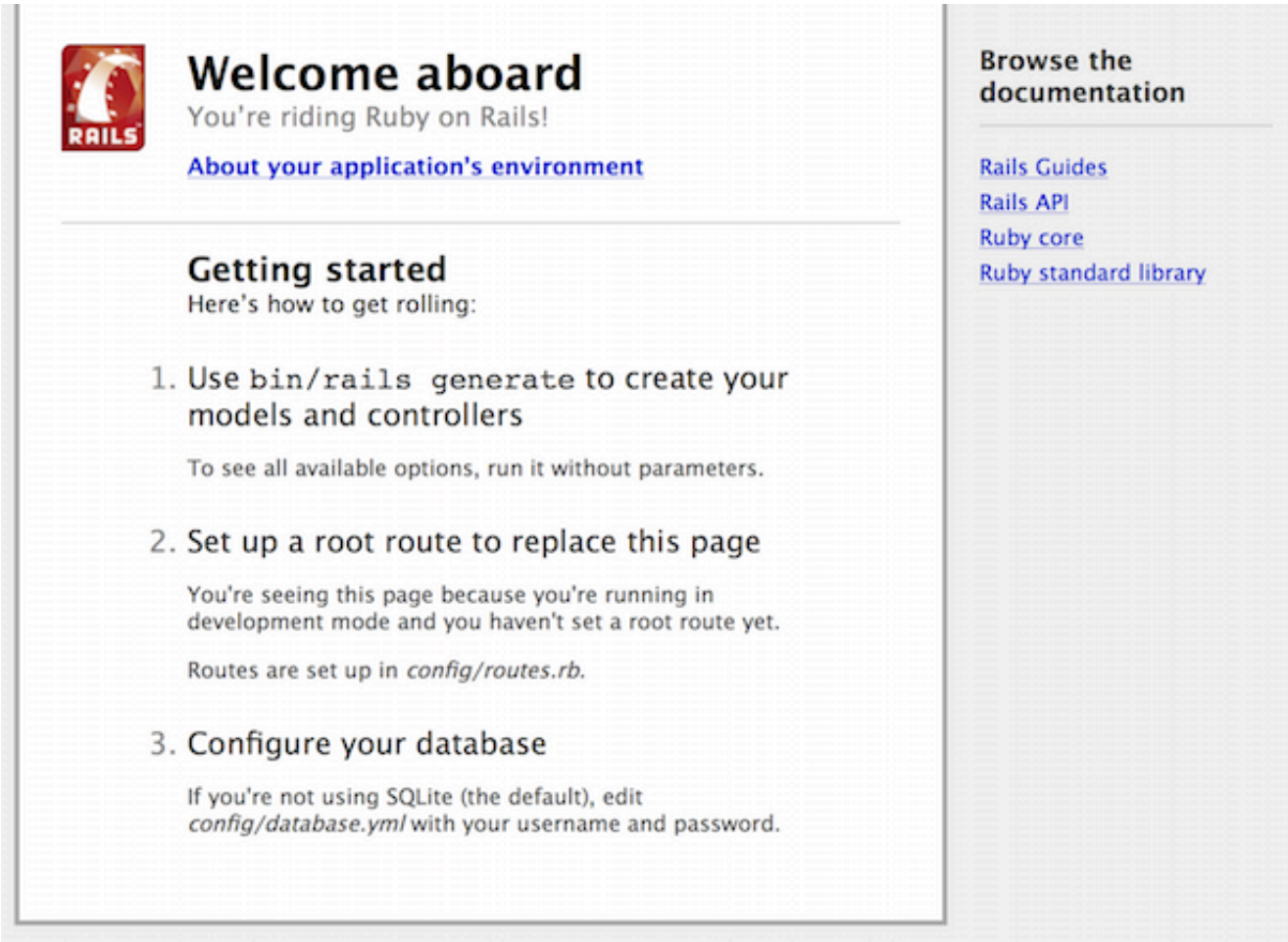
OR

```
bin/rails server
```

Use the first option if you want to be a super-efficient bad ass.

---

You should see this screen as a friendly reminder that you're awesome:




You are by the way ;)

**Back to the build, you fiend!**

Let's start building our application. Remember, the TDD version of this free book is coming and I don't want you to risk being too incredible just now. That and I want you to have a great foundation from which to learn to test.

Let's be cowboy devs for the moment.

This part of the guide will deal with handling the base of Photogram, the creation, editing and deletion of individual posts and their images. We're not going to use scaffolds here because we want to build competency and understand how the bits work.

If you were test driving this application, the first thing it'd complain about is the simple fact that nothing of value exists at our root route (localhost:3000/ for the moment).

To remedy the situation, we're going to need to create a controller that will handle the index, create, read, update and delete actions for our posts and then direct our routes.rb file to the index action of that controller.

Let's generate a controller and call it posts. Do that now in your terminal.


---

**Need a hand, friend?**

```
bin/rails g controller posts
```

---

Now, let's navigate over to our newly created controller `posts_controller.rb` and add a controller action. First, let's create our index action with no contents.


---

**Need another hand?**

```
def index
end
```

---

Our routes for our posts controller are going to be standard Rails RESTFUL routes. Rails let's us simplify the routing in this scenario by using "resources". Create a resources route for the posts controller in your routes.rb file now.


---

**Resource Shmesource**

That is terrible English. Just awful. Insert the below into you routes.rb file.

```
resources :posts
```

---

Rails is going to want you to have a view associated with our brand new index action. Do that now by creating a new view called index under the posts folder at app/views/posts.

I'm a fan of using HAML instead of the standard erb format but this is entirely optional ( I honestly do believe it's worth your time learning though). If using HAML, call your file `index.html.haml`, otherwise call it `index.html.erb`. Erb will be used in the paid version of this book for what it's worth.

If you do decide to go the HAML route, you'll need to add the haml gem to your gemfile and run the bundle command in your terminal (and restart your server). Read more about haml here.

As for our application, let's keep our new view nice and static for the moment and just add a title. Create a h1 tag in your html and call it whatever you called your application. In my case, it's Photogram.

Haml version:

```
%h1 Photogram
```

Now we have a functional index view, let's set the index action in the PostsController to be our root route within routes.rb.


---

Add the following line to your routes.rb file.

```
root 'posts#index'
```

---

Load up your server again, navigate to the root route and gaze upon the beauty that is your application. This is our MVP (just kidding). *Isn't she beautiful?*

**No it's not, it's awful.**

We'll get back to actually having something of value in our index but for the moment, let's actually create some posts for our index (and therefore root route) to display.

**First, a database!**

We're going to need to generate a model in terminal to store our posts, including caption and image.  Let's create the model "post" with only a string column for "caption".  We'll add the image functionality in a moment.


---

**Show me how, I dare not create a model all by myself!**

```
bin/rails g model Post caption:string
```

---

Now that we've created the migration files for our database we need to migrate those changes.  Run the Rails db migrate command in your terminal.


---

**I forget stuff sometimes!**

```
bin/rake db:migrate
```

---

Let's now add the image uploading functionality through a super fabulous gem called paperclip.

Click here to read about how to implement the gem in your project.  I want you to set up your project with paperclip as per their instructions, their documentation is about as good as it gets.

If you don't trust yourself, read the hidden section below but I implore you to do it yourself and continue without reading it.  If you run into trouble in the next step it'll be a good exercise for you in troubleshooting!


---

**Please help with the Paperclips!**

First things first, as per the docs, you'll need imagemagick running on your development computer.  Set it up as per the instructions here.

Next we'll add the paperclip gem to our gemfile as per their docs.

```
gem 'paperclip', '~> 4.2' #at the writing of this post
```

Read the next bit of the docs, what does it ask you to do now?

**I can't read well.**

Alright, let's add `has_attached _file` to our Post model.

```
class Post < ActiveRecord::Base
  has_attached_file :image
end
```

As per the docs (and a fantastic comment in the comments section) you'll also need to run the rails migration generator in your terminal with:

```
bin/rails g paperclip post image
```

---

How good are these docs!

Alright, we've now got paperclip ready to use on our Post model and we've got the ability to add captions to our selfies and cat pics. Now, let's make sure we're on the same page.

If you've setup Paperclip yourself, just know that I've used the following code in my application (spoiler alert).

**Model**

```
class Post < ActiveRecord::Base
  validates :image, presence: true

  has_attached_file :image, styles: { :medium => "640x" }
  validates_attachment_content_type :image, :content_type =>
/\Aimage\/.*\Z/
end
```

The styles in the Post model should make sense. We're simply re-sizing any image that we accept to the Instagram normal: 640px wide (This may not actually be the case, it seems a little small to be honest). We'll let people post tall images though.

Also note one other thing that I've done differently compared to the Paperclip docs. I've set a validation that I need an image to be submitted in that form. This is an image sharing social site after all. This then negates the need for a default "image missing" image.

As part of this process you should've generated a migration file and migrated it using `bin/rake db:migrate`. If your server is still running, you may have to

restart it.

We can now officially handle image uploads! Give yourself a high ten and forge onwards.

### Out with the old in with the new action

Create a new empty action in your `posts_controller.rb` file called 'new', exactly the same as you did earlier with the index action.

Once again, we'll need a view for this action, so let's create a new file under `views/posts/new.html.haml` (or .erb you sicko). As part of our new view, we're going to need to create a form that will accept the users caption and image.

A popular Rails gem for form building is simple_form. I'll personally be using this on my new view to create the form. Once again, head over to the simple_form repo here and read the docs.

Now construct the form for our purposes.

Not sure how to add the image upload functionality? I'll give you a hint, go back and check the Paperclip docs again.

If you're still struggling, that's ok, just do what you can and then read the answer below. Create your new view under app/views/posts and call it `new.html.erb` or `new.html.haml` and build the form there.


---

### Oh god, Please tell me, how do I create the form with simple_form?

First you'll need to add the simple_form gem to your gemfile as below:

```
gem 'simple_form', '~> 3.1.0'
```

Next run bundle in your terminal to incorporate it into your project. You may need to restart your server or it'll potentially throw an error when you try to navigate to your form.

---

Now you should be able to start creating your form in your new view! Follow along with the simple_form docs.


---

Here's a haml example. You should be able to translate this back to erb if required.

```
= simple_form_for @post do |f|
  = f.input :image
  = f.input :caption
  = f.button :submit
```

Simple_form needs to know the purpose of this form via an instance variable from the appropriate action in our controller. In this case, we want to create a new post. Add the `@post` variable to the new action.

```
def new
  @post = Post.new
end
```

---

Now we have a form that we can view on our local server. With your server running, navigate to `localhost:3000/posts/new` in order to look at your creation.

Even though it's ugly, I assure you it's very functional!

Let's try submitting our first post with any old offensive image you can find on

your computer.

We seem to be getting this:

**The action 'create' could not be found for PostsController**

Ok, that's pretty clear as to what we need to do. Let's create an empty 'create' action in our posts controller. This is exactly the same as our other two actions so there'll be no hints this time.

Let's try submitting again.

Hmmm, it's asking for a template but we don't really want to create a template for the create action, we want it to be a "behind the scenes" type action, one that saves the data in our form to the database and then sends us on our merry way.

Let's write some code that will save the data from the form to our database. You've probably done this before but it's easy to forget.


---

**Show me how!**

We want to use the create method on the Post model, using the data in our form for submission.

```
def create
  @post = Post.create(post_params)
end
```

We're using post_params as an argument to the create method so we'd better define that somewhere too. post_params will be a private method, ie. it can't be called from outside this class.

Here we can define the exact data we want to accept as the parameters in our

form. In this case, our image and our caption text. Insert the following code below your other actions.

```
private

def post_params
  params.require(:post).permit(:image, :caption)
end
```

---

Now that the information is being saved in the create action, we need to redirect the user to somewhere useful. Let's send them back to the index action for the time being with the redirect_to method.


---

**What is this magic?**

Add the following code to the bottom of your new action.

```
redirect_to posts_path
```

Which makes our create action look like:

```
def create
  @post = Post.create(post_params)
  redirect_to posts_path
end
```

---

Alright, let's try to submit the form again. Navigate back to localhost:3000/posts/new, add your picture and your optional caption and submit!

Finally, we've avoided errors! But wait, where the hell is your post? Let's quickly set it up now. To make sure our post even exists, jump into your terminal and run your rails console.


---

**I forget how...**

```
bin/rails c
```

---

Now try to find that post we submitted using your ruby and rails skills.


---

**I've lost all my Ruby skills in a freak water slide accident...**

Let's assign the very first post in the Post model to the variable "post".

```
post = Post.first
```

---

And look at what's returned! Our submitted post, complete with caption and image path!

Right then, now that we know that we're actually saving something, let's get ALL of our posts that will be submitted now and in future onto our index path in a big beautiful stream of visual delight.

Create an instance variable under your index action that collects all of the posts in your Post model.  This will let us display all of our posts in our index view!


---

**Please explain this "variable"...**

It's easy peasy!

Let's call our instance variable @posts.  All we then want to do is capture all of the posts in the Post model!

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

---

Once you've done this, we want to iterate over each post in that collection and display it for our viewing pleasure.  We'll need to do this in the index view.  Do your best to iterate this collection using a simple block, outputting the captions and the images.


---

A very simple haml version of this is seen here if you need a hand:

```
-@posts.each do |post|
  =post.image
  =post.caption
```

---

If we refresh our index, all we're getting is a crappy image path on each post. Let's use rail's image_tag helper to actually output our image. You can read up on it yourself here or you can see how to implement this the easy way below.


---

**I don't trust your dodgy links, show me the easy way!**

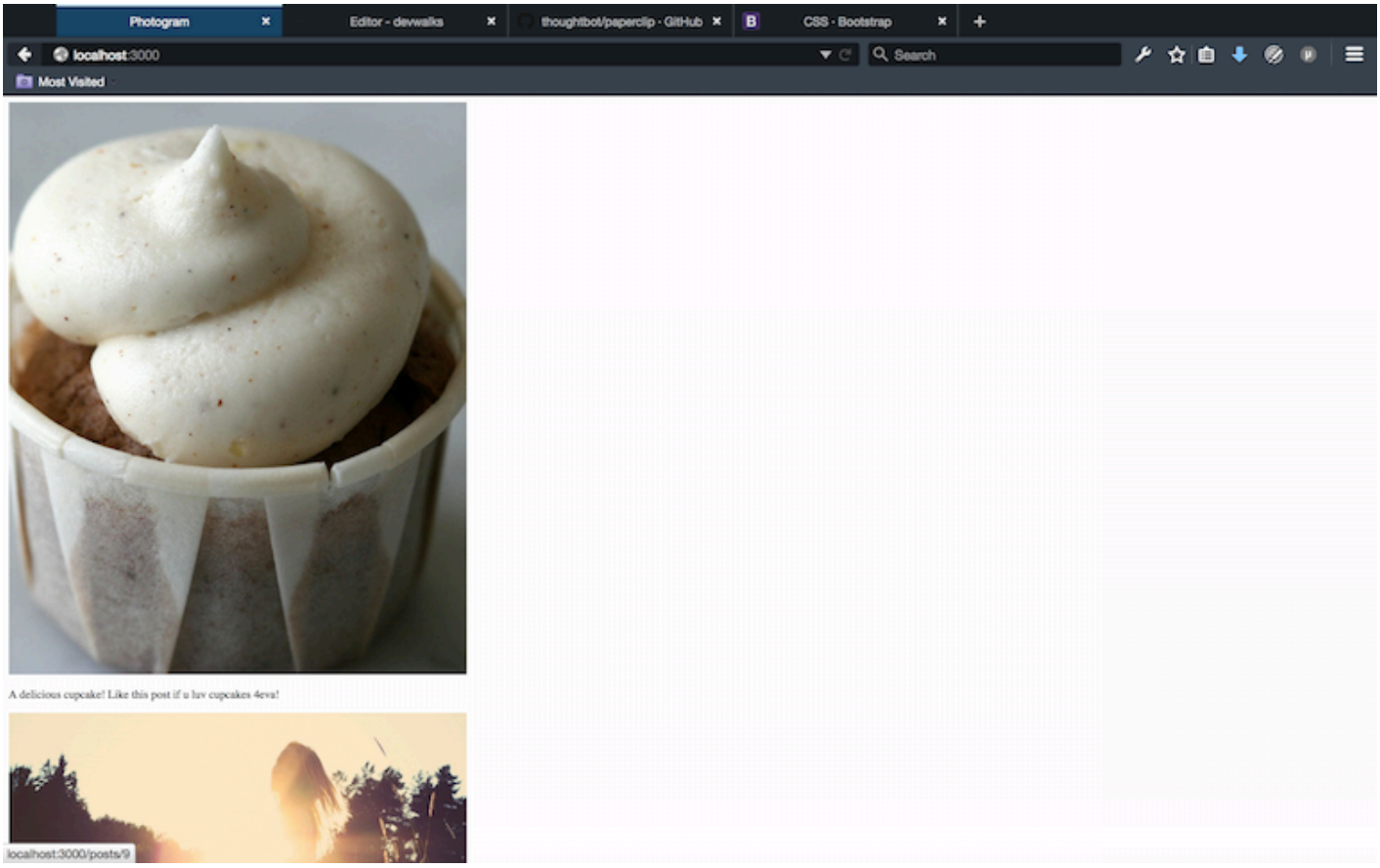
Replace post.image with the below line.

```
=image_tag post.image.url(:medium)
```

---

Now navigate to your index page (http://localhost:3000) and....

*Oh my god. Look at it.*




What an incredible achievement. It's not super functional yet though, I can't even submit a new post without manually navigating to /posts/new and we might want to be able to edit or delete our posts in case we upload a picture of our reproductive organs (accidentally).

Being able to show individual posts would be nice too but that can all wait because I'm sick of this thing looking awful.

### Styling: Let's steal like an artist or something

Let's blatantly steal from Instagram, for this is an Instagram clone tutorial after all!

Instagram has recently revamped their desktop site so I'm just going to do that. BUT to make it easier I'm going to use the CSS framework, Bootstrap as my foundation. Install the bootstrap gem found here in your gemfile as per their docs.


---

You're getting lazy.

```
gem 'bootstrap-sass', '~> 3.3.5'
```

---

Make sure to run "bundle" in your terminal, follow the steps in their docs as far as adding lines to your application.css file (and renaming it to .scss) and adding a line to your application.js file.

Restart your server and you should be good to go. Refresh your index and things look almost exactly the same because we literally had NO styling whatsoever. The h2 looks a bit different which is a nice touch.

Now, let's add a navbar up top and move our company name up there. Look at the bootstrap docs here. It's almost a matter of exactly copying over their suggested navbar and adding it to your layout.html.erb(or .haml) file.

If you're using haml be prepared to translate. I've deleted some extra bits that we won't need below (I've also added some extra bits so make sure to have a peek):

```
%nav.navbar.navbar-default
  .navbar-container
    .navbar-header
      %button.navbar-toggle.collapsed{"data-target" => "#bs-
navbar-collapse-1", "data-toggle" => "collapse", type: "button"}
        %span.sr-only Toggle Navigation
        %span.icon-bar
        %span.icon-bar
      .navbar-brand= link_to "Photogram", root_path
    .collapse.navbar-collapse#bs-navbar-collapse-1
      %ul.nav.navbar-nav.navbar-right
        %li
          = link_to "New Post", new_post_path
        %li
          = link_to "Login", '#'
        %li
          = link_to "Register", '#'
```

I've added this to the application.html.haml view under views/layouts/ because this is shown in ALL of our views. This means we don't have to add the navbar to everything! Huzzah!

Please note that I've added a "New Post" button that links to our new Post action/ route AND the Photogram logo links to the index. Genius.

Let's give each of our individual posts their own full-width row using the bootstrap divs. Remember, try to do it yourself first.


---

**I just want to see how you did it...**

```
-@posts.each do |post|
  .row
    =link_to (image_tag post.image.url(:medium)),
post_path(post)
    %p= post.caption
```

---

Refresh your index page and bask in the glory that is your photo feed.

*But it still doesn't look like the Instagram feed...*

Let's change that now. First let's add our own classes to the html on our index view. This section won't be hidden because it's kind of hard to implement this yourself. If you want the challenge or have the CSS / HTML ability to do so, please do so now. Simply steal Instagrams styling cues and plop them in your application.css and index.html.haml / .erb files!

My implementation looks like this in my index view:

```
.posts-wrapper.row
  -@posts.each do |post|
    .post
      .post-head
        .name
          Ben Walker
      .image.center-block
        =link_to (image_tag post.image.url(:medium), class:'img-
responsive'), post_path(post)
      %p.caption
        =post.caption
```

And my CSS (for the rest of this project) looks like this:

```
body {
  background-color: #fafafa;
  font-family: proxima-nova, 'Helvetica Neue', Arial, Helvetica,
sans-serif;
}

.navbar-brand {
  a {
    color: #125688;
  }
```

```
}

.navbar-default {
  background-color: #fff;
  height: 54px;
  .navbar-nav li a {
    color: #125688;
  }
}

.navbar-container {
  width: 640px;
  margin: 0 auto;
}

.posts-wrapper {
  padding-top: 40px;
  margin: 0 auto;
  max-width: 642px;
  width: 100%;
}

.post {
  background-color: #fff;
  border-color: #edeeee;
  border-style: solid;
  border-radius: 3px;
  border-width: 1px;
  margin-bottom: 60px;
}

.post-head {
  height: 64px;
  padding: 14px 20px;
  color: #125688;
  font-size: 15px;
  line-height: 18px;
  .thumbnail {}
  .name {
    display: block;
  }
}

.image {
  border-bottom: 1px solid #eeefef;
```

```
  border-top: 1px solid #eeefef;
}

.caption {
  padding: 24px 24px;
  font-size: 15px;
  line-height: 18px;

}

.form-wrapper {
  width: 60%;
  margin: 20px auto;
  background-color: #fff;
  padding: 40px;
  border: 1px solid #eeefef;
  border-radius: 3px;
}

.edit-links {
  margin-top: 20px;
  margin-bottom: 40px;
}
```

Whether you've copied me or stolen it yourself, it's time to refresh your index page and look!

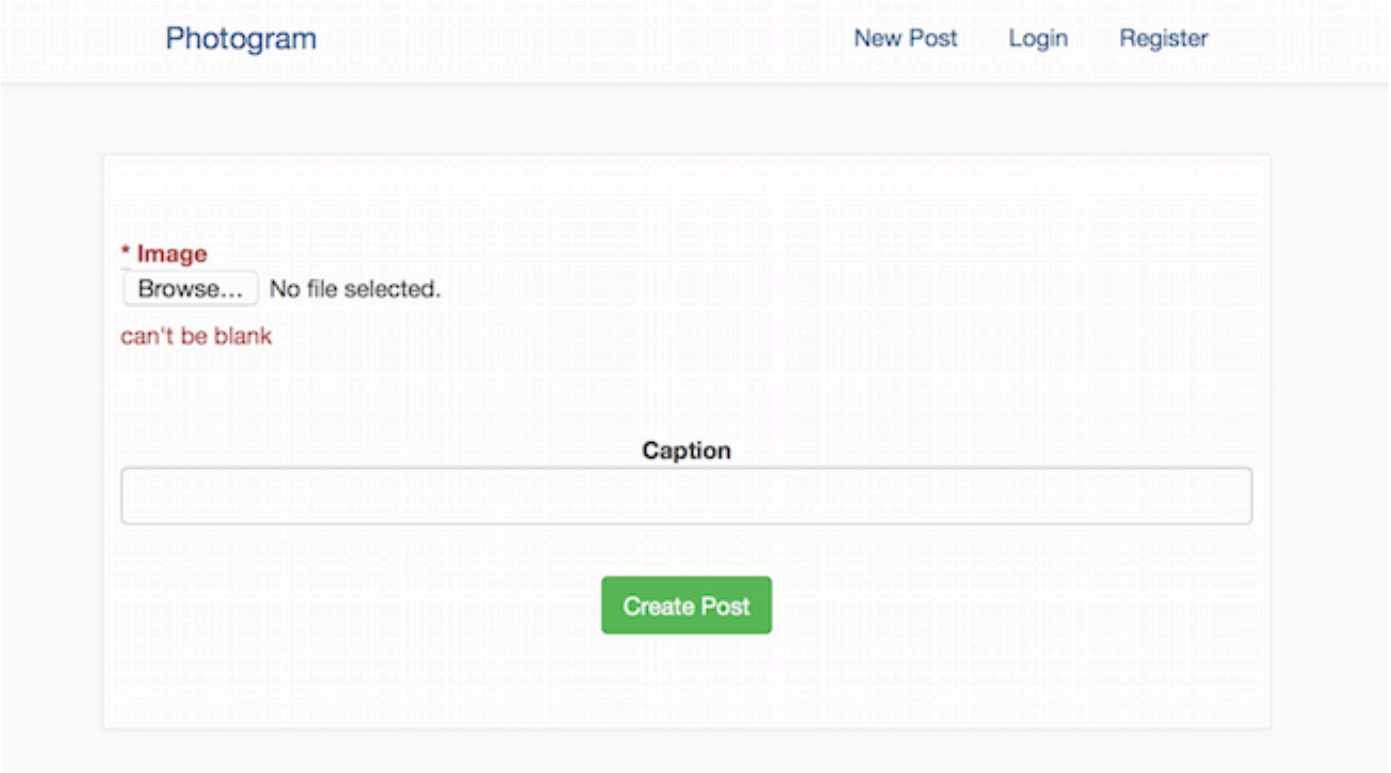

I don't like to give myself too much praise but this is looking pretty bad-ass now. Personally, I much prefer working on a site that looks good. It's my inner artist coming out.

Let's quickly tidy up our new post view and form.

Once again, feel free to do this yourself. Keep in mind that Instagram is a mobile app primarily so you can't actually post photos from your browser. For this reason, you might want to copy my code below, but first, you should try to simply add the bootstrap classes to your form so it looks like the bootstrap standard seen here and then decide on updating styling from there. My version looks like this in the view:

```
.form-wrapper
  = simple_form_for @post, html: { class: 'form-horizontal',
multipart: true } do |f|
    .form-group
      = f.input :image
    .form-group.text-center
      = f.input :caption
    .form-group.text-center
      = f.button :submit, class: 'btn-success'
```

Navigate to the new view if you haven't already and admire your handy work. I'm still not not super happy with the way this form looks but it'll do for the moment.




Let's keep building the apps functionality now.

**Show me a single post you silly application!**

Hey, you talk to Photogram nicely please.

Let's create the ability to show a single post (and later on, all the associated comments & likes with that post.)

First create a show action in your PostsController. You know how to do this by now. We want this action to have a `@post` instance variable that points to the specific post we're referring to.


---

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

---

Now let's fiddle with the image_tag on our index view so that each image will link to the appropriate post!

Combining image_tag and link_to tags will be commonplace in your rails career, work out how to do it via google searches before cheating below.


---

Adjust your image tags to include the link_to helper.

```
=link_to (image_tag post.image.url(:medium), class:'img-
responsive'), post_path(post)
```

---

This will let us click our images in order to take us to the individual post. This takes us one step closer to pure Instagram forgery.

The only issue? We don't have a view for our show action. Create one now that

shows us the same information that can be seen on our index view. The only difference is that we won't be iterating over many posts, we're just referring to one.


---

It's so similar to the index!

```
.posts-wrapper.row
  .post
    .post-head
      .name
        Ben Walker
    .image.center-block
      =image_tag @post.image.url(:medium)
    %p.caption
      =@post.caption
    .text-center
      =link_to "Cancel", posts_path
```

---

Now you should have the ability to click one of your posts on the index and it should take you to the individual post!

### Let me edit my posts, I've made a horrible mistake

We all make mistakes but at least we can fix this one for you. Take a moment and have a think about what we're going to need to do to implement this. In fact, you've probably done this before in your previous applications!

We're going to need some extra actions in our controller.....

We're going to need a view with a form to update our post.....

Give it a go yourself. Hint: You're going to need two extra actions in your posts

controller and only a single view (It will look very similar to your new and create actions). Refer to your old tutorials if you have to, it'll be much more useful than me holding your hand through it.


---

*But your hands are so moist...*

Oh you.

Just like the creation of posts needs a new action and a create action, editing our posts will need an edit action and an update action. Create both an empty edit action and an update action in our posts controller, just like we have many times before with other actions.

```
def edit
end

def update
end
```

Now let's add some logic to the actions. For the sake of the form, we want to include an instance variable that refers to the specific post we're looking at.

Update edit to include the instance variable below.

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

And now, we need to actually perform an update that touches our database (not in a creepy way). It's very similar to our create action but rather than creating, we're UPDATING! Let's make magic.

```
def update
  @post = Post.find(params[:id])
  @post.update(post_params)
  redirect_to(post_path(@post))
end
```

And last but not least, let's create a view for our edit action along with a form. Call this view edit.html.haml (or .erb) and ensure it's residing in the same location as your other post views.

The form within this view will be exactly the same as that used in your new action, so copy and paste the form from your new view to your edit view.

---

We have two shiny new actions in our posts controller and a brand spanking new form in our edit action. Before I continue though, I'd like to add an image to the edit view as well.

I simply want to display the existing image being used for the post. Look at how you display your image in your index view and use the same method in your edit view to display it.


---

I just want to be sure I'm doing it right...

It's OK, I trust that you're doing your best. Simply add a centered div and an image_tag.

```
.text-center
  = image_tag @post.image.url(:medium)
.form-wrapper
  = simple_form_for @post, html: { class: 'form-horizontal',
multipart: true } do |f|
    .form-group
      = f.input :image
    .form-group.text-center
      = f.input :caption
    .form-group.text-center
      = f.button :submit, class: 'btn-success'
```

---

Oh yeah, we're going to need to link to our edit action from our show view, let's quickly add the following code to the bottom of our show view.

```
.text-center.edit-links
  = link_to "Cancel", posts_path
  |
  = link_to "Edit Post", edit_post_path(@post)
```

This will be just under our caption.

That looks a bit better! What if editing alone isn't enough to reverse our terrible actions?

**Please Let me delete my nude selfies.**

OK, but you should know that I'm not happy about it (I think you look great in those chaps). Deleting is super simple and can be fleshed out nice and quickly. First you'll need to create a destroy action in your posts controller that deletes the specific record and then you'll need to link to that action on your index view with the addition of a :destroy method. Then we want to redirect back to the index.

Simply create a link on the edit page for now so all of our users can delete each-others posts at will (We'll fix this later I promise, this is a terrible idea). Go forth and destroy!


---

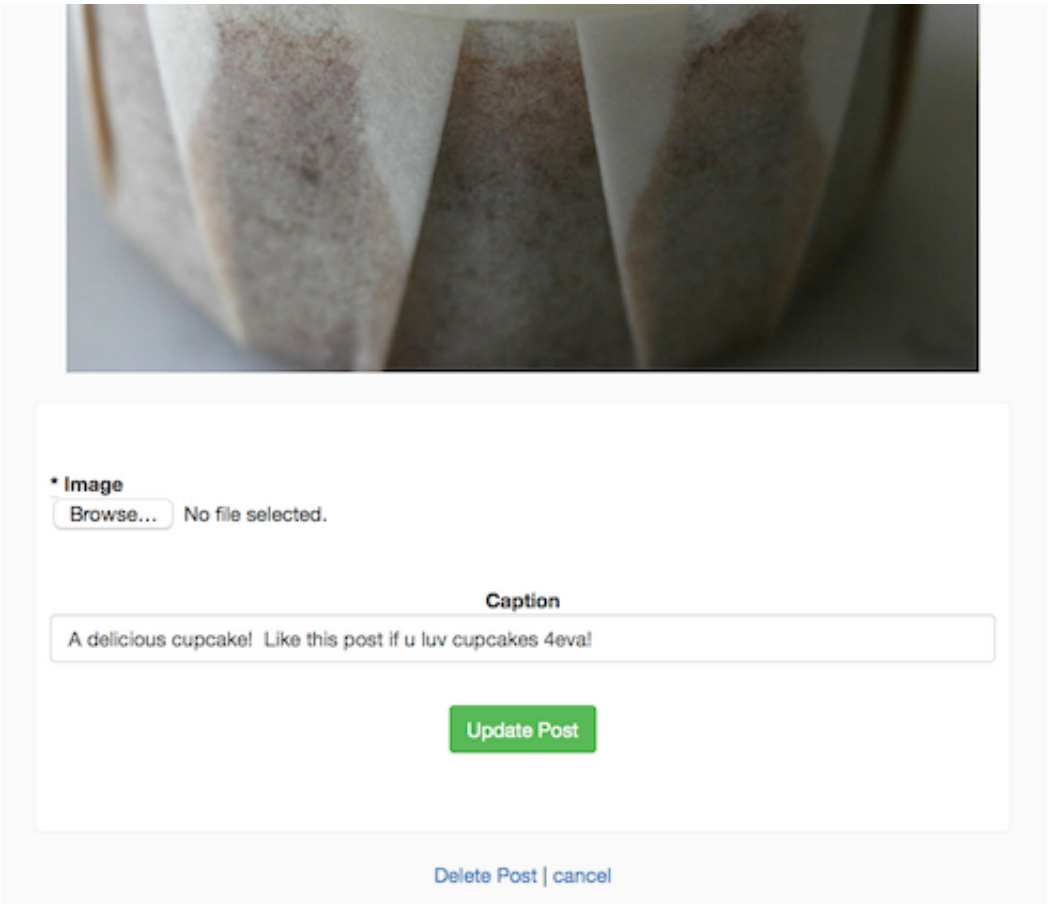
First our destroy action in posts_controller.rb

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

And our simple link to that action with a warning on our edit view:

```
# Add this to the bottom of your edit view.

.text-center.edit-links
  = link_to "Delete Post", post_path(@post), method: :delete,
data: { confirm: "Are you sure you want to delete this post?" }
  |
  = link_to "cancel", posts_path
```

It'll look like this, which is perfect!




And it's done! Try deleting one of your pics now and see what happens.

## Tidy, tidy, tidy

Basic functionality has been created, let's clean up a little before we finish for the moment.

### DRYing up

First, we're repeating ourselves with our edit and new forms, let's move that code into a partial and call it _form.html.haml (or .erb). Copy and paste the form code onto the new _form.html.haml view. Delete that repeated code in both new and edit and then add the following line to both:

```
= render 'form'
```

Remember how the forms differ though, our edit form has the image displayed where the new form does not. Ensure you keep the image in the edit form and add the partial render below that.

### Where else are we repeating ourselves? The controller!

Let's create a private method that sets the post that we're trying to identify using

the id parameter. Create a new private method called set_post that finds the specific post we're looking for by the id parameter and assigns it to the instance variable `@post`. Pro-tip, we're using it in a couple of our controller actions already.

Now we can add a before_action to the top of our controller to set the @post variable for specific actions. This looks like this:

```
before_action :set_post, only: [:show, :edit, :update, :destroy]
```

Go ahead and create the set_post private method now.


---

I wanna check yours...

Well, here's an overview of my PostsController so far.

```
class PostsController < ApplicationController
  before_action :set_post, only: [:show, :edit, :update,
:destroy]

  def index
    @posts = Post.all
  end

  def show
  end

  def new
    @post = Post.new
  end

  def create
```

```
    if @post = Post.create(post_params)
      redirect_to posts_path
    else
      render :new
    end
  end

  def edit
  end

  def update
    if @post.update(post_params)
      redirect_to posts_path
    else
      render :edit
    end
  end

  def destroy
    @post.destroy
    redirect_to root_path
  end

  private

  def post_params
    params.require(:post).permit(:image, :caption)
  end

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

Now that you've got the before action, you can delete each instance variable that achieved the same thing!  Refer to my code above :).

**Useful and Legal Flashing**

I want to achieve two things here, I want to flash the user a message upon completing an action and I also want a fall-back if for some reason, the action couldn't be completed.

First, let's create the plan B if something goes wrong when submitting a form.

Create a simple if / else statement in your create block that reads like this for your index. If your post is successfully created, redirect to the index page, else render the new page. Your update action will be similar but it'll ask, if your post is successfully updated, redirect to the index page, else render the edit page.


---

What does this look like?

```
  def create
    if @post = Post.create(post_params)
      flash[:success] = "Your post has been created!"
      redirect_to posts_path
    else
      flash.now[:alert] = "Your new post couldn't be created!
Please check the form."
      render :new
    end
  end
```

AND

```
  def update
    if @post.update(post_params)
      flash[:success] = "Post updated."
      redirect_to posts_path
    else
      flash.now[:alert] = "Update failed.  Please check the
form."
      render :edit
    end
  end
```

---

We can implement 'flashes' into our `layout/application.html.haml` file with the following code :

```
  .container
    -flash.each do |name, msg|
      = content_tag :div, msg, class: [:alert, alert_for(name)]
    = yield
```

Also add the following code to your `app/helpers/appication.rb` in order to include the styling cues from Bootstrap.

```
def alert_for(flash_type)
  { success: 'alert-success',
    error: 'alert-danger',
    alert: 'alert-warning',
    notice: 'alert-info'
  }[flash_type.to_sym] || flash_type.to_s
end
```

Full credit to suslov on Stack Overflow for this bit of code.

Also, when our users create a new post, edit a post or delete a post, they're not notified of whether what they've done was successful or unsuccessful and this doesn't lead to a great user experience. Let's add some friendly messages that'll be rendered in our user's browsers upon completing an action.

Start from the top action in your controller and work your way down, ask yourself if the user should be told something once that specific action is complete. Create the flash message using the following code and insert it just prior to the redirect or render. If using the render method, adjust the flash message to `flash.now`:

```
flash[:success] = "Your personalised message here"
```

If there's a failure scenario, adjust slightly to this:

```
flash[:alert] = "Oh god something is wrong"
```

(or a version of that that is actually useful to the user).

**CRUD Complete**

You have now created a great CRUD application that looks pretty snazzy, but we're still missing a few things I'd like to see. Users with authentication, comments and likes to be specific. Maybe even a profile page for each user.

Let's keep fighting the good fight in Chapter 2, where we add users, ownership and comments to our application.

# Chapter 2 - Log In & Tell Me I'm Beautiful

Now we can:

- Create posts with an image and caption.
- Show an individual post.
- Edit our posts.
- Delete our posts.

Let's go forth and add some handy new features:

- The ability to create an account and login.  I want the posts I create and the comments that I make to belong to me only.
- I want to have a user name.  My lame email address just won't cut it when I'm interacting with others.
- I want to be able to comment on my own posts and others.
- I want to delete the above posts the next day once regret sets in.

Sound good?  Of course it does! Let's get going.  Oh yeah, it'll look like this by the end:


Let's start this part of the journey with… users!

## Devise: You want to give yourself a baddass user name, I understand.

Of course you do! I do too… In fact, who doesn't want the ability to give your interactions *personality* within this anonymous pit that is the web?

Let's give our application and our users what they want, let's let them create and use user accounts!

For this purpose I'm going to use the incredibly popular authentication gem: Devise. Why not build my own? Well, first, other tutorials can show you that if you *really* want to know (it's certainly worth doing at least once). In fact, the Devise guys recommend you don't use Devise for your first experience with Rails authentication.

*Second*, I want this series to be about you building familiarity with Rails. Which

means building familiarity with incredibly popular Rails gems. Plenty of dev shops use Devise and it's nice and easy to use if you want to build your own user auth features for your own app.

**The point is: It's worth learning, so let's learn it.**

Checkout the devise docs here and have a quick read. Read the installation instructions and the features that devise has.

Read it all? Great! Install it as per their instructions. Add gem to gemfile, run their installer and so on. Call your Devise model `User`. Do your best (and don't worry about the user name functionality just now..).


---

We're just warming up again for this article so let me walk you through the installation process.

Yes, I know you could've done it yourself. No, I don't think you're silly.

As per the Devise docs:

- Add the gem to your gemfile like so:

```
gem 'devise'
```

- Run bundle in your terminal (at the root of your application folder).

```
bundle install # or just bundle for the super-efficient readers.
```

- Question the meaning of life (optional).
- Install Devise via terminal by running their generator.

```
rails g devise:install
```

- Have an existential crisis (not optional).
- Follow the instructions that have just appeared in your terminal, thanks to the devise generator.
    - Add the default_url_options to your `development.rb`.
    - Make sure we have a root url (we do). You can check your `routes.rb` file if you don't trust me (or just visit your roots route in your browser).
    - Check we have flash messages in our application.html.haml( or .erb) file? You check and tell me.
    - We're not using Rails 3.2 so don't worry about point 4.
    - We can customise our views as per point 5 (which we will do because we want a good looking site overall, no offence Devise).

Now that we've finished the above instructions, let's continue. First, let's create our Devise 'User' model by tappety tapping the following in your terminal:

```
bin/rails g devise User
```

Before you migrate your newly created migration, jump into the migration file found here: `db/migrate/*last_file*.rb`.

Have a read.

- Do the columns that are being created within the table make sense?
- Note how the creators have commented on the columns based on the different optional features Devise provides you?
- Notice some parts of the migration are commented out? If we wanted those additional features we could un-comment them, prior to migrating the changes.

Comfortable with your new creation? Good! It's time to migrate your database by running the following code in your terminal.

```
 bin/rake db:migrate
```

Oh yeah, last but certainly not least, let's copy those Devise views as mentioned earlier. In your terminal, write:

```
rails g devise:views
```

Now we can easily access the Devise views for things such as new registrations, logins, password reminders etc etc with ease! We will want to change these at a later point in order to make it match the rest of our application's styling.

---

**Sneaky Extra Section**

Want to auto-convert those default Devise .erb views to haml? Well check this out.

Still interested? Well install the html2haml gem via your terminal and run the commands as stated. Beautiful haml views will be returned!

**End of sneaky section.**

And that's it, you're officially ready to rumble and start adding Devise functionality to your application!

Let's just have quick moment to consider what we've done here. We've really blitzed a lot of things in a relatively short amount of time.

By following the Devise instructions thus far we've now got the ability to:

- Visit specific paths to create a new user, sign in with a user, log out with a user.
- Create functionality around letting only signed in users perform certain actions or visit certain areas of our application.
- Access a range of Devise helper methods. One that will let us refer to the `current_user`when creating, editing and deleting posts.

We're not done though, oh no. Now we'll have to adjust our application to fit this new concept of 'users'.

## Changing stuff to make stuff work properly

Devise is installed and our application uses approximately none of it's features so far. Let's change that fact this very instant.

Run your rails server in your terminal. Once loaded, navigate over to `localhost:3000/users/sign_up` and check out our Devise registration form. Nice! But wait a second, it's only asking us for email and password... I wanted a cool username!

## Adding custom columns to Devise users

### Adjusting our User Model

To get our badass usernames working, we're going to have to add a new column in our users table and we're also going to need to make sure our Devise forms accept 'username' as an allowed parameter. You should google for a solution to this first, how do we add custom fields / columns to Devise?


---

What did you find? Did you even search? Of course you did, I trust you unconditionally.

Because you did a *very thorough* search, you might be there, sitting on that old chair of yours thinking, "Oh god, this is tricky", but fear not good friend, I'm here to re-reassure you that everything is going to be A-O.K. Let's make magic together.

First, let's add the new column to our User table by generating a migration file and migrating. In your terminal move your fingers on your keyboard until this appears:

```
bin/rails g migration AddUserNameToUsers user_name:string
```

Before we get too excited with migrating our new file let's index the user_name column and make sure each user name is unique.

Add the code below to your migration prior to running it. Open the file via `db/migrate/*last-file-in-that-folder*`. You want this line of code to still be within the 'change' method of the migration.

```
add_index :users, :user_name, unique: true
```

Once you're extremely happy with your migration file, run migrate in your terminal.

```
bin/rake db:migrate
```

And to think your school teachers said you'd amount to nothing!

Now we've got a user_name string column in our User table.

While you're on this current roll, let's quickly add a validation to our `user.rb` model. Why, you ask? Because we **DEMAND** a user name from our users and we also want to set some limitations around user name lengths and such. I don't want random people on Photogram called 'a' or 'b', that's pure insanity and you know it.

Just below the `class User` line in the model, add:

```
validates :user_name, presence: true, length: { minimum: 4,
maximum: 16 }
```

Fiddling With our Views**

How about we add the user_name field to our registration view? This way, the user will have to create a user name at registration time, just what I want.

Open `app/views/devise/registrations/new.html.haml` and add an input to the form for user_name below the email input. It should look similar to:

```
= f.input :user_name, required: true
```

Add the same line to your `app/views/devise/registrations/edit.html.haml` file in the same position. This just means our user has the ability to edit their user name in the future if they so desire.

Alright, so view is sorted and model is sorted. Naturally you'd expect that if you were to navigate to the sign up page on your server (`localhost:3000/users/sign_up`) you'd think that your brand new field is good to go.

**Well think again!**

The fact is that in Rails 4 we have to specifically state what we'll be accepting in our submitted forms. We looked at this previously when creating the private `post_params` method in the last article. With Devise we'll have to allow our user_name to pass through in a way that's a little trickier.

We'll have to create our own controller that inherits from the original Devise Registrations Controller. This way we can allow the extra data we want without too much hacking.

In your controllers folder, create a new file called `registrations_controller.rb`. Copy or tap away on your keyboard until the following code appears:

```
class RegistrationsController < Devise::RegistrationsController

  private

  def sign_up_params
    params.require(:user).permit(:email, :user_name, :password,
:password_confirmation)
  end

  def account_update_params
    params.require(:user).permit(:email, :user_name, :password,
:password_confirmation, :current_password)
  end
end
```

Now we need to make sure Devise is looking at our newly created controller for the `sign_up_params` and `account_update_params` that are allowing the user_name parameter to pass through. Add the following line to your routes.rb file. Replace the `devise_for :users` line with the code below.

```
devise_for :users, :controllers => { registrations:
'registrations' }
```

Full credit for the above Devise functionality goes to Jaco Pretorius over at his blog. I'd give him a high five if possible.

---

Now we're officially all done! Try creating a new user with a badass user name via your sign up page.

You *should* be redirected to your home page and get a lovely default welcome message.

Let's quickly check that the user creation process all worked ok. In your terminal run the rails console with `bin/rails c`. Let's see what information we have for our first user. Type:

```
user = User.first
```

and see what's returned. Does `user.user_name` return the user name? It should, because you're awesome and Ruby knows it!

I'm super stoked with how we're going. Let's now make sure each post *belongs* to a single user.

### My post are wittier than yours!

At the moment, anyone can create an individual post and that post doesn't actually *belong* to anyone. Which means anyone can do anything to it! This is completely outrageous and must be fixed immediately.

Let's change a few things.

1. Let's adjust our navbar so it shows specific links based on whether the user is logged in or not. We'll make sure they link to the right spots too.
2. Let's make sure that only logged in users can create a new post and view existing posts.
3. I want each post to belong to the user who created it.
4. I want to show the user name of the creator of the post above each post.
5. I want only the owner of the post to be able to edit or delete it.

You. Me. These features. Now.

# 1. The Clever Navigation Project

Our navbar lives here: `app/views/layouts/application.html.haml` for the moment. Let's move it into its own partial view and add some logic with regards to displaying specific information for logged in users. This will tidy up our application view and let us separate functionality, a good thing.

Create a new file under the 'layouts' folder called `_navbar.html.haml`, in this

folder, you want to relocate all of the code relating to your navbar. This looks like:

```
%nav.navbar.navbar-default
  .container-fluid.navbar-container
    .navbar-header
      %button.navbar-toggle.collapsed{"data-target" => "#bs-
navbar-collapse-1", "data-toggle" => "collapse", type: "button"}
        %span.sr-only Toggle Navigation
        %span.icon-bar
        %span.icon-bar
      .navbar-brand= link_to "Photogram", root_path
    .collapse.navbar-collapse#bs-navbar-collapse-1
      %ul.nav.navbar-nav.navbar-right
        %li
          = link_to "New Post", new_post_path
        %li
          = link_to "Login", '#'
        %li
          = link_to "Register", '#'
```

Where that code used to belong in `layouts/application.html.haml` insert:

```
= render 'layouts/navbar'
```

Now it's time to add some logic to our newly separated partial view. We're going to use the Devise helper `user_signed_in?` to determine whether the user is in fact signed in or not. Where your current list elements are, replace with this:

```
%ul.nav.navbar-nav.navbar-right
  - if user_signed_in?
    %li
      = link_to "New Post", new_post_path
    %li
      = link_to "Logout", destroy_user_session_path, method:
:delete
  - else
    %li
      = link_to "Login", new_user_session_path
    %li
      = link_to "Register", new_user_registration_path
```

Makes sense, right? *If* the user is signed in, display the "New Post" and "Logout" links. Otherwise, display the "Login" and "Register" links. Those links point to the appropriate routes based on what Devise provides us!

Want to know where I got those paths from for the links? If you were to run `bin/rake routes` in your terminal, you'll quickly see the routes we can use, thanks to Devise.

What next? Well, anyone can still do anything on our application. Let's crush their dreams and block access to certain parts of the site.

# 2. Photogram users only. We're super exclusive.

Jump back onto the Devise docs. At that link, notice the `authenticate_user!` helper? Hit command + F and search for it if not.

The reason it's useful is that by adding a before_action to our posts controller with this helper, we can block any unauthorised user from accessing any actions we define.

For the moment, let's make sure only registered (and logged in) users can access ALL of our actions.

At the top of your `posts_controller.rb` file add the before action.


---

**Oh come on, you can do this!**

Just below the first line, add the before filter:

```
before_action :authenticate_user!
```

---

All done? Great. Now jump back onto your running version of the app (or run `bin/rails s` in your terminal if it's not already running) and navigate to the root route. Can you see stuff? Great, you're logged in! Try logging out now by clicking the link in your navbar.

What happens??

You're blocked! Try navigating to `localhost:3000/posts/new`.

**NOPE!**

It isn't going to happen. Cool huh?

# 3. I want my own posts!!!

Yes I'm throwing a tantrum and no I don't care. How can I show the world how amazing my life is via photos if people don't know it's MY photos they're looking at??!?!

It's time to personalise our posts.

Let's think about the relationship that our users will have with the posts. You've dealt with relational databases before, how will this work?

I think a User will have many Posts and a Post will belong to a User. Happy with that?

Good.

Let's add the relationship to our models. Add the following to `models/post.rb`:

```
belongs_to :user
```

And this next line to `models/user.rb`:

```
has_many :posts, dependent: :destroy
```

This is all well and good, but we now need a way for these tables to reference each other, we need our posts table to have a reference to the creator's user id.

We're going to need to generate another migration for this purpose. You know what? I think you should try to do it first. Generate a migration file that'll add a user id to the posts table.


---

Solution time! In your terminal, run the following:

```
bin/rails g migration AddUserIdToPosts user:references
```

---

As per usual, you can check out the result in your `db/migrate` folder, it'll be the last file.

Happy with it? Migrate in your terminal with:

```
bin/rake db:migrate
```

Now have a look in your `db/schema.rb` file. Notice how the posts table now has a `user_id` column? This is great! This means we can now link each post to a specific user through their id! In fact, I want to go a step further. I **DEMAND** that each post **MUST** have an associated user!

At the top of the post model (below the class line) add:

```
validates :user_id, presence: true
```

This simply means that with every new post object that's created we *need* an associated user_id. Test it for yourself, try to create a new post in your application

now and see what happens.

You'll be redirected to the index as per usual and you'll even get a nice flash message, but where's your new post? It doesn't exist!

You fell for my trick! Ha!

Let's fix this now and stop tricking our friends by incorporating the current user into the create action within our posts controller.

Want to be awesome and give it a go yourself? Go ahead! Maybe commit your changes to git first though. Just in case...


---

Adjust the code within the post controllers' `new` and `create` actions.

```
  def new
    @post = current_user.posts.build
  end

  def create
    @post = current_user.posts.build(post_params)

    if @post.save
      flash[:success] = "Your post has been created!"
      redirect_to posts_path
    else
      flash[:alert] = "Your new post couldn't be created!
Please check the form."
      render :new
    end
  end
```

Make sense? For the new action, we're creating a new current_user.posts object

for the sake of our form and in the create action, we're creating that object using the post_params and either saving it or not.

Awesome.

# 4. Identify Yo Self

Now, how can we identify which post belongs to who? How about in the header of each post?

Try it yourself.

Replace "Ben Walker" (great name) in each post with the user name of the user who posted it! You'll have to chain together a few methods to achieve this, just try to think about the relationship of the post to the user and the user_id to the user.


---

How'd you go?

Here's how I did it.

I opened my index view and replaced `Ben Walker` with:

```
= post.user.user_name
```

Once you've done that, jump back onto your site and refresh your index. Get an error?

That's OK!

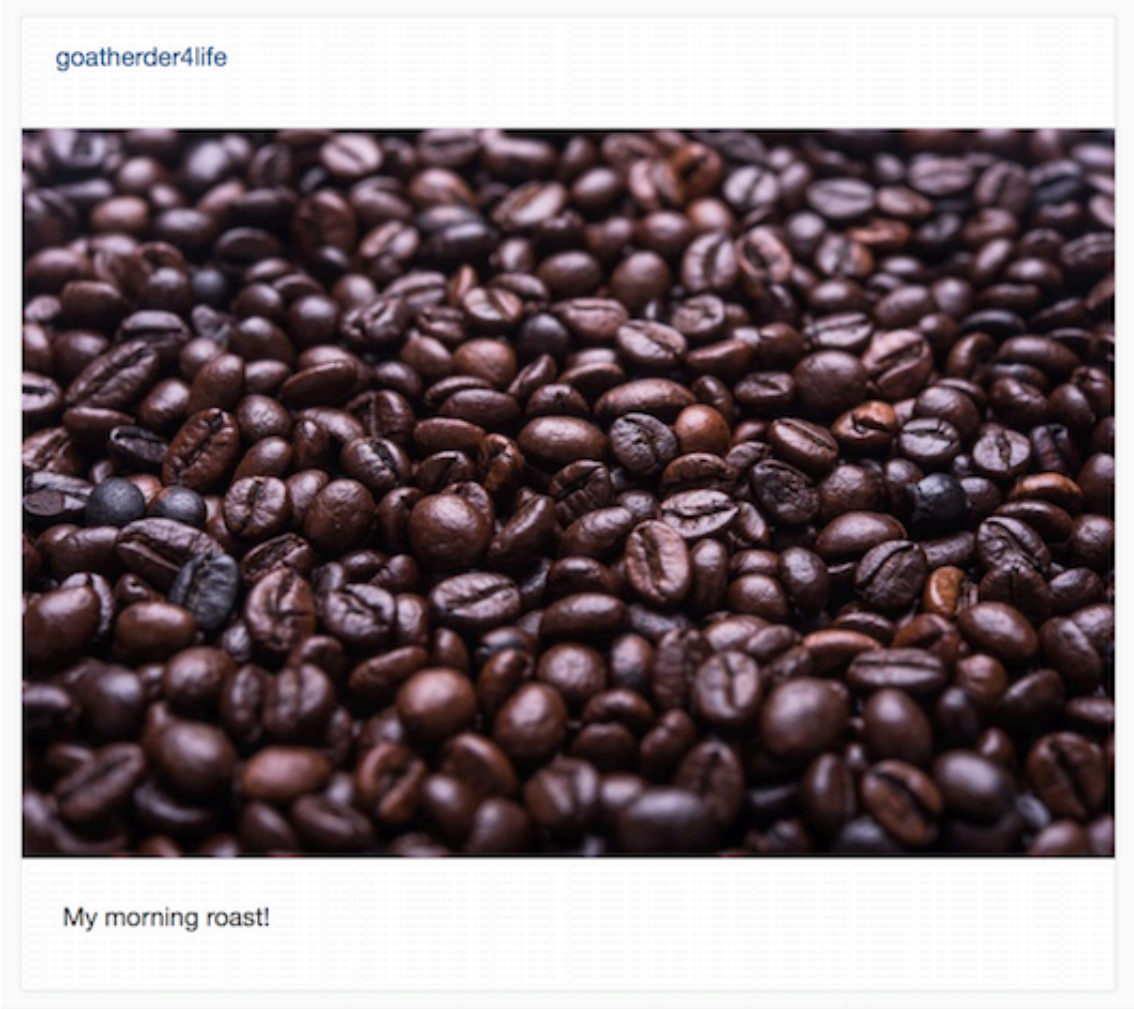
It's only because some of your posts at the moment don't have an association to a

user. You've got a couple of options here. You can:

1. Open the Rails console and manually add a user_id to the posts that *don't* have one existing (the posts that were created prior to implementing users).
2. Undo the change to your index and manually delete the posts that you think don't have an associated user_id. Adjust the index again and re-try.
3. Delete all of your existing posts via the Rails console and start fresh.

---

Check this out!




I'm super proud of this, it takes us one step closer to an actual Instagram clone with some pretty crucial functionality.

Now that you've absorbed how to do this, go ahead and adjust the `show` view so that it also shows the user name of the creator instead of 'Ben Walker', just like we did above.

# 5. Protecting Personal Posts

Let's now adjust a few things within our application so that only the creator of a post can edit or delete it. This is naturally a good idea and should prevent complete anarchy from prevailing in the short term.

First, let's adjust the logic in our views so that the edit and delete links are shown on an individual post only if you have the ability to perform those actions.

In the `show` view, create an if / else statement that compares the user id of the creator of the post to the user id of the current user. If they match, show the 'Edit Post' link, otherwise, just show the cancel link.


---

Did you have a win? Here's how I implemented this feature:

```
- if @post.user.id == current_user.id
  .text-center.edit-links
    = link_to "Cancel", posts_path
    |
    = link_to "Edit Post", edit_post_path(@post)
- else
  .text-center.edit-links
    =link_to "Cancel", posts_path
```

Pretty simple right? If the user id of the post we're looking at matches the user id of the currently logged in user, we'll show you the 'Edit Post' path. Otherwise? Only 'cancel' for you I'm afraid.

---

How secure is this implementation though? If you haven't tried already, log out and create another user. Log in and then click on one of the posts that you'd

created with the last user.

Great, that 'Edit Post' button is missing! But wait, what if we just navigate to `localhost:3000/posts/#idnumber/edit`? (Replace that id number there with one you actually have.)

We can edit and delete other people's posts! Have a think about how you could ensure that only the owner of the post can access that edit and delete action. Try to implement it now.


---

Here's how I solved this problem. First, I created a private method in the posts controller below the other two private methods. It looks like this:

```
def owned_post
  unless current_user == @post.user
    flash[:alert] = "That post doesn't belong to you!"
    redirect_to root_path
  end
end
```

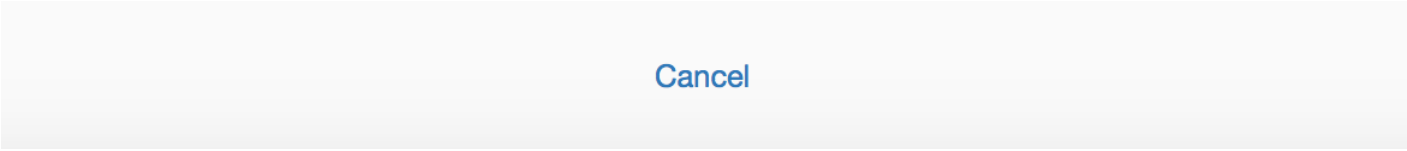
Then I inserted a before_action at the top of the controller, specifying the owned_post method for the edit, update and destroy actions only.

```
before_action :owned_post, only: [:edit, :update, :destroy]
```

So unless you meet the requirements set by the `owned_post` method, you'll be redirected back to the root route with a nice little message *before* any of the actions in the controller are called.

---


My post.




Not my post.

How good is this! We've setup what we wanted to for our Devise implementation so far. Some other features I'd love to have in this regard would be:

- Having a page for each user that only shows their posts.
- Following / unfollowing other users.
- Let other users follow / unfollow me.
- Have my own profile picture that is shown along with my posts and comments.

Guess what? We're going to build those features in the next articles. But for now..

## I want to comment on your photo, that's a killer selfie and you deserve to know it!

What an incredibly composed selfie! The lighting, the focus, the angle of your nose in relation to your cheeks! The fact that I can't comment on such a beautiful image is a crime against humanity, so let's build the functionality now.

We'll create comments for posts by doing a few things. Here's some of the stuff we're going to need:

- A comment model to store the lovely (and offensive) comments.
- A comment controller to handle the comment actions in a similar fashion to our post actions (think 'create' and 'destroy').
- A comment form for the index and show views so we can submit and / or delete our comments.
- A list of comments for each post so we can see our own and others comments too.

First, let's generate a model for the comments. In your terminal, generate a model that has these columns:

- user:references
- post:references
- content:text

Not sure about 'references'? What a great time to perform some self-driven research!


---

**Need help hombre?**

I generated my model by running the following code in my terminal:

```
bin/rails g model Comment user:references post:references
content:text
```

Once generated, jump into the `app/models/comment.rb` file and ensure your model looks like this:

```
class Comment < ActiveRecord::Base
  belongs_to :user
  belongs_to :post
end
```

Finally, jump into the `app/models/user.rb` and `app/models/post/rb` files and add the following line to each:

```
  has_many :comments, dependent: :destroy
```

This means that both users and posts "have many" comments and finishes off creating our associations between the models!

The dependent: destroy line means that if that object is destroyed, the associated

objects will be destroyed too. In practice, this'll mean that if a User is destroyed, all of their associated comments will too. If a Post is deleted, say "Goodbye" to the comments too!

---

Now that our model / database table / associates are sorted, let's add our comments as a 'nested route' within our posts.

You can read more about Rails routing here if you want to add the nested route yourself or simply read how it works. Once you've had a read, go ahead and add the nested route in your routes.rb file!


---

Open up your routes.rb file in `config/routes.rb` and adjust your posts resources like so:

```
resources :posts do
  resources :comments
end
```

---

Alright, so we've got somewhere to store our comments and we've got sensible routes too.

Now we need a controller and a form. First, the controller!

Generate a new controller for the comments in your terminal now.


---

In your terminal, type:

```
bin/rails g controller comments
```

---

Now that we have an empty controller, it's time to add the actions that we want. Let's add a create action first. Make it blank to start with, and then have a ponder about what logic we'd like to include in that action. The feature will work like this.

- On the posts index view and the posts show view, a form will exist under each post for us to add a comment if we so desire.
- By writing something in that form and submitting, we save the comment to that post and it identifies who made the comment.
- We're redirected back to wherever we were after submitting the form.

Have a think about this will work and then try to implement the feature yourself. I believe in you.


---

Well, here's how my create action looks (as well as a bit more of the controller), it's probably not as good as yours to be honest, you did a fantastic job just now.

```
before_action :set_post

def create
  @comment = @post.comments.build(comment_params)
  @comment.user_id = current_user.id

  if @comment.save
    flash[:success] = "You commented the hell out of that post!"
    redirect_to :back
  else
    flash[:alert] = "Check the comment form, something went
horribly wrong."
    render root_path
  end
end

private

def comment_params
  params.require(:comment).permit(:content)
end

def set_post
  @post = Post.find(params[:post_id])
end
```

---

Make sense? At the very top we run the `set_post` `before_action` which we can see at the bottom of the snippet in the private method area. We're just setting the `@post` instance variable to the post from the Post model based on the post_id params.

Looking at the create action itself, it should be reasonably self-explanatory. We build the new `@comment` object and then assign it the user_name field based on the user currently logged in. If the comment is saved, we get a lovely message and are redirected back. If not, we get sad.

While we're at it, let's add a destroy method to our comments controller. Implement it now if you dare. You've done this before after all!


---

Below my create action, I have the following:

```
def destroy
  @comment = @post.comments.find(params[:id])

  @comment.destroy
  flash[:success] = "Comment deleted :("
  redirect_to root_path
end
```

Pretty simple stuff! It looks very similar to the way we handled deletion of posts in Part 1.

---

So we've got a model, we've set up our routing appropriately and we have our controller doing important things when those actions are called. Let's now implement the ability to *actually create and delete comments* in our user-facing app. Let's attack our views!

Do you want to attack your own views and create this functionality? I'll be honest, it was a bit tricky but hey, learning new things isn't meant to be easy! Give it a go. Add a form to add a comment below the caption area of each post in your index.

My solution is below! Please note I've allowed a spot here to actually show the comments for each post too. I'm pretty sneaky like that.

```
.posts-wrapper.row
  -@posts.each do |post|
    .post
      .post-head
        .thumb-img
        .user-name
          =post.user.user_name
      .image.center-block
        =link_to (image_tag post.image.url(:medium), class:'img-
responsive'), post_path(post)
      .post-bottom
        .caption
          .user-name
            = post.user.user_name
          = post.caption
        - if post.comments
          - post.comments.each do |comment|
            .comment
              .user-name
                = comment.user.user_name
              .comment-content
                = comment.content
              - if comment.user == current_user
                = link_to post_comment_path(post, comment),
method: :delete, data: { confirm: "Are you sure?" }
        .comment-form
          = form_for [post, post.comments.new] do |f|
            = f.text_field :content, placeholder: 'Add a
comment...'
```

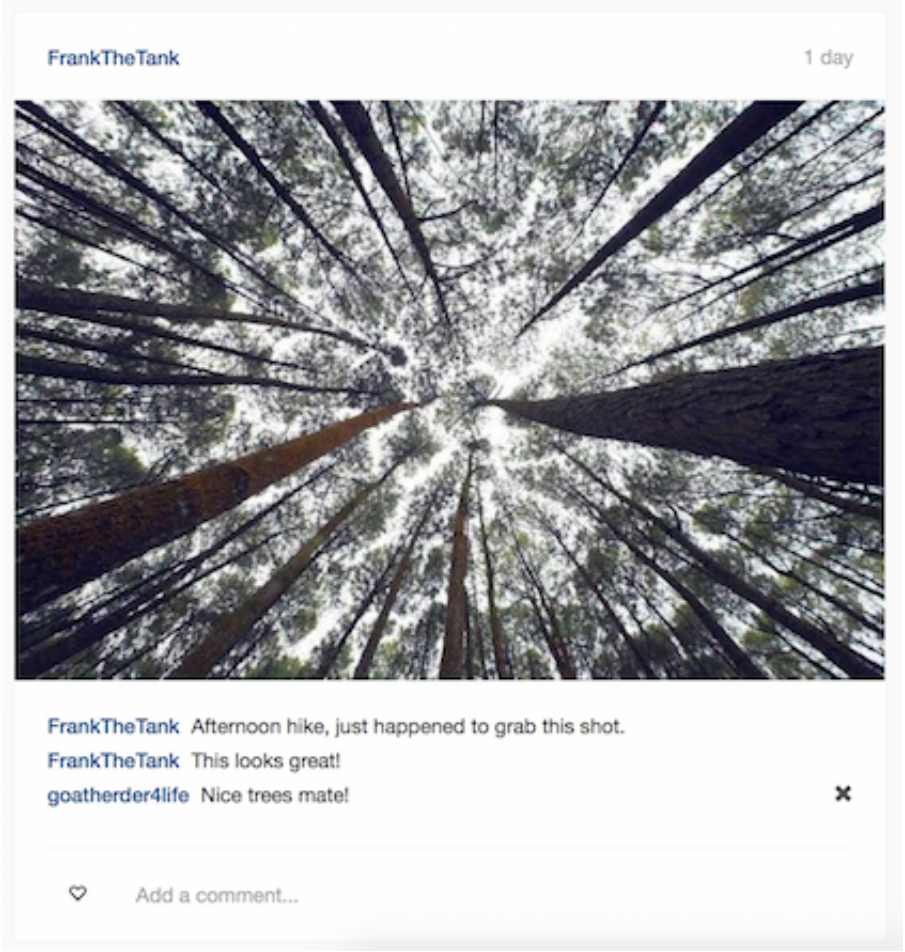
A few things here. First, I didn't use `simple_form` for this form because it wasn't worth the hassle, it was much cleaner to use the standard Rails form_for.

Second, notice that link above the comment-form? Yep, we're letting the user delete comments that belong to them.

While I think we're making great progress, something is wrong... the app is just so... ugly now... Check out that new comment form you've just made via your index page. Gross.

### Let's fix the ugly! (and add a touch of extra functionality)

This is how each post will look after this small makeover:




This is great!

I'm super excited about how close we're getting to Instagram-ness here so let's get right into it.

First, for the sake of tidiness and being a good developer, let's not repeat ourselves and drag our haml code for each individual post into it's own partial view. That way, we can use it for both the show and index views and not repeat ourselves!

The partial view should be called `_post.html.haml` and it should live under the `views/posts` folder. You want it to look like this (I've added some different divs for styling):

```
.posts-wrapper
  .post
    .post-head
      .thumb-img
      .user-name
        = post.user.user_name
    .image.center-block
      = link_to (image_tag post.image.url(:medium), class:'img-
responsive'), post_path(post)
    .post-bottom
      .caption
        .user-name
          = post.user.user_name
        = post.caption
      - if post.comments
        - post.comments.each do |comment|
          .comment
            .user-name
              = comment.user.user_name
            .comment-content
              = comment.content
            - if comment.user == current_user
              = link_to post_comment_path(post, comment),
method: :delete, data: { confirm: "Are you sure?" } do
                %span(class="glyphicon glyphicon-remove delete-
comment")
    .comment-like-form.row
      .like-button.col-sm-1
        %span(class="glyphicon glyphicon-heart-empty")
      .comment-form.col-sm-11
        = form_for [post, post.comments.new] do |f|
          = f.text_field :content, placeholder: 'Add a
comment...'
```

Now that we can render this partial for the show and index views, let's adjust them to suit. Adjust your `index.html.haml` so it looks like:

```
-@posts.each do |post|
  = render 'post', post: post
```

Make sense? We're still iterating through each post and for each of those posts we're rendering our `post.html.haml` partial.

We're passing `post` from the index block to the partial as `post` too, so the partial know's how to interpret it. Check out how we do this in the `show` view below to reinforce the concept.

Here's the *whole* of the `show.html.haml` view now:

```
= render 'post', post: @post
- if @post.user.id == current_user.id
  .text-center.edit-links
    = link_to "Cancel", posts_path
    |
    = link_to "Edit Post", edit_post_path(@post)
- else
  .text-center.edit-links
    =link_to "Cancel", posts_path
```

We're rendering the `_post.html.haml` partial, passing the `@post` instance variable as `post`, which is how we refer to the post in the partial! Want to experiment? Try just rendering the 'post' partial without the second argument. What happens?

Now we've cleaned up our code a little, we still have a problem. Our posts remain ugly.

Let's add the CSS to beautify things! Replace your current `application.scss` file with the code below. Don't delete the bootstrap imports though, you crazy cat. I know how much you like to delete things.

```
body {
  background-color: #fafafa;
  font-family: proxima-nova, 'Helvetica Neue', Arial, Helvetica,
sans-serif;
}

/* ## NAVBAR CUSTOMISATIONS ## */

.navbar-brand {
  a {
    color: #125688;
  }
}
```

```
.navbar-default {
  background-color: #fff;
  height: 54px;
  .navbar-nav li a {
    color: #125688;
  }
}

.navbar-container {
  width: 70%;
  margin: 0 auto;
}

/* ## POST CUSTOMISATIONS ## */

.posts-wrapper {
  padding-top: 40px;
  margin: 0 auto;
  max-width: 642px;
  width: 100%;
}

.post {
  background-color: #fff;
  border-color: #edeeee;
  border-style: solid;
  border-radius: 3px;
  border-width: 1px;
  margin-bottom: 60px;
  .post-head {
    flex-direction: row;
    height: 64px;
    padding-left: 24px;
    padding-right: 24px;
    padding-top: 24px;
    color: #125688;
    font-size: 15px;
    line-height: 18px;
    .user-name, .time-ago {
      display: inline;
    }
    .user-name {
      font-weight: 500;
    }
    .time-ago {
```

```
      color: #A5A7AA;
      float: right;
    }
  }
  .image {
    border-bottom: 1px solid #eeefef;
    border-top: 1px solid #eeefef;
  }
}

.post-bottom {
  .user-name, .comment-content {
    display: inline;
  }
  .caption {
    margin-bottom: 7px;
  }
  .user-name {
    font-weight: 500;
    margin-right: 0.3em;
    color: #125688;
    font-size: 15px;
  }
  .user-name, .caption-content {
    display: inline;
  }
  #comment {
    margin-top: 7px;
    .user-name {
      font-weight: 500;
      margin-right: 0.3em;
    }
    .delete-comment {
      float: right;
      color: #515151;
    }
  }
  margin-bottom: 7px;
  padding-top: 24px;
  padding-left: 24px;
  padding-right: 24px;
  padding-bottom: 10px;
  font-size: 15px;
  line-height: 18px;
}
```

```
.comment_content {
  font-size: 15px;
  line-height: 18px;
  border: medium none;
  width: 100%;
  color: #4B4F54;
}

.comment-like-form {
  padding-top: 24px;
  margin-top: 13px;
  margin-left: 24px;
  margin-right: 24px;
  min-height: 68px;
  align-items: center;
  border-top: 1px solid #EEEFEF;
  flex-direction: row;
  justify-content: center;
}

/* ## Wrapper and styling for the new and edit views ## */

.form-wrapper {
  width: 60%;
  margin: 20px auto;
  background-color: #fff;
  padding: 40px;
  border: 1px solid #eeefef;
  border-radius: 3px;
}

.edit-links {
  margin-top: 20px;
  margin-bottom: 40px;
}
```

Jump back onto your index page and refresh.  Good?

Good.

Now that we can stop gloating about how brilliant our application currently looks, I want to think about the functionality of our application.

Have you tried submitting a comment yet? It's not very nice is it?  You hit enter,

you're taken to the top of the page and are given the crappy message.

Every time I write a message, I can't help but think one thing…

> *This feels nothing like Instagram.*

And for good reason! It's not! When I comment on someones post on Instagram, it's beautiful! I simply write my scathing personal criticism in the comment box and then it appears magically on the post. The page doesn't refresh and I'm not taken to the top of the page!

You know what? Let's do that too. Let's make our comments feel magic.

## Magic Commenting, aka AJAX

You've heard of AJAX right? It's a bleach cleaner that does wonders for your bathroom.

The other AJAX that I care about for the sake of our comments is described thusly on w3schools.com:

> *AJAX is the art of exchanging data with a server, and updating parts of a web page – without reloading the whole page.*

This sounds perfect! Let's make this happen.

### In-built Rails AJAX

DHH is kind enough to include some magical AJAX goodies in Rails for free with every new project. This lives in the form of the included jQuery functionality within Rails.

The way it works is like so:

- We let Rails know we want to send our form data via AJAX by adding the `remote: true` argument to our form_for helper method. In practice, this is what it'll look like in our comment form (don't implement just now, this is just an example):

```
= form_for([post, post.comments.build], remote: true) do |f|
```

With that taken care of, we now need to make sure that our appropriate action (the create action in this case) will respond to this form via javascript and not just render a whole page of html for us again.

You might've noticed the `respond_to` method in your controllers in the past when you've used scaffolds to generate your controller. In those scenarios, respond_to will return JSON or html, depending on the request type. In our case, we want our 'create' action to return javascript. This is what it'll look like:

```
respond_to do |format|
  format.html { redirect_to root_path }
  format.js
end
```

Lastly, we're going to need to inform our application on how we now want to modify the DOM. We're not refreshing the whole page after all, so you're going to need to write some jQuery that adjusts the page appropriately. I'll show you my jQuery implementation for the comment functionality in the below section.

Let's add the AJAX functionality for only our create action for the moment. If you want to try this yourself first, now's your chance!

Remember to commit your git changes first in case you need to revert during your adventure (no offense).


---

Let's begin!

But first...

I want to quickly move our comments to their own partial view for the sake of this exercise. AJAX calls within rails work hand in hand with partials.

Why?

Well, we want to re-render the comments of an individual post once our *new* comment has been submitted. This way we can see it immediately, all without reloading the whole page!

The comment partial is shown below. Call it `_comment.html.haml` and create it in the `views/comments` folder.

```
#comment
  .user-name
    = comment.user.user_name
  .comment-content
    = comment.content
  - if comment.user == current_user
    = link_to post_comment_path(post, comment), method: :delete,
data: { confirm: "Are you sure?" } do
      %span(class="glyphicon glyphicon-remove delete-comment")
```

Pretty basic right? We've just moved the comment part of our form into a separate file. Now, we'll have to adjust our `_post.html.haml` partial so it renders the comments appropriately. Adjust your post partial so it looks like the code seen here:

```
 .posts-wrapper
  .post
    .post-head
      .thumb-img
      .user-name
        = post.user.user_name
    .image.center-block
      = link_to (image_tag post.image.url(:medium), class:'img-
responsive'), post_path(post)
    .post-bottom
      .caption
        .caption-content
          .user-name
            = post.user.user_name
          = post.caption
        .comments{id: "comments_#{post.id}"}
          - if post.comments
            = render post.comments, post: post
    .comment-like-form.row
      .like-button.col-sm-1
        %span(class="glyphicon glyphicon-heart-empty")
      .comment-form.col-sm-11
        = form_for [post, post.comments.build] do |f|
          = f.text_field :content, placeholder: 'Add a
comment...', class: "comment_content", id: "comment_content_#
{post.id}"
```

There you go! But wait, `= render post.comments` is new! We're not even specifically defining which partial view to render!

The above is Rails shorthand that we can use due to the fact that our partial's name is 'comment'. Rails assumes that this is the partial we want to render and we're left with lovely, neat code!

**Back to the AJAX features you fiend!**

**Adding remote: true to our form.**

Our comment form is a part of our `_post.html.haml` partial, so open that file and add 'remote: true' to the form_for method.

Oh and while you're in there, let's added some extra code that will give us some unique div id numbers, based on the id of the object. (You might've noticed this on the code above).

```
    .post-bottom
      .caption
        .caption-content
          .user-name
            = post.user.user_name
          = post.caption
        .comments{id: "comments_#{post.id}"}
          - if post.comments
            = render post.comments, post: post
    .comment-like-form.row
      .like-button.col-sm-1
        %span(class="glyphicon glyphicon-heart-empty")
      .comment-form.col-sm-11
        = form_for([post, post.comments.build], remote: true) do
|f|
          = f.text_field :content, placeholder: 'Add a
comment...', class: "comment_content", id: "comment_content_#
{post.id}"
```

**Let's allow our controller to return javascript on a call, not only html!**

Adjust the create action in your comments controller to look like this:

```
  def create
    @comment = @post.comments.build(comment_params)
    @comment.user_id = current_user.id

    if @comment.save
      respond_to do |format|
        format.html { redirect_to root_path }
        format.js
      end
    else
      flash[:alert] = "Check the comment form, something went
wrong."
      render root_path
    end
  end
```

This means that our create action can now respond with both html and javascript. The fact that we added 'remote:true' to our form means it'll naturally be looking for a javascript response!

Also note, I deleted the flash message that we used to get upon creation of a comment. The fact that comments now appear in the list should be confirmation enough for our users (it is for the *actual* Instagram page).

**Create the javascript response.**

jQuery to the rescue! Create a new file in your `views/comments` folder. Call it `create.js.erb`. In that file, add the following javascript / ruby combination:

```
$('#comments_<%= @post.id %>').append("<%=j render
'comments/comment', post: @post, comment: @comment %>");
$('#comment_content_<%= @post.id %>').val('')
```

What exactly are we doing here? We're selecting the `comments_(specific id)` div and appending the comment partial to begin with. After that, we select the `comment_content_(specific id)` div and change it's value to an empty string.

What's that doing in actual english?

- Adding our newly added comment to the list of comments.
- Clearing out the form that we just typed in.

Cool bananas batman!

Try it out now. You should be able to add comments with ease and they'll **MAGICALLY** appear in the comment list of a post!

Try to do the same now for the deletion of comments. Remember, the process itself isn't too tricky.

1. Add 'remote: true' to the delete form_for.
2. Make sure the destroy action in the controller will return javascript when requested.
3. Last but not least, manipulate the DOM via jQuery. You'll probably want to render the comments again, just like we did for the create action!


---

Here's how I implemented AJAX into the deletion of comments.

**Add remote:true to the form**

This time we're adding it to the destroy form, not the create form (you can find the form for the destroy action in the `_comment.html.haml` partial view). Check out my version below:

```
#comment
  .user-name
    = comment.user.user_name
  .comment-content
    = comment.content
  - if comment.user == current_user
    = link_to post_comment_path(post, comment), method: :delete,
data: { confirm: "Are you sure?" }, remote: true do
      %span(class="glyphicon glyphicon-remove delete-comment")
```

Now its time to...

**Add the javascript response to the controller action**

Just as before, we can now make sure Rails responds with not only html but also javascript. Add the `responds_to` method to your destroy action within the comments controller.

```
def destroy
  @comment = @post.comments.find(params[:id])

  if @comment.user_id == current_user.id
    @comment.delete
    respond_to do |format|
      format.html { redirect_to root_path }
      format.js
    end
  end
end
```

Please note I've added some extra security in the delete action to ensure only the owner of a comment can delete it.

And last but not least..

**Finalise with some lovely jQuery**

Our jQuery should make sense after seeing the comment create feature in action. We're simply appending our updated comments list to the comment div!

Create your new `destroy.js.erb` file within your `views/comments` folder (the same location as your `create.js.erb` file). It should look something like:

```
$('#comments_<%= @post.id %>').html("<%= j render
@post.comments, post: @post, comment: @comment %>");
```

And guess what? That's it for the delete functionality as well!

Try it now! Beautiful, right? You click the 'x', you get the prompt, you accept and....

Comment deleted.

## Tidying Up

Let's fix up a few loose ends in our application. These are a few small little items that are certainly worth looking at now that the core functionality is built.

1. **Add length limitations to caption column in Post.rb**

We probably should've added this is Part 1 of the guide but let's not hold a grudge. I think you'll have the ability to do this now actually. Go ahead and add a minimum and maximum length validation to the Posts model for the content column. I think minimum should be 3 characters and maximum... Let's say 300.

No hints here, validations should be easy as pie for a developer of your calibre. If your brain fails you, google will quickly show you how.

1. **Add a post 'time ago' helper as per the desktop Instagram app**

You know the one! It's the bit on the top right hand corner that says that your last selfie was posted 44m ago! Rails makes this *super* easy to implement.

As per usual, try it yourself, you're awesome and can do it. If you're feeling lazy due to the time of day though, pilfer my code below. I won't be angry...

**just disappointed.**

In your `_post.html.haml` partial view, add the .time-ago div and the time_ago_in_words helper with the post.created_at argument.

```
.user-name
  = post.user.user_name
.time-ago
  = time_ago_in_words post.created_at
.image.center-block
  = link_to (image_tag post.image.url(:medium), class:'img-
responsive'), post_path(post
```

The surrounding code hasn't changed, it's just there for a reference.

Refresh your index or show view and you should be presented with a handsome timer!

## Concluding Chapter Two

Think about what you've achieved in this chapter:

- You've built great user functionality, associated users and posts along with control over who can edit or delete specific posts.
- Your users can now comment on each others posts and delete them if required.
- Not only that, we implemented some great AJAX functionality so the whole commenting process is incredibly low-friction and beautiful to use!

In the next chapter, let's add some likes to our posts.

# Chapter 3 - Fabulous Forms & Pleasant Pagination

In this chapter, we're going to go side-step a little from adding more features. Instead, we're going to improve on what we already have.

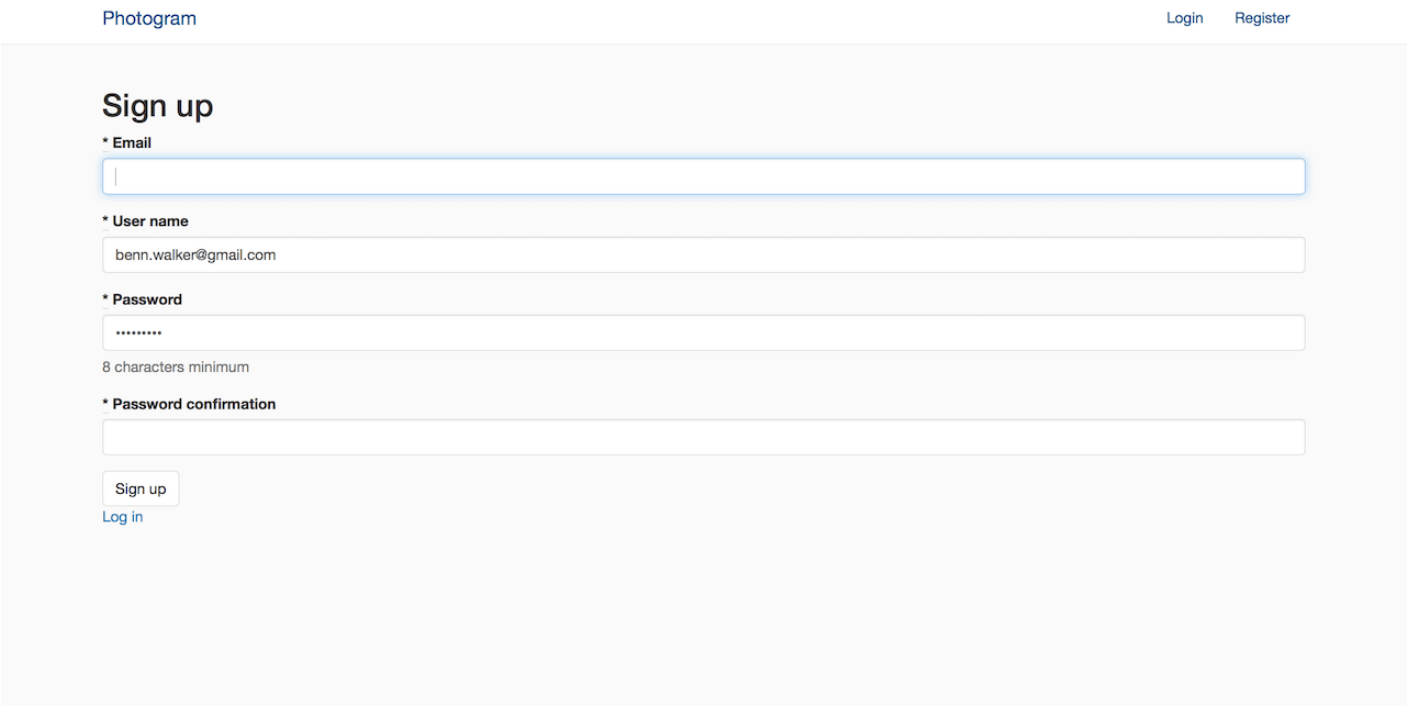
Here's what we're going to implement this time:

- We're going to beautify ALL of our forms. This includes: registration, sign up, post creation, post editing. Why? Because we can do better dammit.
- We're going to paginate our main news feed so our server doesn't explode with new visitor to the feed.
- We're going to limit how many comments are shown on an individual post, before we expand them with a 'view all comments' link. This is similar to how Instagram handles it and I'm in too deep with plagarism to change now.

Now that you're primed for an incredible learning experience, let's get into it. Brace yourself.

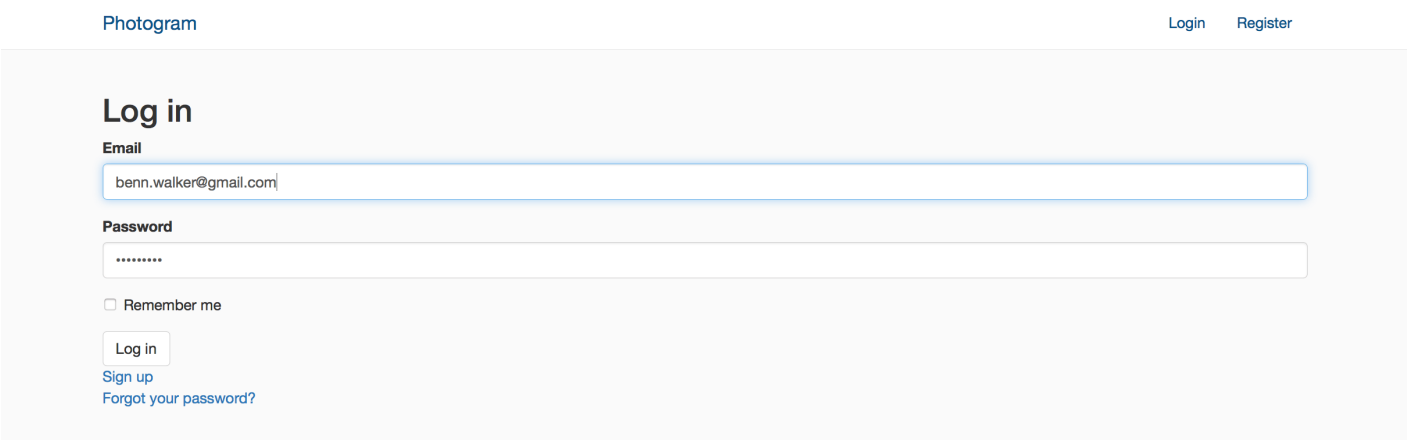
## Our Forms are awful

Look at this, just look at it:

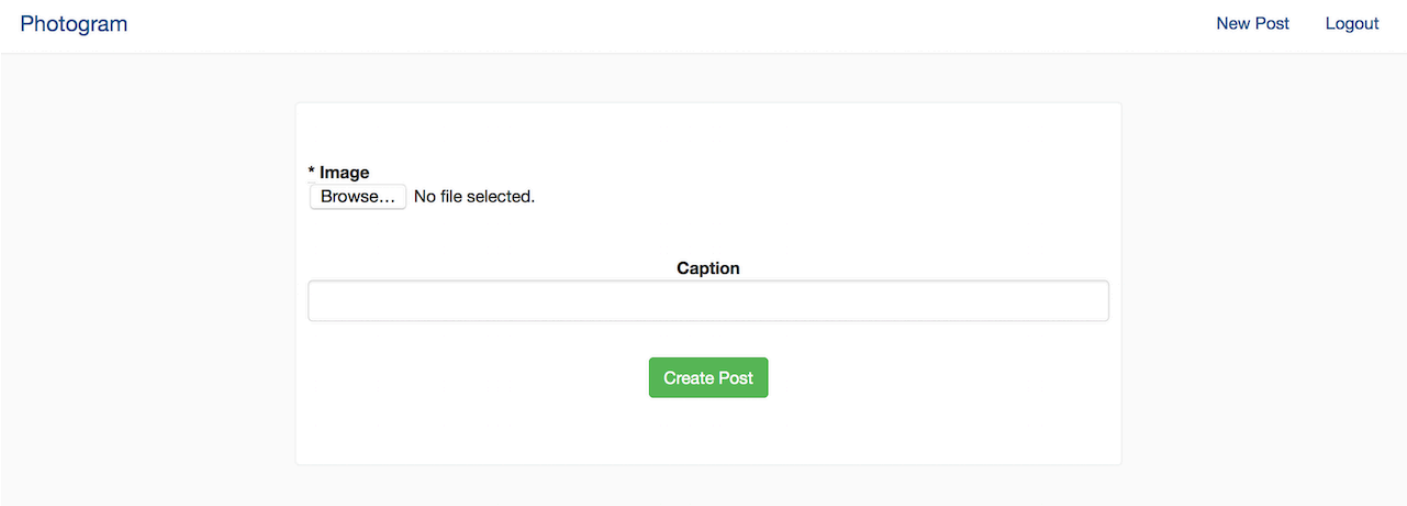



This is our so-called signup page. Who would want to sign up to this? I have no idea. Probably some sort of monster.

Now, look at this:

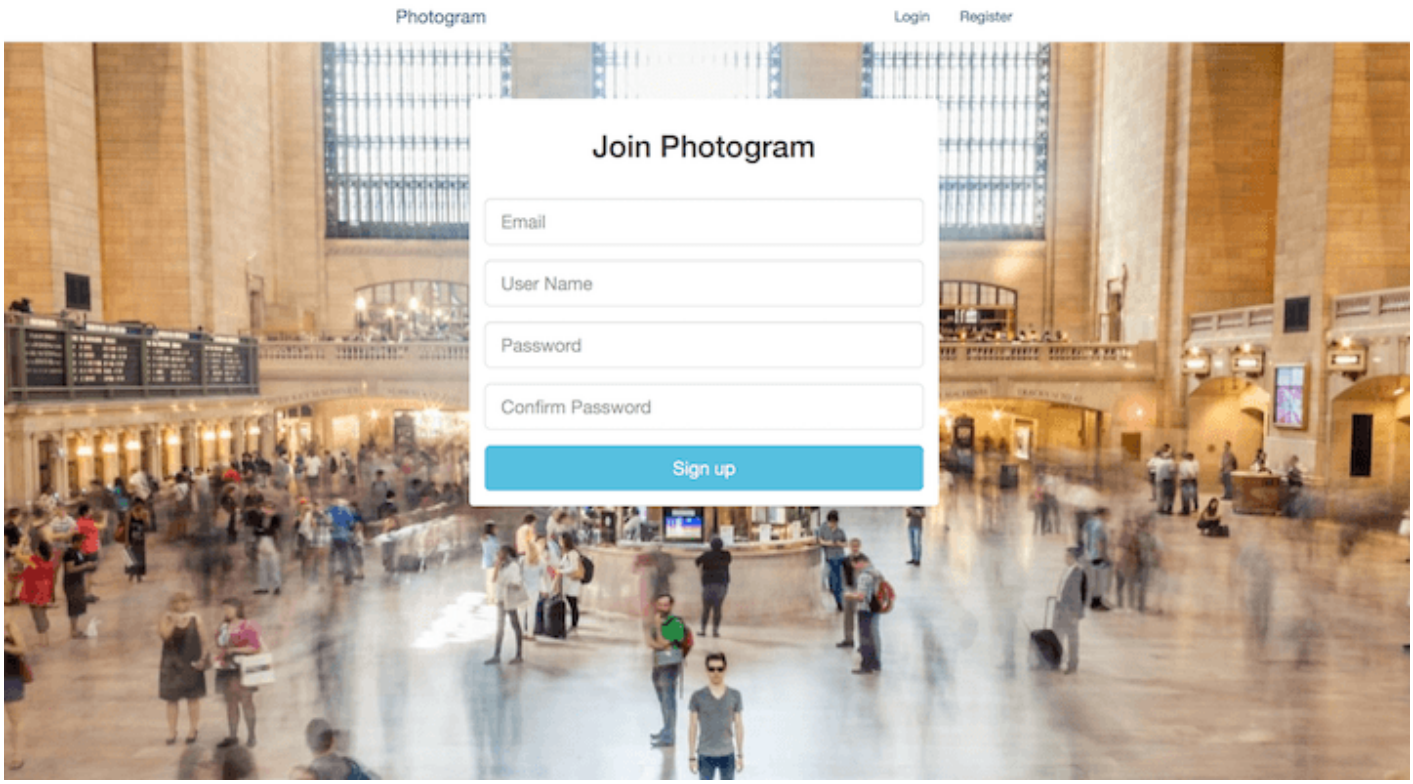



Disgusting, right? If your stomach can handle it, check this out:

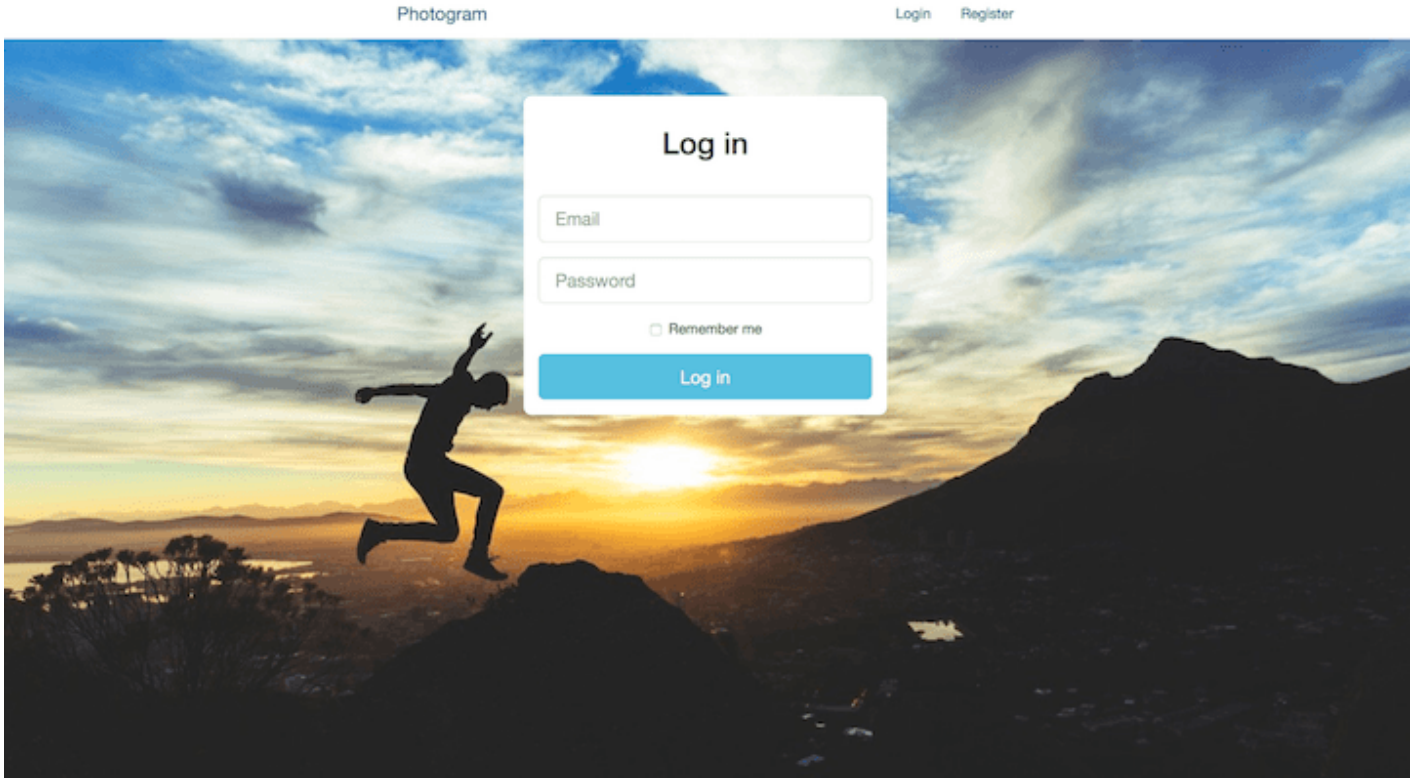



What the hell is that image upload button doing over there? Why is it so... unattractive?

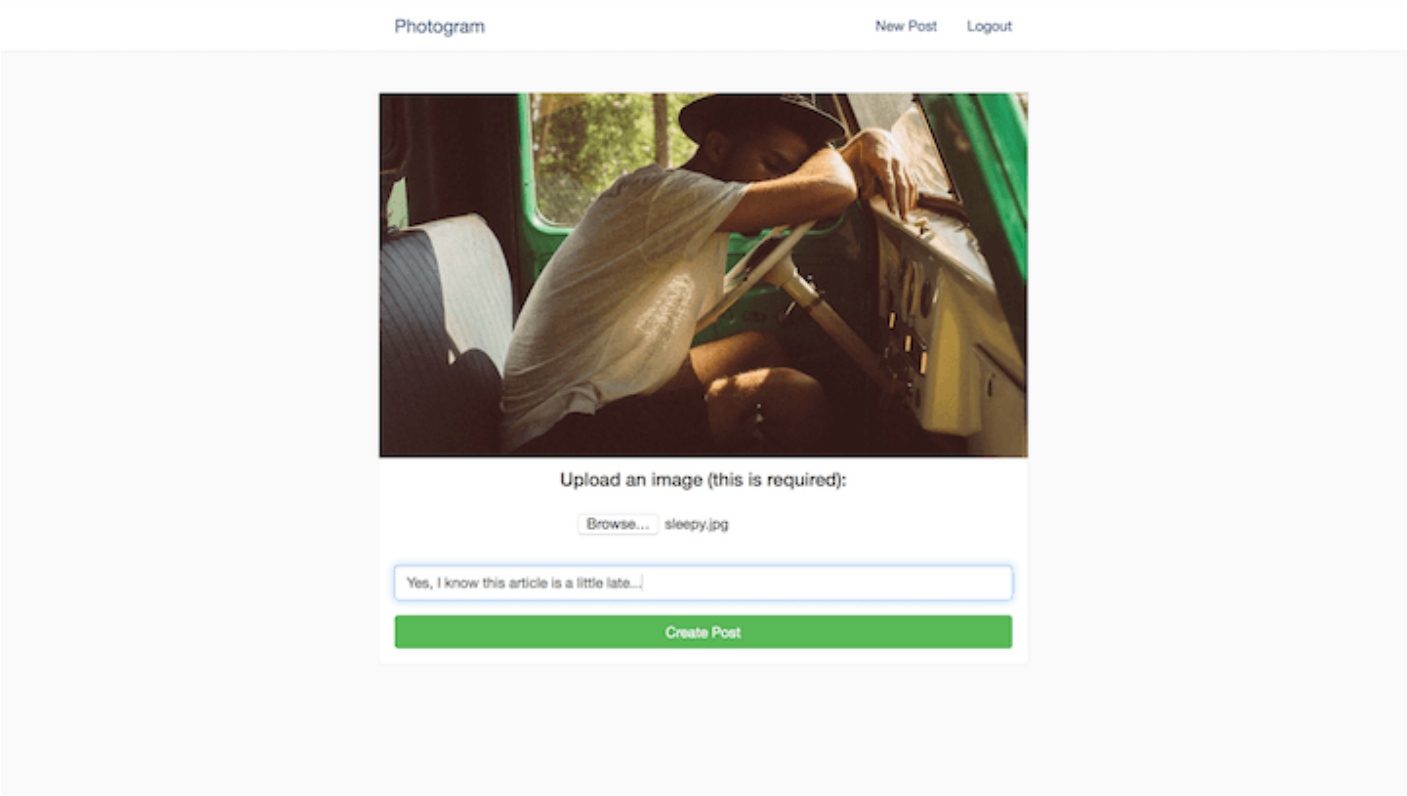
Nobody knows, I'm afraid, but fear not. I'm here to save your eyeballs from further monstrosities. This is what we're going to create:




Ooh!  I'd want to join that!




Oh my!  I'd like to login to that site.




*Noise of fainting*

How great do those forms look now? It's ok, you don't have to tell me, they look fantastic. I know it and you know it.

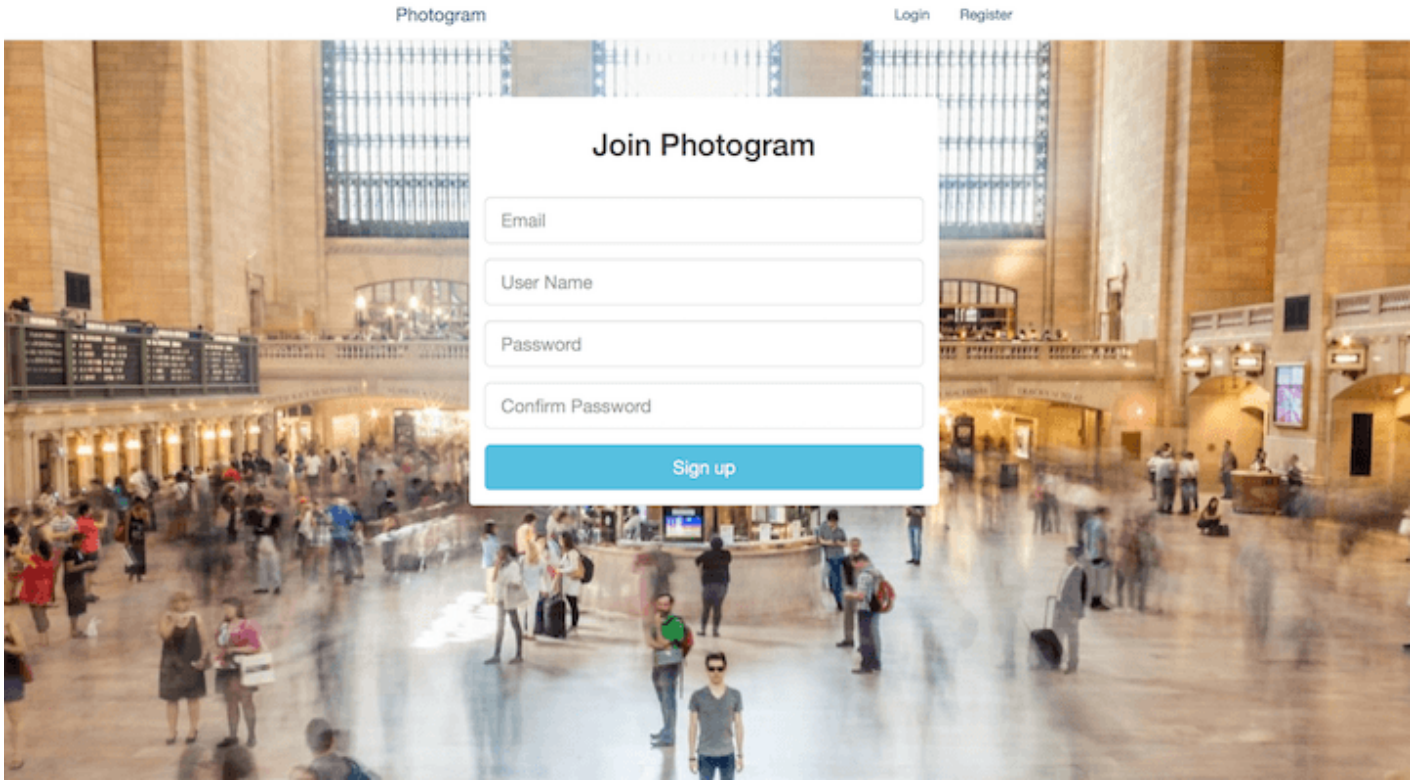
## Sign Me Up to This Glorious Looking Site

Let's first attack our registration form, it is the form our users will see first after all.  To begin, let's make some simple styling choices and implement them with our bootstrap divs and some simple CSS.

This is what we begin with:

```
%h2 Sign up
= simple_form_for(resource, as: resource_name, url:
registration_path(resource_name)) do |f|
  = f.error_notification
  .form-inputs
    = f.input :email, required: true, autofocus: true
    = f.input :user_name, required: true
    = f.input :password, required: true, hint: ("#
{@minimum_password_length} characters minimum" if
@minimum_password_length)
    = f.input :password_confirmation, required: true
  .form-actions
    = f.button :submit, "Sign up"
= render "devise/shared/links"
```

Just a simple HAML'd version of the standard Devise register view. The below image is what we want this form to look like:




Using the boostrap documentation for forms and the screenshot above, try to implement the new styling yourself. If you need some extra tips, think of it like this. You're going to need to add the background image somehow, and you're going to need to create the area for the form. I may have used Bootstrap's inbuilt panels for basic layout too...

Oh yeah, don't forget that the view for our registration form can be found at `app/views/devise/registrations/new.html.haml`.


---

Did you have a win? Of course you did! I'll just show you my new form so we can compare notes.

Here's my new view:

```
.registration-bg
  .container
    .row
      .col-md-4.col-md-offset-4
        .log-in.panel
          .panel-heading
            %h2 Create Your Account
          = simple_form_for(resource, as: resource_name, url:
registration_path(resource_name)) do |f|
            = f.error_notification
            .panel-body
              = f.input :email, required: true, autofocus: true,
label: false, placeholder: 'Email', input_html: { class: 'input-
lg' }
              = f.input :user_name, required: true, label:
false, placeholder: 'User Name', input_html: { class: 'input-lg'
}
              = f.input :password, required: true, label: false,
placeholder: 'Password', input_html: { class: 'input-lg' }
              = f.input :password_confirmation, required: true,
label: false, placeholder: 'Confirm Password', input_html: {
class: 'input-lg' }
              = f.button :submit, "Sign up", class: 'btn-lg btn-
info btn-block'
```

And my new scss in my `application.css.scss` file.

```
html {
  height: 100%;
}

.registration-bg {
  padding-top: 4em;
  height: 100%;
  background-image: image-url('regbackg.jpg');
  -moz-background-size: cover;
  -webkit-background-size: cover;
  -o-background-size: cover;
  background-size: cover;
}

.log-in{
  margin-left: auto;
  margin-right: auto;
  text-align:center;
  border: none;
}

.panel {
  border-radius: 8px;
}

.panel-heading{
  h1 {
    text-align: center;
  }
}
```

Want a beautiful background image for yourself? Checkout unsplash and take your pick of pics.

Do you notice anything strange with your application now that we've implemented the above styling? You're getting some ugly margins pushing away the background image, right? We've got to counter a couple of Bootstrap defaults here. In your `application.scss` file, add the following lines:

```
.navbar {
  margin-bottom: 0px;
}

.alert {
  margin-bottom: 0px;
}
```

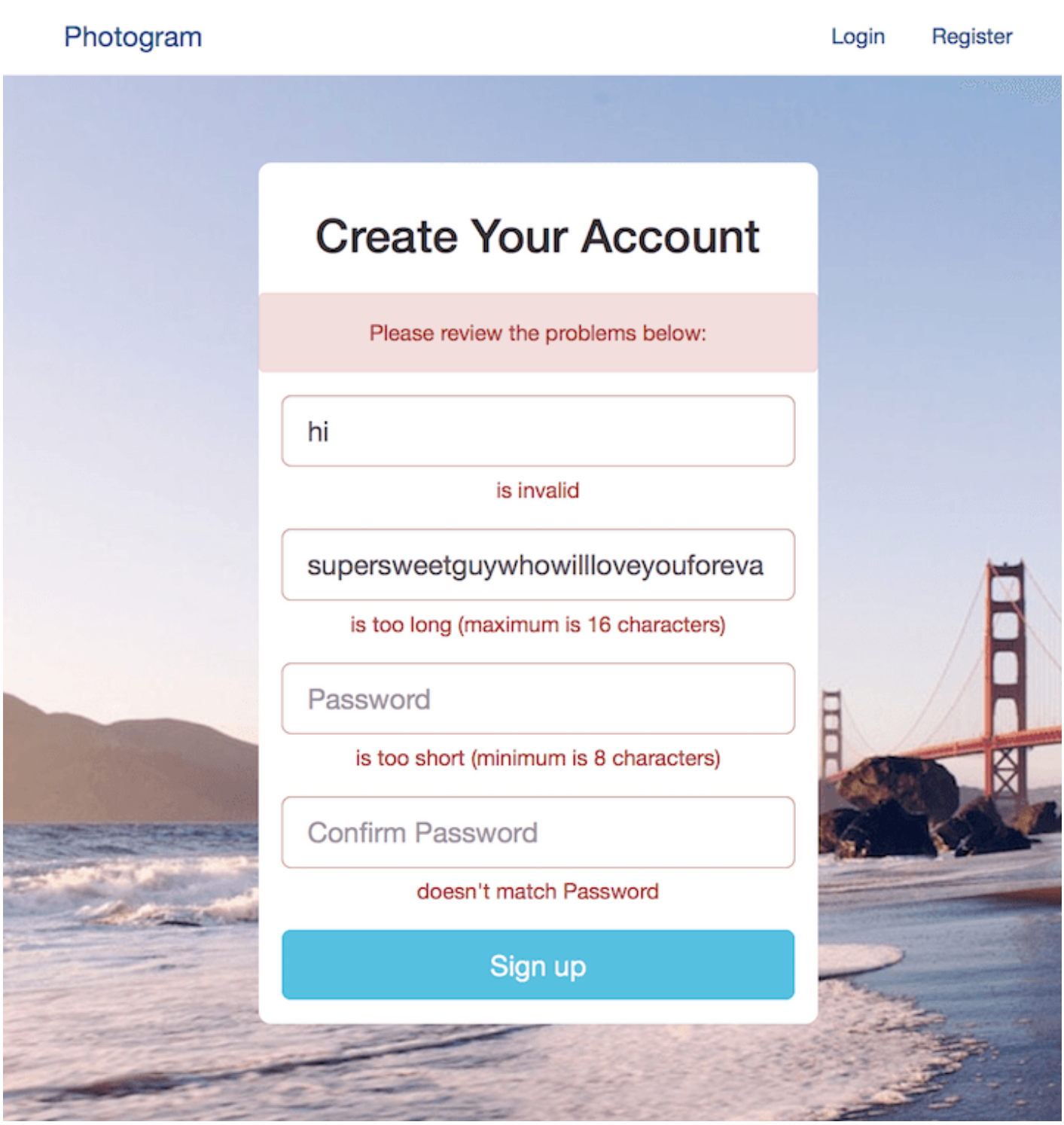
Nice and simple.  If you notice anything else that conflicts, remember to play around with the dev tools on your browser.  It's the easiest way to see what fiendish css rule is foiling your beautiful layout.

---

Also take notice of the included line of code in our form (it was in the old version too), the line that says:

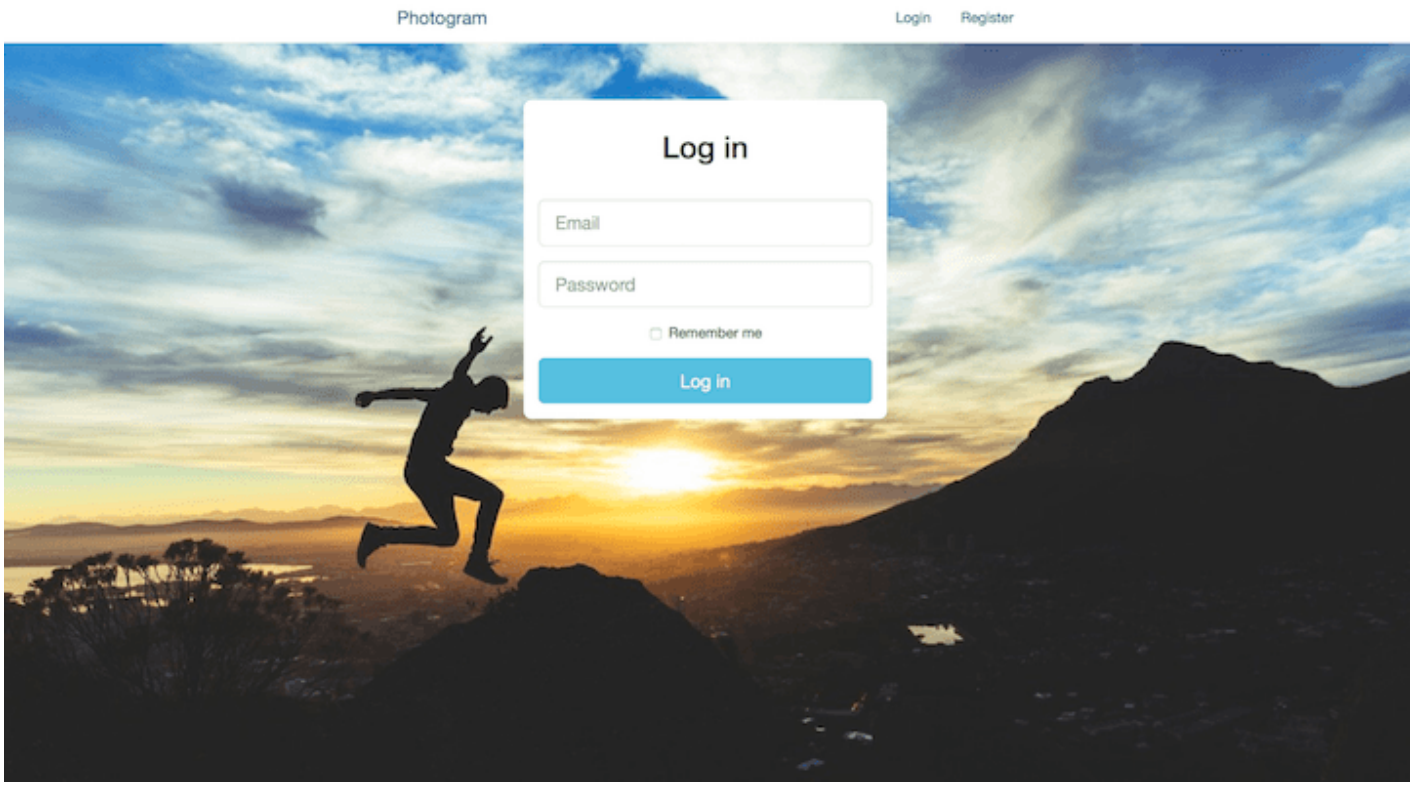
```
= f.error_notification
```

Open the registration form now in your browser and submit some crappy information that will fail the validations.  Simple form handles the errors brilliantly.  Not only does it tell us that we've messed up our form somehow, it guides us through exactly what went wrong.  Good stuff.




## Logging in should feel like a win

You've already built one beautiful form, go ahead and try to build this one now. Use the exact same principles and heck, the same divs and css. let's not be wasteful here. Here's the new and improved login form again for reference:




Looks similar to the registration form, right? Well it is. Go forth and create it. You can find it at `app/views/devise/sessions/new.html.haml`.


---

All done? Oh yours is better than mine? Oh god you're right...

Well here's mine anyway:

The new haml view over at ```app/views/devise/sessions/new.html.haml````:

```
.login-bg
  .container
    .row
      .col-md-4.col-md-offset-4
        .log-in.panel
          .panel-heading
            %h2 Log in
          = simple_form_for(resource, as: resource_name, url:
session_path(resource_name)) do |f|
            = f.error_notification
            .panel-body
              = f.input :email, required: true, autofocus: true,
label: false, placeholder: 'Email', input_html: { class: 'input-
lg' }
              = f.input :password, required: true, label: false,
placeholder: 'Password', input_html: { class: 'input-lg' }
              = f.input :remember_me, as: :boolean if
devise_mapping.rememberable?
              = f.button :submit, "Log in", class: 'btn-lg btn-
info btn-block'
```

And, the associated scss:

```
.login-bg {
  padding-top: 4em;
  height: 100%;
  background-image: image-url('loginbg.jpg');
  -moz-background-size: cover;
  -webkit-background-size: cover;
  -o-background-size: cover;
  background-size: cover;
}
```

The rest of the styling is shared with the scss above ;)

---

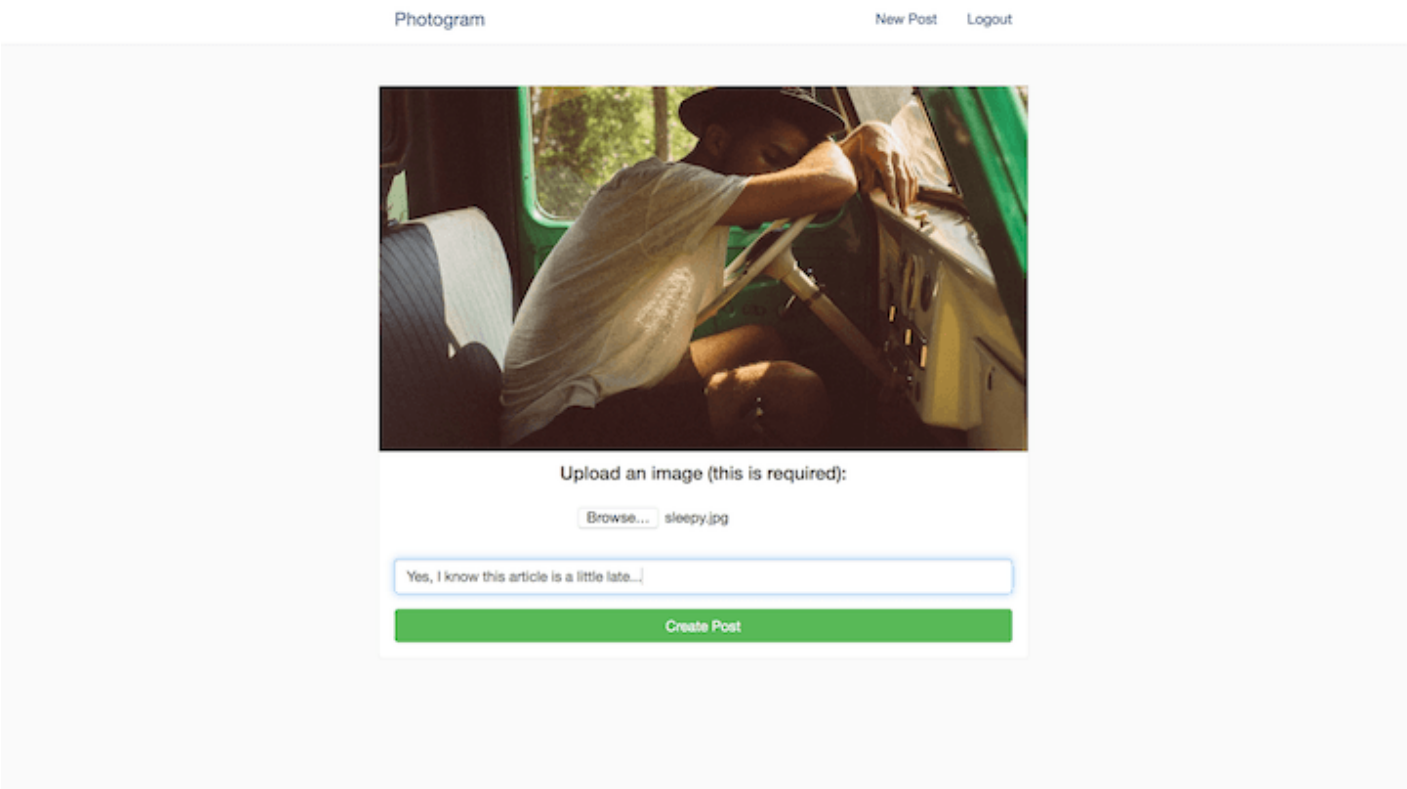
Look good? You bet your ass it does. Now, onto...

## Pretty Post Posting & Previews

Our old form looks a bit yuck and we don't get to preview our images either! It's just not right. let's shake things up a bit so our form actually looks OK, and we can have a quick glimpse at our image before we post, just in case we were to

accidentally upoad a picture of our glutes again...

This is what the new form looks like again, for reference:




The image part of the form is automagically changed when the attached image is changed in the form. We have a default image in it's place before we've chosen something to upload. The rest of the form is pretty similar to what it once was, we've just spiced things up a little.

Give it a go now, try to make it look better than mine! Feel free to give fancy uploaders like Dropzone or jQuery Upload a go too for the sake of a challenge (maybe just create a git branch first.)

Good luck you crazy thing.

Alright, we've got a few bits to this puzzle but it's much simpler than using Dropzone / jQuery. First, here's our new view file, found at `app/views/posts/_form.html.haml`:

```
.posts-wrapper
  .post
    .post-body
      .image-wrap
        = image_tag 'placeholder.jpg', id: 'image-preview',
class: 'img-responsive'
        = simple_form_for @post, html: { multipart: true } do
|f|
          .row
            .col-md-12.text-center
              = f.error_notification
          .container-fluid
            .form-group.text-center
              %h4 Upload an image (this is required):
              = f.input :image, label: false, input_html: {
onChange: 'loadFile(event)' }
            .form-group.text-center
              = f.input :caption, label: false, placeholder:
'Add your caption'
            .form-group.text-center
              = f.button :submit, class: 'btn-success btn-block'
```

Alright, so let's discuss the three differences from the old form, to the new.

- We've got a simple image_tag image as a placeholder with a specific id.
- We've got a suspicious looking html addition to our image input, called `onChange: 'loadFile(event)'`
- We've added some extra divs for styling purposes.

Let's look at the javascript that has caused us to make these changes. Here's some fresh javascript that now lives within my `app/assets/javascripts/posts.js` file (I had to rename it from .coffee):

```
var loadFile = function(event) {
  var output = document.getElementById('image-preview');
  output.src = URL.createObjectURL(event.target.files[0]);
};
```

That's all! How good is that? On change, we grab the 'image-preview' element, and change it's src attribute to the file path of our input (if it is in fact, an image file).

I really love this, because of it's sheer simplicity. It achieves what we want, an image preview and therefore better form, and does so with three (and a bit) lines of javascript.

Here's some extra styling you can add to your `application.scss` file too:

```
#image-preview {
  margin: 0px auto;
}

#post_image {
  padding: 1em;
  margin: 0px auto;
}
```

---

So let's say you ran with my idea regarding the new posts form, do me a favour now and edit one of your existing posts. Change the image and see what happens...

Yep, we're duplicating the image preview in the edit view and it's not being friendly with our new javascript preview feature. Let's fix that now.

## Editing editing

So we're duplicating the display of our images... Wouldn't it be great if the default image was in fact the image for the post we're looking at, when available?

Do your best to fix this issue now. I'll give you a hint, I used a helper to move the logic out of my view. The helper method uses a simple if / else statement to determine whether the post at hand has an existing image or not. If it does, display that image, otherwise, display the placeholder image.

It's your turn to take the reigns.


---

Ok, here's how I implemented the fix. First, I deleted the image_tag from the `edit.html.haml` view that was handling our original preview functionality. The `edit.html.haml` file now looks like:

```
= render 'form'
.text-center.edit-links
  = link_to "Delete Post", post_path(@post), method: :delete,
data: { confirm: "Are you sure you want to delete this post?" }
  |
  = link_to "cancel", posts_path
```

Moving onto the `_form.html.haml` partial, I removed the image_tag and instead am calling my new helper:

```
.posts-wrapper
  .post
    .post-body
      .image-wrap
        = form_image_select(@post)
        = simple_form_for @post, html: { multipart: true } do
|f|
          .row
            .col-md-12.text-center
              = f.error_notification
          .container-fluid
            .form-group.text-center
              %h4 Upload an image:
              = f.input :image, label: false, input_html: {
onChange: 'loadFile(event)' }
            .form-group.text-center
              = f.input :caption, label: false, placeholder:
'Add your caption'
            .form-group.text-center
              = f.button :submit, class: 'btn-success btn-block'
```

The helper is simply called `form_image_select` and I've created it within the `app/helpers/application_helper.rb` file, below our bootstrap flash helper. My new helper looks like:

```
def form_image_select(post)
  return image_tag post.image.url(:medium),
                   id: 'image-preview',
                   class: 'img-responsive' if post.image.exists?
  image_tag 'placeholder.jpg', id: 'image-preview', class: 'img-
responsive'
end
```

Hopefully that makes sense. In our form partial, we call the `form_image_select` helper which simply determines if the `@post` we're referring to has an image or not. If it does, we display that image, along with the associated id and class. If not, we display `placeholder.jpg` which is safely stored in `app/assets/images/`. Feel free to make your own for the sake of the exercise. Resize to 640 px wide for best results.

---

**Success!**

Congratulations. You now have great looking forms for the marjority of your application! Combine that with a image preview for your new and edited posts, you're truly winning. Not only will this save you the embarrassment of revealing your junk to your siblings, it also provdes for a great user experience.

There's something else I'd like to attack in this article, and it also has to do with user experience. User experience and preventing our server from imploding...

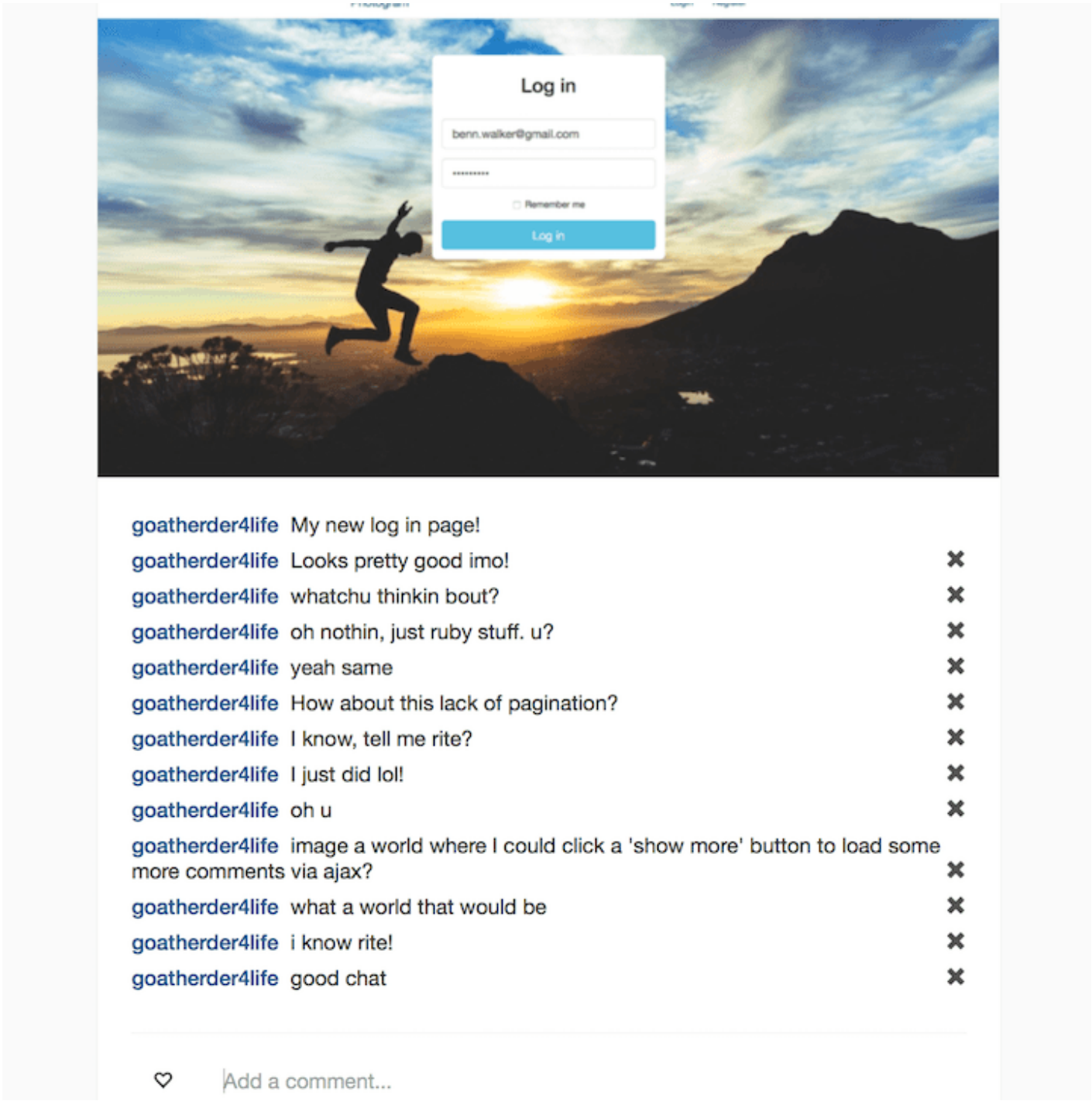
## Paginate all of the things

Yes. We're going to paginate everything you can dream of. Luckily for me, you have a poor imagination, which means we're going to paginate our posts on the index and our comments for each post.

Why paginate? Well consider our application for the moment. Every.single.post and every.single.comment is being called on our index. In our posts controller, our `@posts` instance variable is set to `Post.all` after all.

**It's giving us litererally everything.**

And that's ok... until it isn't. It's ok on my development server because I have half a dozen posts. It's not ok in production when we have many, many more posts. The same is true for our comments. Looking at our `_post.html.haml` partial, we can see that we're simply iterating through *every single* comment that exists on a post and displaying them all.




It's not ok, so let's fix it now.

## Pleasantly Paginating Posts

We'll paginate comments soon, I assure you. First, let's paginate our posts.

Instagram handles this on the desktop version of their site with a big round 'LOAD MORE' link (when it doesn't use 'infinite scrolling'), so let's just do that too. Instagram also loads 12 posts at a time which seems pretty sensible. Want to go completely mad?

Load 13 at a time.

You're on your own though, I dare not push my sanity that far. let's incorporate this feature now by installing a lovely gem called, kaminari.

Before you get too excited, read through the docs first. I don't expect you to understand everything, I just want you to have a good high level overview of what it does.

Once informed, add the gem to your `Gemfile` as:

```
gem 'kaminari', '~> 0.16.3'
```

and run `bundle install` in your terminal. Once that's done, run the kaminari config generator in your terminal with:

```
rails g kaminari:config
```

The only option we're going to change here is setting the `default_per_page` option to 12. We're now done setting up! Oh, for what it's worth, if you want to lower the `default_per_page` option to make it easier to see pagination in action within your running server, go for it. You can always go back and change it easily enough. I personally dropped it to 3 while writing this very article!

Open your `posts_controller.rb` file now and adjust your `@posts` instance variable within your index action. We want to paginate it as per our default setting:

```
def index
  @posts = Post.all.order('created_at DESC').page params[:page]
end
```

And that's it! Restart your server within your terminal and then re-visit your index to see the changes. You'll only see the number of posts specified in your config. Want to see the rest? Well we can manually move to a specific 'page' by adding the page param to your path. Try `http://localhost:3000/?page=2` and you should be able to see the next round of posts! If you have enough posts, you'll be able to go onto page 3 and 4 and so on.

This is all well and good but we don't want our users to manually input parameters into the address bar because we're not complete psychos. It's

incredibly simple to add page number links to your index view. In fact, it's only just hit me how I've been completely holding your hand through this whole process! Do me a favour now, and refer back to the kaminari docs and look for the sections that refers to **views**. It'll show you how to add the links.


---

So you clicked that link and did it yourself right? Of course, of course, I'm sorry I questioned your honesty.

Jump into you `index.html.haml` view and adjust the file so it looks like this:

```
-@posts.each do |post|
  = render 'post', post: post

= paginate @posts
```

Could it be any easier? Probably not. How good is kaminari?

---

If you jump back into your index again, you should be able to click around on the page numbers and be taken to each. Whilst it's fantastic that we've implemented pagination and our server is no longer imploding, the user experience still isn't what I expect from a modern day web application. I feel like it should be... *nicer*.

## Where art thou AJAX?

We've rumbled in the AJAX jungle back in Part 2 of this guide, when we created and deleted new comments for posts via AJAX calls and used some jQuery magic to adjust the DOM appropriately. Let's do something similar now so that we don't need a full page refresh when we request more posts.

Give it a go yourself first though. Just a simple google search for 'kaminari ajax' will give you plenty of guides and stackoverflow questions to look at. We want our

functionality to work like so:

- Click on a LOAD MORE button at the bottom of the page and then *append* the next 'page' of posts below an id or class used to contain the existing posts.

Go forth and create! If you want bonus points, maybe consider an 'infinite scroll' feature where the extra content will be loaded simply by scrolling down. Instagram seems to do it both ways, and there are guides for both, so let your heart guide you.


---

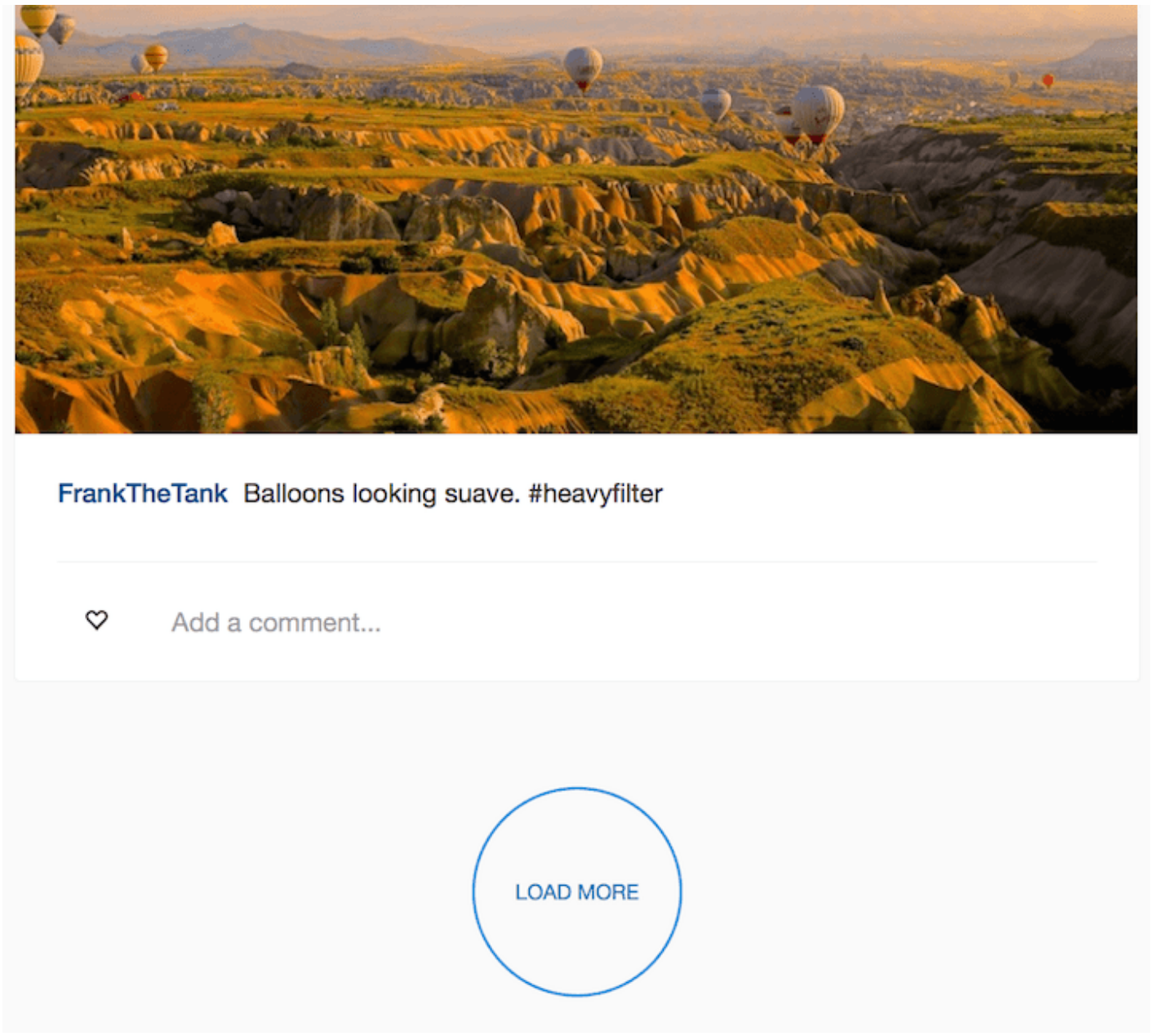
Here's how I implemented this feature, manual clicking style. First, I slightly adjusted the index view and added a new partial (for the sake of AJAX). Here's my updated `index.html.haml`:

```
#posts
  = render 'posts'

#paginator.text-center
  = link_to_next_page @posts, 'LOAD MORE', remote: true, id:
'load_more'
```

Differnces? We'll I've got a new div with the 'posts' id. Also, instead of iterating through each post at this point, I'm rendering a new partial called 'posts'. Lastly, I've added `remote: true` and an id to the link_to_next_page paginator, as well as a div above that with the id 'paginator'. The added divs are all for the sake of either jQuery or styling our new button.

Speaking of styling, below is the the additional lines in my `application.scss` file. Just add them to the bottom of your existing code.

```
#paginator {
  color: #4090DB;
  height: 120px;
  width: 120px;
  margin: 60px auto 40px;
  position: relative;
  display: table;
  border: 2px solid #4090DB;
  border-radius: 50%;
  :hover {
    border: 1px solid #2D6DA8;
    border-radius: 50%;
  }
}

#load_more {
  display: table-cell;
  font-size: 12px;
  padding: 0px 9px;
  vertical-align: middle;
  text-decoration: none;
  a :hover {
    color: #2D6DA8;
  }
}
```




Our index is now rendering a `views/posts/_posts.html.haml` partial, which should look like this:

```
-@posts.each do |post|
  = render 'post', post: post
```

It's simply the block that iterates through each post in the `@posts` collection, rendering the `_post.html.haml` partial for each itme. Why do this? Well, it's for the sake of our AJAX & jQuery magic, something that I'll run through in a moment.

Finally, I'd like to show you a new file called `index.js.erb`.

This file is very similar in nature to the `create.js.erb` and `destroy.js.erb` files

that we created for the sake of our comments in the last article. It contains jQuery for the sake of manipulating the DOM once the 'LOAD MORE' link is clicked and we've been given some data to use in our view.

The following will live in `app/views/posts/index.js.erb`.

```
$('#posts').append("<%= escape_javascript(render 'posts')%>");
$('#paginator').html("<%=
escape_javascript(link_to_next_page(@posts, 'LOAD MORE', remote:
true, id: 'load_more'))%>");
if (!$('#load_more').length) { $('#paginator').remove(); }
```

What's happening here? In the first line, we append the `#posts` div with the new `_posts.html.haml` partial, ie. we add it on the bottom.

In the second line, we select the `#paginator` div and set it's contents to our `link_to_next_page` kaminari helper method. This will 'refresh' our link, meaning it will take us continually onto the next page as we keep clicking, rather than constantly giving us a link for page 2 and will also reset it's position.

In the third line, we remove the `#paginator` div if the `#load_more` element (our link) doesn't exist. This is purely to remediate some styling quirks that I was seeing once I'd run out of posts to paginate. Feel free to remove that line and experiment for yourself.

I'll mention a little quirk I found in my application that you may not experience. At one point when implementing the above funcitonality, Rails was complaining that there was no `user_name` method for Nil when referring to `comment.user.user_name` in the `_comments.html.haml` partial. If you run into this error, add a conditional above that line, something like below:

```
.user-name
  - if comment.user
    = comment.user.user_name
.comment-content
```

This no longer occurred once I built out some more features I'll talk about below, I'm just mentioning it here in case it helps you, dear developer.

---

Refresh your browser and be amazed wiht our new functionality. You scroll down, click the 'LOAD MORE' link (which looks kinda similar to Instagrams...) and

you're then presented with some brand new posts!

Great stuff.

## Paginating Comments. AJAX first, ask questions later.

We've seen how pagination works as standard with Kaminari, we get a 'paginator' that links to pages, which correspond to a particular sub-set of our data model. This won't work for our comments, so let's not go over that again. With the comments, let's just jump straight in to some sexy AJAX action.

How does Instagram handle 'showing more' comments? Well, the maximum comments they'll show below a post is four. If there's more than that for the particular post, you'll be given a link above the four comments that says 'view all x comments' where x is the total number of comments for the post.

At this point, Instagram are clearing trolling their users though, because clicking that button goes on to only reveal 20 more comments. If there are any more than that, you'll get a new link that says 'load more comments'. It'll just sit there all smug as proof that 'load all x comments' was a complete lie.

Let's build this feature now (but a little different). I'm *not* going to troll my users, but feel free to if you want. I'm simply going to add a link above the comments that says 'view all x comments' and then loads them all.

What I want you to do now is have a ponder about how you could implement this yourself. How will this differ from your posts pagination? Well first, the order will be a bit different... you might also have to get specific regarding the element id's that jQuery will be playing with too... oh yeah, you might have to jump through some crazy hoops to get it working on the index...

Oh, I'm just thinking out loud in writing, go on, give it a go!

I want to quickly mention here, that this exact feature has had a lot to do with this article being so late. This drove me mad for a few days trying to work out the best way to implement it. I'm happy with what I ended up with, BUT if you can recreate exactly what Instagram does OR if you can implement a feature where you click 'load more comments' and it loads them incrementally in a more standard way, please, leave me a comment at the bottom of this article so I can add it on (with full credit of course, you pessimist.)


---

Before I give you my solution, I want to tell you the issues I faced when implementing the 'show more comments' feature. I moaned briefly about this above.

- The index. My god, what a pain. You can't assign `@comments` in your controller of course because you're not dealing with a singular `@post`. So how then, do you paginate your comments? If you call a standard `remote: true` AJAX event on a link or button, you're still going to have troubles accessing the specific post's comments.
- You can't paginate the comments easily on the index (did I mention this?). You can try to add `= paginate post.comments.page params[:page]` for each posts comments but try to add the paginator for that collection and you're going to have a bad time.
- Want to paginate comments on the show action of an individual post? No problemo, but that index...

**So how did I achieve my result?**

It turns out that the remote AJAX calls can return many things, javascript, JSON and even a whole bunch of html. So what I did is to write some javascript that would listen for a click on a `more-comments` class and return html in the form of a `_comments.html.haml` partial that iterates through each of a post's comments and renders them via a `_comment.html.haml` partial.

It's not as complex as it may sound and I assure you it'll make more sense after looking at the code. First, here's my adjusted portion of the `_post.html.haml` partial:

```
.comments{id: "comments_#{post.id}"}
  - if post.comments.any?
    .paginator{id: "#comments-paginator-#{post.id}"}
      - unless post.comments.count <= 4
        = link_to "view all #{post.comments.count} comments",
post_comments_path(post), remote: true, class: 'more-comments',
data: {post_id: "#{post.id}", type: "html"}
    = render post.comments.last(4), post: post
```

A bit gross, right? Yeah, it should be tidied up and thrown into a helper method but let's just deal with this for the moment. I want you to notice a few things:

- We're adding `#{post.id}` inline ruby in multiple positions in order to identify the specific post we're referring to in our jQuery. We're also giving our `link_to` helper method a html 5 data attribute of `data-post-id` that is set to the id of the post for the sake of our jQuery too.
- Next, notice that we're rendering `post.comments.last(4)` on the last line, we're not even using kaminari's pagination in this solution.
- Above the `link_to` method, we've got an 'if' statement that will only reveal the 'view all posts' link if there are greater than 4 comments for that specific post, otherwise the link would be redundant.
- Inside the `link_to` helper method, we've got the `remote: true` attribute, but we're also specifying that we want html to be returned in the data attributes hash.

Good? Good. It's worth noting what our `link_to` helper method is actually *linking to* also. We're linking to `post_comments_path(post)` which say, for the post with an id of 23, would be `/posts/23/comments`. This will be calling the index of the comments controller.

Ok, so we've got our link set up and we have some dynamically generated post.id's around the place so we can manipulate the correct items. Let's now see what we're doing in our javascript. I've stored this file in the standard javascripts folder, `app/assets/javascripts` and called it, `loadMoreComments.js`.

```
$( document ).ready(function() {
  $('.more-comments').click( function() {
    $(this).on('ajax:success', function(event, data, status,
xhr) {
      var postId = $(this).data("post-id");
      $("#comments_" + postId).html(data);
      $("#comments-paginator-" + postId).html("<a id='more-
comments' data-post-id=" + postId + "data-type='html' data-
remote='true' href='/posts/" + postId + "/comments>show more
comments</a>");
    });
  });
});
```

I'm going to humour you and run through this line by line. If you don't need me to, just skip the dot points below and bask in your smugness.

1. Wait for the document to be ready before running the code.
2. Listen for a click event on the `.more-comments` classes (each of our 'view all x comments' links).
3. Once the AJAX call has been successful, move on to the next lines.
4. Assign a `postId` variable based on the contents of the `data-post-id` html attribute (which we assigned in our post partial above).
5. Replace the `#comments_ + postId` div with the contents of the returned AJAX data.

Remember, we're returning html in our AJAX call. Based on our `link_to` method, we're sending a GET request to the comments of a specific post id. What else does this mean? We're going to need to create an `index` action within our comments controller in order to present some html to send over.

Open up the comments controller found at `app/controllers/comments_controller.rb` and add the following code above your create and destroy actions.

```
def index
  @comments = @post.comments.order("created_at ASC")

  respond_to do |format|
    format.html { render layout: !request.xhr? }
  end
end
```

So we're returning all of the comments for a post in an ascending order. We're responding to html (as is the default) but we're also adding a little bonus. We're requesting that the application's layout view is *only* rendered as part of this request, **if** the request isn't made with AJAX. Simply put, we won't get the added layout html returned during an AJAX call, which is great news, we only want the extra comments from this index after all, not an extra navbar!

What else do we need? How about a view for our new controller action? We need to tell Rails what html we want rendered after all. Here's my new `index.html.haml` file found under the `app/views/comments` folder.

```
= render @comments, post: @post
```

Well that was brief! What are we doing here? We're rendering each of our`@comments` via our existing `_comment.html.haml` partial that lives in the `views/comments` folder. We're passing it the `@post` instance variable from our index action for the sake of the attached delete feature.

Quite simple, we simply render the `_comment.html.haml` partial for each comment.

Now, here's the last piece of the puzzle, our unchanged `views/comments/_comment.html.haml` partial.

```
#comment
  .user-name
    = comment.user.user_name
  .comment-content
    = comment.content
  - if comment.user == current_user
    = link_to post_comment_path(post, comment), method: :delete,
data: { confirm: "Are you sure?" }, remote: true do
      %span(class="glyphicon glyphicon-remove delete-comment")
```

This should be familiar as it remains unchanged from what we built in Part 2 of the guide.

Now that it seems like it's all working and I even used the term 'last piece of the puzzle'... it's *still* not quite right.

**There's another last peice to the puzzle and I LIED!**

If you've been following along up until now, I want you to do something. Click some 'view x more comments' on a few of your index posts and ensure that it works. Now, click on an individual post and attempt it there.

**NOPE!**

Why?

Turbolinks. It can conflict with jQuery when waiting for `$(document).ready`. Luckily for us, it's an easy fix. Add `gem 'jquery-turbolinks'` to your gemfile and the line `//= require jquery.turbolinks` to your `application.js` file.

Retry the experiment above and you'll have a win. Finally...

It won't look great at the moment, so if you want to add some pizazz, add the following scss to your `app/assets/stylesheets/application.scss` file:

```
.comments {
  .paginator {
    margin: 0.5em;
    .more-comments {
      color: #A5A7AA;
    }
  }
```

---

The above process seems complex for what seemed initially to be a simple feature but it really *isn't* so complex. The flow is simply this:

**Click a link with remote: true that expects html to be returned -> Use jQuery to pick up on that click and add the retrieved data to a specific point on the page**

We just needed to add our extra partial and some dyanamic id's and data-attributes to make this simple flow work correctly!

## The End, for now

Let's run through what we achieved in Chapter 3.

- We beautified our forms so that users will actually want to join our glorious, completely original social network.
- We've got a great little image preview feature on our post uploads.
- We're only loading a specific number of posts per page now in order to stop our server from melting down. We append the next page of posts without

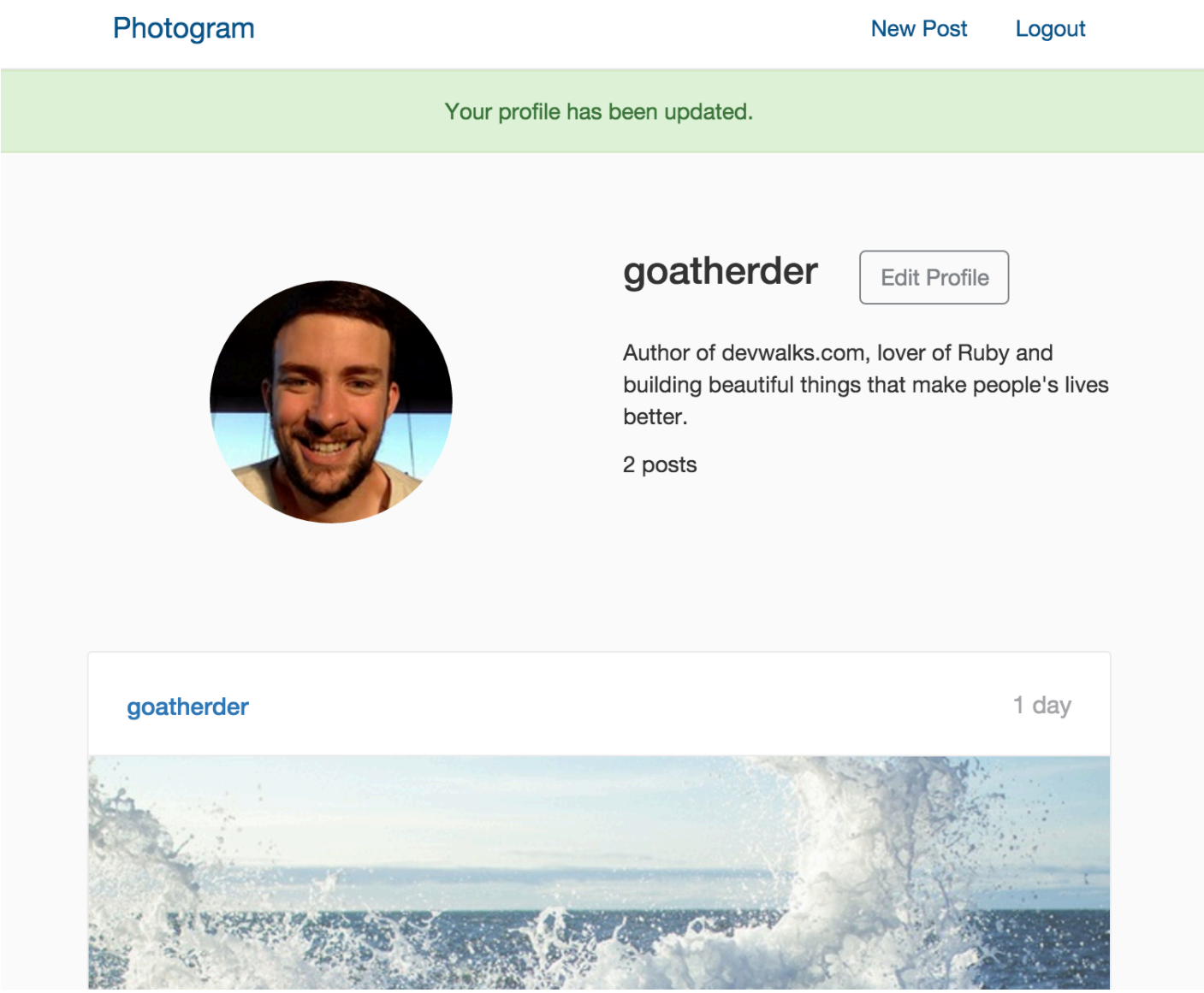
refreshing the page.
- We now only list 4 comments for each post by default.  Want more? Click the button and you'll see a full list without a page refresh.

Pretty good!

In the next chapter, let's build user profiles!

# Chapter 4 - Presenting Pretty Profiles

In this chapter, let's build a pretty handy feature for any social network: user profiles.




Unsatisfied with a single stream of posts that mixes everyone together, we're going to create profile pages for our users! We'll let our users choose a photo for their profile picture and let them create a brilliant bio to sum up their existence at the top of their profile. We'll be able to check out our user's profile pages by clicking on their names and we'll be presented with a stream of their posts and only their posts. I also want to make sure the url looks nice by ensuring we're using the user name, rather than the id of the user (yuck).

Let's begin.... meow.

## An Explanation of the Feature

Let's think about what a profile page is, in order to flesh out the features. We want to have:

- A user name in the URL.
- A profile picture that the user can change if they'd like (we'll have a default image). If you're on your own page, you'll be able to click the image and change it.
- An editable bio for each user (which is empty by default). If you're on your own page, you'll be able to click an 'edit profile' button that will let you change your bio.
- The url for editing your profile details should be '/:user_name/edit'
- A stream of only the appropriate user's posts shown below the profile picture and bio. We'll stick to the vertical stream in this guide but you could always implement something like masonry.js if you'd like something similar to the grid-view that Instagram uses by default.

## Presenting Pretty Profiles

Let's start our build by building out our user profiles.

First, let's make sure our usernames are 'clickable' links, rather than just the placeholders they've been thus far. This will let our users actually visit the profile pages in a simple way (just as Instagram has handled it!).

Track down your `app/views/posts/_post.html.haml` partial view and let's add a `link_to` helper to the user name at the top of the post. Replace the existing code:

```
.user-name
  = post.user.user_name
```

With this:

```
.user-name
  = link_to post.user.user_name,
profile_path(post.user.user_name)
```

Please note that we don't have a `profile_path` route defined yet, so if you navigate over to your dashboard, you'll see a big fat error. Which is good, we can fix that!

Now, we're going to be creating a specific route for this path and not using Rails 'resourceful routing'.

*Why?*

Well first, I don't want the users profile to look like this: `photogram.com/users/benwalker`. I want it to simply be `photogram.com/benwalker`. I also want the route to use the user-name rather than the user's ID. This is possible with the standard route, but let's remove the `users` namespace altogether.

Go on, give it a go yourself. You can read about routing in Rails here.


---

Let's first create our new route. In your `routes.rb` file, add the following line:

```
get ':user_name', to: 'profiles#show', as: :profile
```

Want to read what's going on here?

- We're going to GET (rather than POST or PATCH etc) our user name path. Why are we using a symbol? Well, that let's us use a dynamic parameter for the sake of our route. If we just wrote `get 'user_name'`, our route would be fixed as 'photogram.com/user_name'. BUT with our current setup, we can pass user names as the param and get all dynamic and snazzy.
- We're going to get the details for what to do with this route from the 'profiles' controller and the 'show' action. We don't have a profiles controller yet, so we'll have to take care of that.
- The `as: :profile` line let's us use handy Rails routing helpers in our code. This will let us user the `profile_path` helper (just as we've used `post_path` or `posts_path` previously).

Beautiful! Now if you were to reload your page, you should be error-free and you

should also be able to hover over a post's user name and be greeted with a beautiful link to their profile, just where we wanted.

If you were crazy enough to click on said link, you'll be greeted with yet another error. That will teach you for being too bold and that you should only do things exactly how and when I'm saying (I'm joking please don't ever do that. Click on all of the things).

You should be ok to generate your own controller. Remember, we want it to be called 'profiles' and we want a single 'show' action for the moment.


---

Let's generate our new controller to control our profile. Jump on into your terminal of choice and tap away these words:

```
rails g controller profiles show
```

After a blur of progress, you'll be greeted with a few extra files in your text editor! Hurrah!

Now if you were to jump back to Photogram and click on a user's user name on a post, you'll actually be linked to the appropriate route which is 100% badass. Now it's time to make that route actually do something useful.

First, we'll need our 'show' action to actually point us to some useful data. Remember, we want to only show the specific user's posts. We also want it to show a bio and profile picture, but let's start with the posts.

Give it a go yourself. Remember, we have a param in our url that will be quite useful when trying to find the appropriate user. We might even be able to use associations to keep the code nice and clean!


---

Alright, let's find the right posts for each profile's specific user. We have a user name to work with in our url, so let's track down our user. Under the 'show' action, tap away until this appears:

```
def show
  @posts = User.find_by(user_name: params[:user_name])
end
```

How good is that? But wait, that's only going to get us a user object, how do we get their posts?

Easy Peasy!

We just add a `.posts` method to the end. The code will look like:

```
def show
  @posts = User.find_by(user_name:
params[:user_name]).posts.order('created_at DESC')
end
```

Beautiful. Notice that I've ordered the posts too, just for good measure.

Now that we have an instance variable to work with, let's get the view built to use that data.

First, rename the `app/views/profiles/show.html.erb` to `show.html.haml` (if you value your sanity and use haml) and then just copy and paste your existing code from the index view of your posts controller. That should look something like this:

```
-@posts.each do |post|
  = render 'posts/post', post: post
```

Good work.  Click on a user name and you should be greeting with a list of their posts and their posts only.

Give yourself or a nearby human a high five to celebrate.

## Expressing Egos with Elegant Essays

It's bio and profile picture time!  If you want a refresher on the features we're building here, go back up to the test section and have a quick scan.  If a profile page belongs to us, we want to be able to click an 'Edit Profile' link and there, have the ability to change our profile picture or our bio, save the info and be redirected back to our profile to be greeted with our new information.

Not only this, I want the profile route to NOT be 'photogram.com/users/1/edit'.

**NO!**

It's ugly and I don't like it!  I want to simply be 'photogram.com/benwalks/edit'.

First, let's create our link.

No wait, first **YOU** create the link.  I'll be waiting over here, behind my monitor, for you to finish.

Please hurry though, I'm tired and thirsty.


---

Thank the gods you're finished, I've been waiting for at least a few micro-seconds for you to completely ignore the 'Your Turn' goat and just scroll on down to this paragraph.

Let's get rolling with this linking business.

At the top of the `app/views/profiles/show.html.haml` file, before the posts block, add a `link_to` helper, *if* you're the owner of the profile.

```
- if @user == current_user
  = link_to 'Edit Profile', edit_profile_path(@user.user_name)
```

Notice our new instance variable there? Let's add that to our profiles controller now.

Above the `@posts` instance variable, add this line:

```
@user = User.find_by(user_name: params[:user_name])
```

Which will give us the user object for the current profile we're looking at. Also note in the above `link_to` helper, that we're using the `user_name` of the user, not their id. Remember that we're using the user-name as the reference, not the id in this scenario.

We're also going to need to add a route for our new edit profile path. In your `routes.rb` file, add this line somewhere sensible:

```
get ':user_name/edit', to: 'profiles#edit', as: :edit_profile
```

Let's go over this again quickly for repetitions sake. We're GETting a route, and using the edit action within the profiles controller to do something with it. We want to be able to refer to this particular route as `edit_profile_route`.

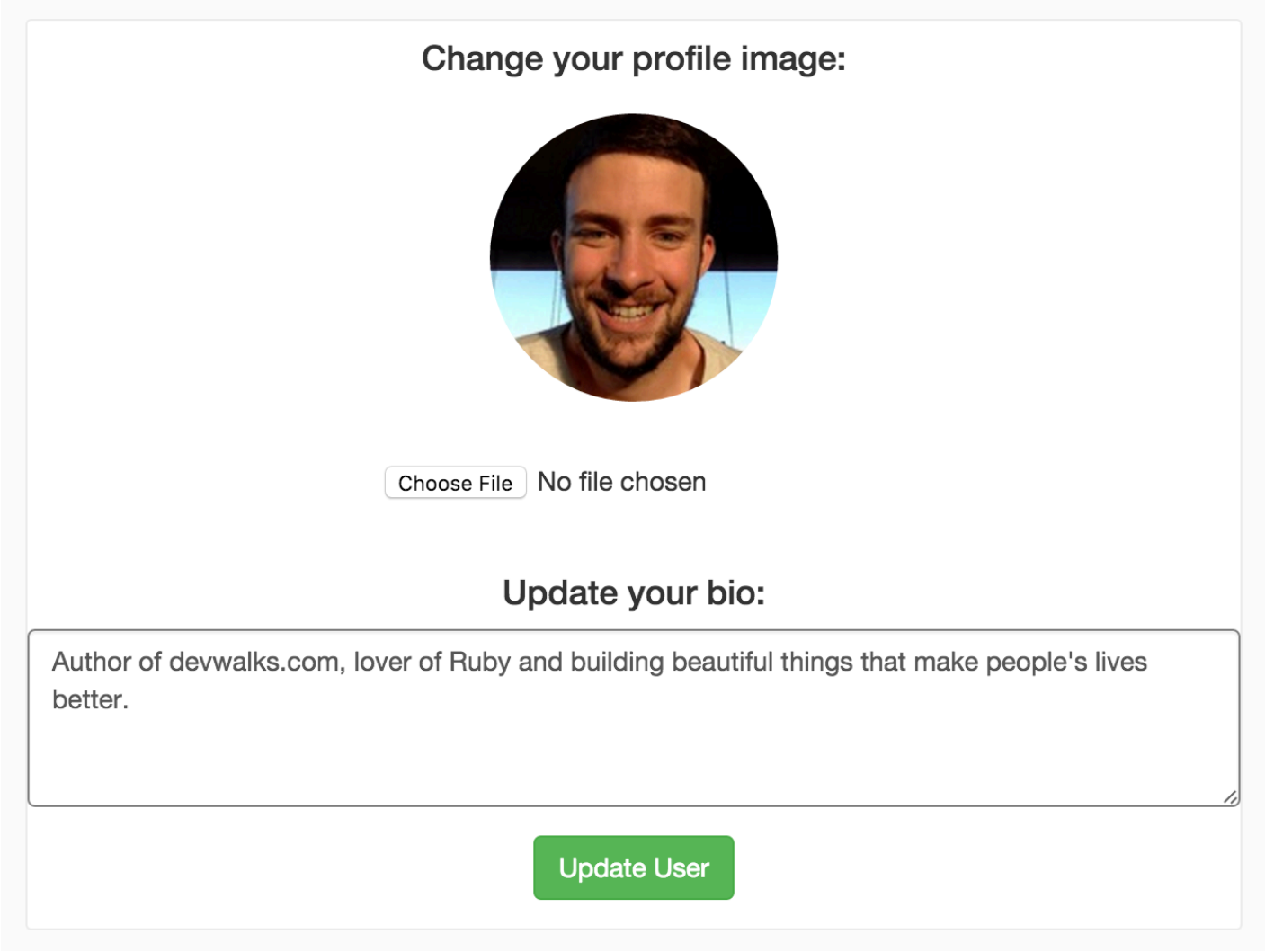
Nice!

If you were to refresh a profile page at this point and hover over our newly created link, you should see a nice looking path that you'd be routed to. If you were to click on said link, you'll be greeted with errors.

### We need a view Ben

Yes, yes of course. Let's build one now. Instagram's 'Edit Profile' page is a little complex for our needs at the moment AND it won't let me update my profile picture via desktop, so let's divert from what they're doing for a moment and go completely wild.

This is what our 'edit profile' page is going to look like:




Looks pretty nice, right?? We've simply got a image upload field as our top row, as per our 'New Post' preview and a single 'Bio' text field beneath.

This is **EXACTLY** the kind of thing you can do yourself. Refer back to your old code if you need to, I can guarantee I will be. Remember though, you're going to need a new route and controller action to POST this information to AS WELL as the route / controller action that you'll GET...

Just something to consider hombre, good look.


---

Let's compare notes.

First, let's create a new, basic route to GET our profile edit page. Open up the `routes.rb` file and tap away on your keyboard device until the following appears:

```
# Existing route
get ':user_name', to: 'profiles#show', as: :profile
# New route underneath
get ':user_name/edit', to: 'profiles#edit', as: :edit_profile
```

Now that we have a route we'll have to display a page! Let's create the edit view.

Create a new file under the `app/views/profiles` folder and simply call it `edit.html.haml` or `.erb` (shame on you).

Remember what our simple view entailed, we just want a image picker and preview as our top row and a text field for the user's bio as the bottom row. This could look something like the code below but as per usual, I highly recommend you attempt it yourself. If you've made it this far in the guide, you are more than capable! Use our old 'edit post' view for inspiration.

In `app/views/profile/edit.html.haml`:

```
.posts-wrapper
  .post
    .post-body
      .image-wrap
        = form_for @user, url: {action: "update"}, html: {
multipart: true } do |f|
          .row
            .col-md-12
              .form-group.text-center
                %h4 Change your profile image:
                .img-circle
                  = profile_avatar_select(@user)
                = f.file_field :avatar, onChange:
'loadFile(event)'
          .row
            .col-md-12
              .form-group.text-center
                %h4 Update your bio:
                = f.text_area :bio, label: false, rows: 4
          .form-group.text-center
            = f.submit "Update Profile", class: 'btn btn-
success'
```

Not too tricky to comprehend, right? We have two rows for our form, where we refer to an instance variable, `@user`. You'll notice a new helper method, `profile_avatar_select` in there, but we'll get to that in a moment.

We have our form basics, let's quickly add a few lines of css to our `app/assets/stylesheets/application.scss` file:

```
#user_avatar {
  margin: 20px auto;
}
```

Lovely.

Let's now create the required `@user` instance variable in our profiles controller under the 'edit' action now in order to avoid inevitable errors.

What are we referring to in this form? Just the user object, right? With the form, we just want to be able to update two specific parts of the user object, the bio and the avatar. Let's point `@user` to the specific user object. Create an edit action in

your profile controller like so:

```
def edit
  @user = User.find_by(user_name: params[:user_name])
end
```

So we have an action in our controller with an instance variable pointing to the appropriate user object and a form in our view.  Now all we need is to actually have 'bio' and 'avatar' columns in the database for our users to update!

We'll tackle the avatars first.

So far in our Instagram build, we've used the fantastic Paperclip gem to handle our file uploads (images in this case), and we'll keep utilising it for our avatars now.

Checkout the great docs over at github and implement it for yourself!  Take particular notice of the 'Models' and 'Migrations' sections.


---

Here's how I added avatars to my User model.  In my terminal, I tapped away:

```
rails generate paperclip user avatar
```

And then I jumped into my User model found at `app/models/user.rb` and added the following lines:

```
has_attached_file :avatar, styles: { medium: '152x152#' }
validates_attachment_content_type :avatar, content_type:
/\Aimage\/.*\Z/
```

That should be it for the moment (until I migrate the changes to my DB). Let's now add a user bio to the model.

**Here's your challenge, should you choose to accept it.**

Add a new text column to the users table that we'l use for our bios. Call it 'bio'.

Do it all by yourself.

For extra points, generate the full migration in your terminal so you don't have to adjust the migration file itself before running the migrate rake task (which we'll do in a moment). Pro-tip: Read the Rails docs re: migrations.

Here's the goat for inspiration.


---

You did it yourself, right? Superb. Just in case you're suffering a bad case of the nerves, here's the exact line to generate the migration file from scratch:

```
bin/rails generate migration AddBioToUsers bio:text
```

Now let's run our migration so that our changes are put into effect.

```
bin/rake db:migrate
```

New columns successfully created! Last but not least, let's create the new `profile_avatar_select` helper method within our edit form. Spoiler alert, it's *incredibly* similar to our old helper method we used when creating and editing our image posts. Check out what that looked like and try to implement this yourself.

Here's how my new method looks, it lives in

the `app/helpers/application_helper.rb` file:

```
def profile_avatar_select(user)
  return image_tag user.avatar.url(:medium),
                   id: 'image-preview',
                   class: 'img-responsive img-circle profile-
image' if user.avatar.exists?
  image_tag 'default-avatar.jpg', id: 'image-preview',
                                  class: 'img-responsive img-
circle profile-image'
end
```

The functionality with this method is almost exactly the same as our older method. If the user already has an avatar, display that image. Otherwise, display the 'default-avatar.jpg' image. "What is this 'default-avatar' image?", I hear you scream at your monitor. Well... You'll have to create one.

Whatever default image you decide to use, make sure it's a sensible size and make sure it's named 'default-avatar.jpg'. Pop it in the `app/assets/images` folder.

Now! Revisit your 'edit profile' view and you should be faced with a lovely little form. Oh wait, you're faced with an error complaining about no 'update' route. Lame.

Think about it, what are we actually submitting to? What's going to do the work with our lovely new information (bio and avatar)? How would this work with any other form?

We're going to need a new action in our Profiles Controller to handle the submitted form AND we'll need a route to submit the form *to*. Let's get to it.

Route first.

Sticking with our theme of creating nice little routes, let's create a new route for our 'update' profile action.

In your `config/routes.rb` file, add this line:

```
patch ':user_name/edit', to: 'profiles#update', as:
:update_profile
```

We're adding a PATCH route for `:user_name/edit` which points to a 'update'

action in our profiles controller. So, it would make sense at this point to create our new 'update' action.

In your profiles controller, add the new action:

```
def update
end
```

Refresh your edit profile form and it should now actually present something! Hurrah!

Ensure that the image preview is working OK, and let's continue with the form submission logic.

What happens if we submit our form now with an empty update action?

**ERRORS GALORE**

Of what variety? The 'no template' variety. We've had this particular error in the past articles and what it means is this: Make a template. What does it mean if you don't want a template, just like our scenario? Our update action should simply update the user object on the server, not actually display anything to the user.

What we need to do is write some logic for saving our new data and then we'll redirect the user somewhere useful with a nice old flash message displayed on success or failure.

Tap away on your keyboard until your `update` action in your Profiles Controller looks like this:

```
def update
  @user = User.find_by(user_name: params[:user_name])
  if @user.update(profile_params)
    flash[:success] = 'Your profile has been updated.'
    redirect_to profile_path(@user.user_name)
  else
    @user.errors.full_messages
    flash[:error] = @user.errors.full_messages
    render :edit
  end
end
```

And add a new private method below our other actions in order to set context for `profile_params` used above.

```
private

def profile_params
  params.require(:user).permit(:avatar, :bio)
end
```

Let's go through our update action one line at a time, if you know what's going on here, feel free to move your eyeballs past the dot points below and give yourself a pat on the back for being so clever.

- We set the `@user` instance variable by finding their details via the url params (the user's user name).
- If we can update the user object with the profile params that are sent by our form, flash a victory message, and redirect the user back to their profile page.
- If something goes horribly wrong, flash an error message and render the edit page once again, with the user's form entries intact.

What have we achieved here? Our users can now create and edit their profile data! Even if you didn't skip the dot points, give yourself a pat on the back now and kiss the closest human being (with permission) to celebrate this wonderful moment.

## But Ben, I still can't see everyones avatar or bio!

Oh yeah, let's fix that now by being clever!

You should do it yourself of course, but first, let's think about what we want to achieve here.

The users active record object is already available with the `@user` instance variable that we've already set. All we want is to access the `@user.avatar` and `@user.bio`. At the very least, add the data to your profile pages and we'll beautify it in a moment.

Remember, you'll learn FAR more by doing it yourself, the struggle is key. Don't disappoint goat, he's very judgemental.


---

Let's compare notes.

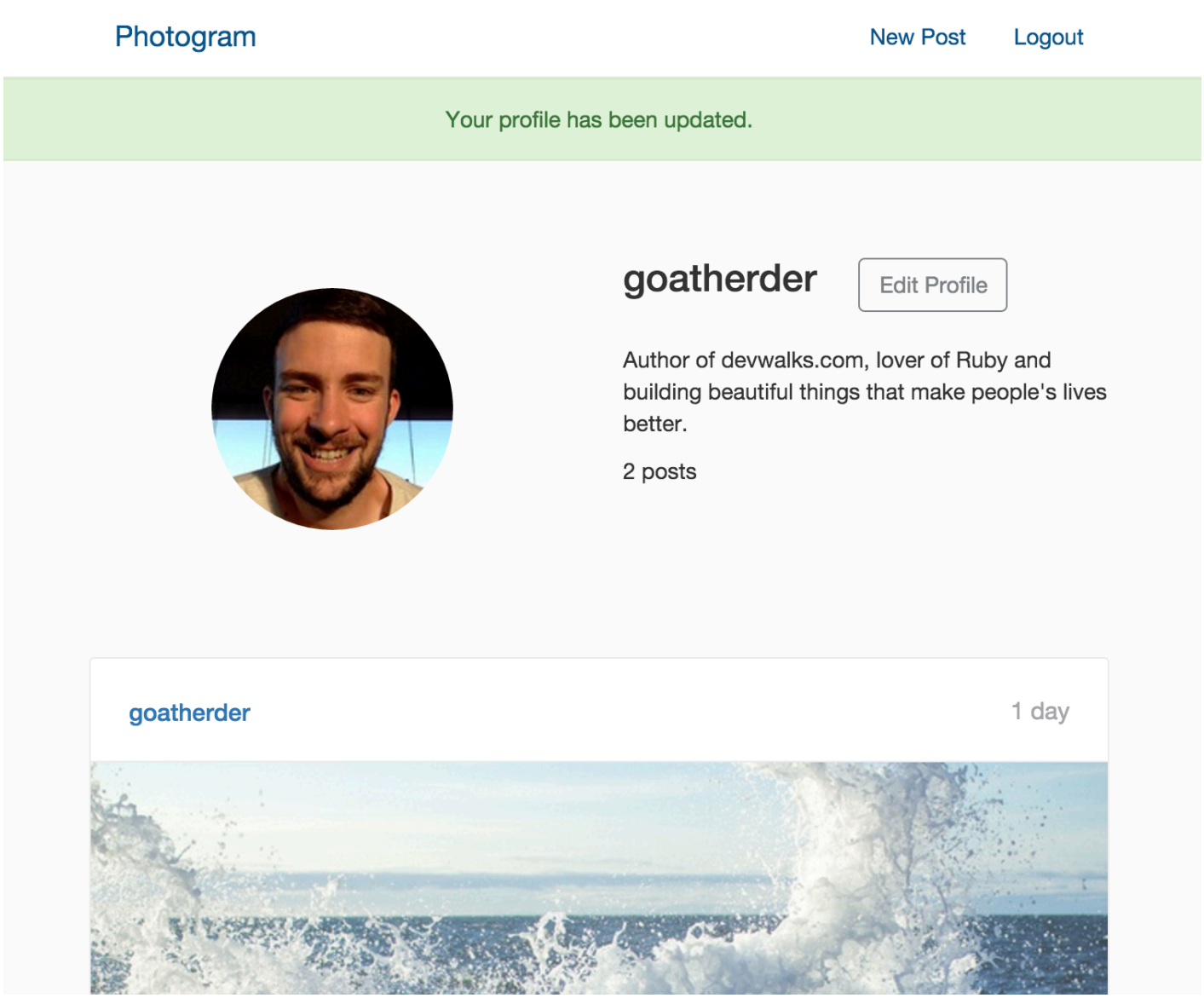
Here's the quick and dirty way to get the user's profile data (the avatar and bio) onto their profile page. Jump into the `app/views/profiles/show.html.haml` file and replace everything above the `@posts.each do |post|` block with this:

```
.posts-wrapper
  .row.profile-header
    .col-md-6
      .img-circle
        = profile_avatar_select(@user)
    .col-md-6
      .user-name-and-follow
        %h3.profile-user-name
          = @user.user_name
        %span
          - if @user == current_user
            = link_to 'Edit Profile',
edit_profile_path(@user.user_name),
                                        class: 'btn edit-button'
      %p.profile-bio
        = @user.bio
      .user-statistics
        %p
          = pluralize @user.posts.count, 'post'
```

If you were to refresh your page now, you'll be presented with our profile data! Add the following to your `app/assets/stylesheets/application.scss` file:

```
.profile-header {
  padding: 20px 0;
}

.profile-image {
  margin: 20px auto;
  height: 152px;
  width: 152px;
}

.user-name-and-follow {
  display: inline;
}

.profile-user-name {
  display: inline;
}

.edit-button {
  border-color: #818488;
  color: #818488;
  margin-left: 20px;
}

.profile-bio {
  margin-top: 20px;
}
```

And refresh!  You're profile page should be looking glorious.




Fantastic. We've not got fully functioning profile pages working for our users... but wait...

## I can kind of edit everyone's profile...

Even though we have some code in our `profiles/show.html.haml` view that only shows the 'Edit Profile' button if you're the owner of a particular profile, let's see what happens if we directly navigate to the url for editing another user's profile?

*We can edit it.*

This is very bad for obvious reasons. Let's fix now. If you have an elephants memory, you'll recall that we fixed this very issue for our Posts feature as well in a prior article!

Check out your existing code and see how you could use a very similar feature here to protect our user's profiles from tomfoolery.


---

Jump on into your `app/controllers/profiles_controller.rb` file and add a new private method as per the below code:

```
def owned_profile
  @user = User.find_by(user_name: params[:user_name])
  unless current_user == @user
    flash[:alert] = "That profile doesn't belong to you!"
    redirect_to root_path
  end
end
```

Back at the top of that file, just below the `:authenticate_user!` you can add the `before_action` for our new method:

```
before_action :owned_profile, only: [:edit, :update]
```

If you're testing, check your tests now. If you're not, manually test a visit to the edit path for a different user. You should be denied access and you can therefore give yourself a high five.

We also want to make sure that only registered users can see our profile pages, so make sure you add the `authenticate_user! before_action` in your controller too.

Either above or below your `owned_profile` action, add:

```
before_action :authenticate_user!
```

Hell yeah.

## Tidying Up

Notice anything fishy about our Profiles Controller. Maybe something looks a little... off?

We're setting the `@user` instance variable constantly! How could we DRY up our code? Have a ponder or simply checkout how we handled it in our Posts Controller. We can tidy this right up very easily.


---

Once again, create another private method in your Profiles Controller. Simply call it `set_user` and make it look somewhat similar to the code below:

```
def set_user
  @user = User.find_by(user_name: params[:user_name])
end
```

Add the before action for your new `set_user` method to the top of the controller file:

```
before_action :set_user
```

At this point you can go forth and delete all lines of code which we've handled by setting the `@user` instance variable with the before action. Your controller will look much nicer as a result.

Last but not least, let's link up the user names shown on comments and captions to the appropriate user's profile page. Try it yourself first, and report back to me once you've had a good try.


---

It's super easy! First, let's take care of linking up the user names shown in comments. Jump into your comment partial file at `app/views/comments/_comment.html.haml`. Adjust the dynamic user name line, so it looks like the third line shown below:

```
#comment
  .user-name
    = link_to comment.user.user_name,
profile_path(comment.user.user_name)
```

Now, let's fix the user name shown for each post's caption. In the `app/views/posts/_post.html.haml` file, adjust line 16 (that once again shows the user name) to link to the profile page.

```
.user-name
  = link_to post.user.user_name,
profile_path(post.user.user_name)
```

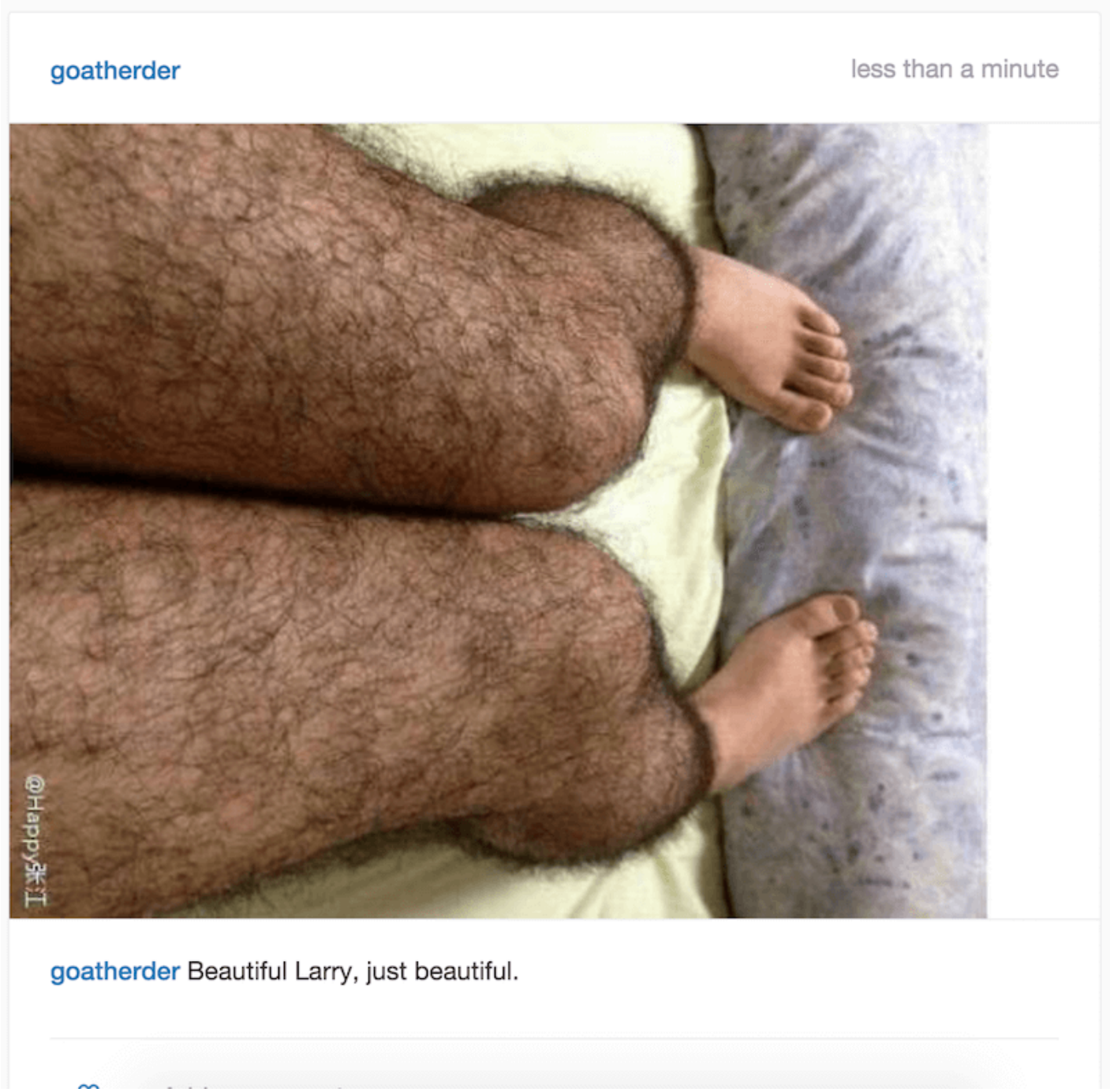
Check it out for yourself and bask in your own glory.

## That's it, until the next chapter

In the next chapter we'll attack 'likes' on posts and how we can build something similar to Instagram's like system. See you there.

# Chapter 5 - Liking Larry's Legs

Larry has a spectacular set of pins, check them out:




Larry needs a feature on his favourite social network (Photogram) where he can accumulate likes for his posts, so he can bask in the glory of sharing his lovely legs with the world. And in this Rails tutorial, we're going to give it to him.

## How do we want our likes to work?

- No matter the context that I'm viewing a specific post, I'd like to be able to click a little heart icon under the image to add a 'like' to the post.
- Once I've liked a post, I'd like to see the 'like' count go up and also have my name added to the post's list of likers.
- My name on the list of likers should link back to my own profile page.
- I want to be able to unlike a post once I've liked it, in case I made a mistake.

Perfect! Now we know exactly what we need to build. Now, let's get building.

## The button, the likers

Cast your mind back to the comments guide and you should remember that we actually already created a placeholder like button for our users:

The problem is... **it's completely useless.**

Let's stop tricking our users and actually make it do something now. This feature will actually be quite familiar to you long time readers, we implemented something very similar for our likes. Here's how it'll work technically:

- User clicks on the like button.
- AJAX call is made to a 'like' action within the Posts controller.
- The like action increments the likes on the post by one and adds the likers user name to the list of likers for the appropriate post (we'll use the `acts_as_votable` gem for this.
- The heart icon for the post will turn solid red (instead of the default).

Try to patch together what you learnt in the previous tutorials. How do we tell that button to make an AJAX call? How do we use jQuery to amend the new user-name to the list of likers for the post and how do we change the icon? Check out the 'comments' tutorial for a reminder.

Also, check out the `acts_as_votable` documents here. They should cover just about everything you could possibly want.

**Let's get building!**

First, let's make sure our heart icon is clickable and also ensure each heart icon has a post-specific id, so we can ensure we're dealing with the correct post on a page of many posts.

Jump into your `app/views/posts/_post.html.haml` file and adjust line 24 to 27 to the following:

```
.comment-like-form.row
      .col-sm-1
        =link_to '', like_post_path(post.id), remote: true,
                                              id: "like_#
{post.id}",
                                              class: "glyphicon
glyphicon-heart-empty"
```

All we've done is add a post-specific id to each comment area and we've also moved our heart icon to the `link_to` helper itself. We've also added the `remote: true` property to the link, just as we added this same property to our forms in the comments tutorial.

Now, we need to make sure that we're linking to something useful. Something useful, like our new to be created 'like' action in our posts controller.

Jump in the posts controller now and add the new action below your existing `destroy` action:

```
def like
end
```

Now that we have a new action, we'll need to upgrade our `routes.rb` file to ensure that our action is accessible to our `link_to` helper. We'll make our like action a 'member' of the posts collection so we can still access the :id of the post we're referring to in the url. This will make more sense once we've completed it. In your routes file, adjust your posts collection to look like below:

```
resources :posts do
  resources :comments
  member do
    get 'like'
  end
end
```

If you were to refresh the dashboard of your Photogram feed and then mouse over the like link now, you should notice the link in the bottom of your browser will point to `posts/:id/like` which is perfect.

Let's get the 'like' action doing something useful now, let's make it increment the 'likes' of the appropriate post and also, list the user_name of the likers.

It's time to install the `acts_as_votable` gem. Add the gem to your gemfile:

```
gem 'acts_as_votable', '~> 0.10.0'
```

And run the `bundle install` command in your terminal.

Once the gem is downloaded and installed, you should be good to continue on this wonderful journey.

As per the `acts_as_votable` docs, we now need to run a migration to create the voters table.

```
rails generate acts_as_votable:migration
rake db:migrate
```

We can now add the `acts_as_votable` method to the top of our Post model file found at `app/models/post.rb`. The whole file should now look something like this:

```
class Post < ActiveRecord::Base
  acts_as_votable

  belongs_to :user
  has_many :comments, dependent: :destroy

  validates :user_id, presence: true
  validates :image, presence: true
  validates :caption, length: { minimum: 3, maximum: 300 }

  has_attached_file :image, styles: { :medium => "640x" }
  validates_attachment_content_type :image, :content_type =>
/\Aimage\/.*\Z/
end
```

Fantastic! Now, we can vote on our posts. More specifically, we can like posts!

## Sneaky Side-note

This is a fantastic time to play with the new functionality in the rails console, so go ahead and have a play (it'll make it easier to understand what we're about to do moving forward). Here's an example, by just following along with the `acts_as_votable` docs (this assumes you have at least one user and one post existing on your application).

Run the rails console in your terminal with `rails c` and then follow along below:

```
# Creating your first like
user = User.last # Sets the user to a variable
post = Post.last # Sets the post to a variable
post.liked_by user # Creates the like as per the docs

# Let's count how many likes our post has
post.get_likes.size

# How about who's liked our post so far?
post.votes_for.up.by_type(User).voters do |voter|
  puts voter.user_name
end
```

Cool, right? It's good to play around like this because it makes the functionality concrete in your mind, once it's time to actually add it to your app. No where were

we...

## Back to it!

Let's add some substance to our 'like' action in our Posts Controller now that we have some useful methods to work with and some knowledge of how they work. Open up the posts controller file and first, make sure you add the 'like' action to your `set_post` `before_action` at the top of your file:

```
before_action :set_post, only: [:show, :edit, :update, :destroy,
:like]
```

Now, add the following lines to your 'like' action:

```
def like
  if @post.liked_by current_user
    respond_to do |format|
      format.html { redirect_to :back }
      format.js
    end
  end
end
```

So what are we doing here? `@post.liked_by current_user` uses the `acts_as_votable` gem to cast a vote for the specific post. So, *if* it works and does indeed vote for the post, we'll respond with either some javascript or html. In this guide we'll be responding with some fancy javascript but alas, that will have to wait for Step two of this particular feature build (Once a week articles are killing me!).

Believe it or not, we can now actually like posts! Refresh your Photogram dashboard, or even a single post and click the like button. You'll be greeted with.... nothing.

Lame. Refresh your page and now see what happens. Oh, ok. Nothing still. Well rest assured that something *has in fact happened* in the background, it's just that we don't have anything in our views to show our likers at the moment.

Let's fix that now!

## Editing our view so we can see who likes you.

That heading rhymes for what it's worth. Alright, now we have some functionality that let's us like posts, we just need a way to show these beautiful likes on said posts. First, we're going to move our whole 'likes' section of each post into it's own partial.

Why?

Well, as a part of the jQuery actions that we'll get into in the next part of this article, we'll actually be re-rendering that partial upon liking or un-liking a particular post so that the whole process feels 'real-time'. It's also nice to seperate concerns as a part of our application. Our new partial will seem kind of lame, but rest assured, it's for a good cause.

Jump into your `_post.html.haml` partial and edit it as below (I've included a bit of the surrounding code for context):

```
.image.center-block
      =link_to (image_tag post.image.url(:medium), class:'img-
responsive'),
                  post_path(post)
    .post-bottom
      = render 'posts/likes', post: post
      .caption
        .caption-content
          .user-name
            =link_to post.user.user_name,
profile_path(post.user.user_name)
```

So all we're adjusting is this line: `= render 'posts/likes', post: post`. What exactly is that line doing? We're rendering the 'posts/likes' partial. Note, we've included the posts directory as a part of this partial render call as we have other views (such as the 'profile' view) that aren't within the 'posts' directory. They'll need some context to find the file.

Also, we're passing the partial view some context for what `post` is. post is post.

Time to create our brand new partial view. Create a new file in the 'app/views/posts' folder and call it `_likes.html.haml`.

This new file should look like this:

```
.likes
  = likers_of post
```

So we have a div with the 'likes' class and we also have this new helper method staring us in the face. Why? It's nice to keep logic out of our views and we *will* need some logic to make this look nice.

So I suppose it's time to create this new helper method!

Navigate over to 'app/helpers/posts_helper.rb' and open that bad boy. I'll show you below what we want inside this file and explain exactly what we're doing below.

```
module PostsHelper
  def likers_of(post)
    votes = post.votes_for.up.by_type(User)
    user_names = []
    unless votes.blank?
      votes.voters.each do |voter|
        user_names.push(link_to voter.user_name,
                                profile_path(voter.user_name),
                                class: 'user-name')
      end
      user_names.to_sentence.html_safe + like_plural(votes)
    end
  end

  private

  def like_plural(votes)
    return ' like this' if votes.count > 1
    ' likes this'
  end
end
```

Quite a lot to take in right? Well, heres that same code again but with some added comments in order for you to make sense of it all:

```
module PostsHelper
  # Our new helper method
  def likers_of(post)
    # votes variable is set to the likes by users.
    votes = post.votes_for.up.by_type(User)
    # set user_names variable as an empty array
    user_names = []
    # unless there are no likes, continue below.
    unless votes.blank?
      # iterate through the voters of each vote (the users who
liked the post)
      votes.voters.each do |voter|
        # add the user_name as a link to the array
        user_names.push(link_to voter.user_name,
                                profile_path(voter.user_name),
                                class: 'user-name')
      end
      # present the array as a nice sentence using the
as_sentence method and also make it usable within our html.
Then call the like_plural method with the votes variable we set
earlier as the argument.
      user_names.to_sentence.html_safe + like_plural(votes)
    end
  end

  private

  def like_plural(votes)
    # If we more than one like for a post, use ' like this'
    return ' like this' if votes.count > 1
    # Otherwise, return ' likes this'
    ' likes this'
  end
end
```

Not so bad once you go through it step by step, right?

If you were to once again refresh your dashboard or individual post in your browser, you will now be welcomed with a lovely little 'like' area, listing all users who have liked a post (probably just yours). It might look a little strange for the moment though, you'll need to patch up some css.

Jump into your `app/assets/stylesheets/application.scss` file and copy the

new code in:

```
@import "bootstrap-sprockets";
 @import "bootstrap";

html {
  height: 100%;
}

body {
  height: 100%;
  background-color: #fafafa;
  font-family: proxima-nova, 'Helvetica Neue', Arial, Helvetica,
sans-serif;
}

.alert {
  margin-bottom: 0px;
}

/* ## NAVBAR CUSTOMISATIONS ## */

.navbar {
  margin-bottom: 0px;
}

.navbar-brand {
  a {
    color: #125688;
  }
}

.navbar-default {
  background-color: #fff;
  .navbar-nav li a {
    color: #125688;
  }
}

.navbar-container {
  max-width: 640px;
  margin: 0 auto;
}
```

```
/* ## POST CUSTOMISATIONS ## */

.posts-wrapper {
  padding-top: 40px;
  margin: 0 auto;
  max-width: 642px;
  width: 100%;
}

.post {
  background-color: #fff;
  border-color: #edeeee;
  border-style: solid;
  border-radius: 3px;
  border-width: 1px;
  margin-bottom: 60px;
  .post-head {
    flex-direction: row;
    height: 64px;
    padding-left: 24px;
    padding-right: 24px;
    padding-top: 24px;
    color: #125688;
    font-size: 15px;
    line-height: 18px;
    .user-name, .time-ago {
      display: inline;
    }
    .user-name {
      font-weight: 500;
    }
    .time-ago {
      color: #A5A7AA;
      float: right;
    }
  }
  .image {
    border-bottom: 1px solid #eeefef;
    border-top: 1px solid #eeefef;
  }
}

.post-bottom {
  .user-name, .comment-content {
    display: inline;
```

```
  }
  .caption {
    margin-bottom: 7px;
  }
  .user-name {
    font-weight: 500;
    color: #125688;
    font-size: 15px;
  }
  .user-name, .caption-content {
    display: inline;
  }
  #comment {
    margin-top: 7px;
    .user-name {
      font-weight: 500;
      margin-right: 0.3em;
    }
    .delete-comment {
      float: right;
      color: #515151;
    }
  }
  margin-bottom: 7px;
  padding-left: 24px;
  padding-right: 24px;
  padding-bottom: 10px;
  font-size: 15px;
  line-height: 18px;
}

.comment_content {
  font-size: 15px;
  line-height: 18px;
  border: medium none;
  width: 100%;
  color: #4B4F54;
}

.comment-like-form {
  padding-top: 24px;
  margin-top: 13px;
  margin-left: 24px;
  margin-right: 24px;
  min-height: 68px;
```

```
  align-items: center;
  border-top: 1px solid #EEEFEF;
  flex-direction: row;
  justify-content: center;
}

/* ## Wrapper and styling for the new, edit & devise views ## */

.edit-links {
  margin-top: 20px;
  margin-bottom: 40px;
}

.registration-bg {
  padding-top: 4em;
  height: 100%;
  background-image: image-url('regbackg.jpg');
  -moz-background-size: cover;
  -webkit-background-size: cover;
  -o-background-size: cover;
  background-size: cover;
}

.login-bg {
  padding-top: 4em;
  height: 100%;
  background-image: image-url('loginbg.jpg');
  -moz-background-size: cover;
  -webkit-background-size: cover;
  -o-background-size: cover;
  background-size: cover;
}

.log-in{
  margin-left: auto;
  margin-right: auto;
  text-align:center;
  border: none;
}

.panel {
  border-radius: 8px;
}

.panel-heading{
```

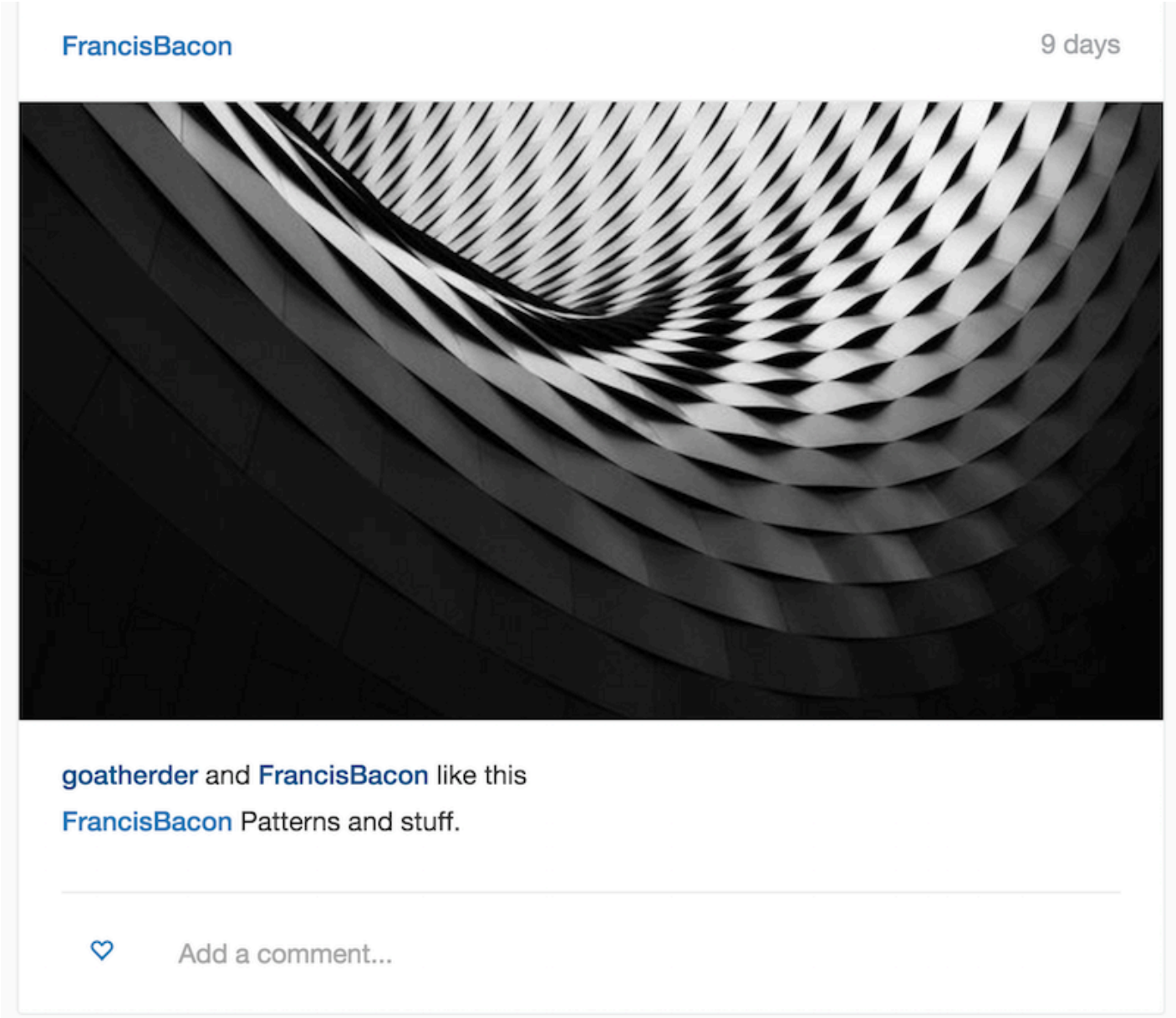
```
  h1 {
    text-align: center;
  }
}

#image-preview {
  margin: 0 auto;
}

#post_image {
  padding: 1em;
  margin: 0px auto;
}


#paginator {
  color: #4090DB;
  height: 120px;
  width: 120px;
  margin: 60px auto 40px;
  position: relative;
  display: table;
  border: 2px solid #4090DB;
  border-radius: 50%;
  :hover {
    border: 1px solid #2D6DA8;
    border-radius: 50%;
  }
}

#load_more {
  display: table-cell;
  font-size: 12px;
  padding: 0px 9px;
  vertical-align: middle;
  text-decoration: none;
  a :hover {
    color: #2D6DA8;
  }
}

.comments {
  .paginator {
    margin: 0.5em;
    .more-comments {
```

```
      color: #A5A7AA;
    }
  }
}

.comment-submit-button {
  position: absolute;
  left: -9999px;
}

.profile-header {
  padding: 20px 0;
}

.profile-image {
  margin: 20px auto;
  height: 152px;
  width: 152px;
}

.user-name-and-follow {
  display: inline;
}

.profile-user-name {
  display: inline;
}

.edit-button {
  border-color: #818488;
  color: #818488;
  margin-left: 20px;
}

.profile-bio {
  margin-top: 20px;
}

#user_bio {
  border: 1px solid gray;
}

#user_avatar {
  padding: 1em;
  margin: 0px auto;
```

```
}

.likes {
  margin-top: 20px;
  margin-bottom: 7px;
}
```

All I've done is added some styling for the 'likes' class and also tweaked how the 'user-name' class was presented.

Refresh your browser once more and it should look a little nicer.




Now, let's discuss what we need to build next:

- In order to see our new 'likes' for a post, we have to refresh the whole page

after clicking the little love heart. This is terrible.
- We have no indication whether our 'like' on a post worked or not. The little heart doesn't turn solid and the likes list isn't updated.
- We can't 'un-like' a post by clicking the button again.
- Each new like will just add another name to the list of likers. The majority of our screen real-estate will simply be names of likers in huge lists. After a certain point, we just want to show '212 likes', rather than list each of the 212 names.
- Some of our tests are failing now due to some dodgy capybara selectors.

But fear not, brave reader! We will be solving these issues and much, much more (well, maybe not too much) now!

## Solid Red, An Ego Fed

Let's start small, let's make sure that the heart turns solid red upon successfully liking Larry's legs. In Step One of this feature, we made sure that clicking the button does in fact 'like' the post, the problem is that it doesn't actually show it. The 'like' action in the PostsController can return javascript thanks to this code:

```
respond_to do |format|
  format.html { redirect_to :back }
  format.js
end
```

So let's start writing some javascript in the form of jQuery to get something happening! Create a new file within the `app/views/posts` folder and call it 'like.js.erb'. Our jQuery will be held here. Within that file, type the following:

```
$("#like_<%= @post.id %>").removeClass('glyphicon-heart-
empty').addClass('glyphicon-heart');
```

This is finding the specific post we've liked by it's unique ID and we're removing the empty heart glyphicon and replacing it with the regular version. Simple!

Now, something that you will notice at this point is that if you've previously liked a post, it'll still be showing up as an empty heart. This is because in our `_post.html.haml` partial, we've hardcoded that glyphicon in like so:

```
=link_to '', like_post_path(post.id), remote: true,
                                                id: "like_#
{post.id}",
                                                class: "glyphicon
glyphicon-heart-empty"
```

Let's adjust it so it'll check if we've liked the post and adjust the class name appropriately.

First, change the above code to this:

```
=link_to '', like_post_path(post.id), remote: true,
                                                id: "like_#
{post.id}",
                                                class: "glyphicon
#{liked_post post}"
```

The `liked_post` helper method will determine whether we've liked the post and will then return the appropriate class. Open up the `app/helpers/posts_helper.rb` file and add the following method (make sure it's *not* a private method):

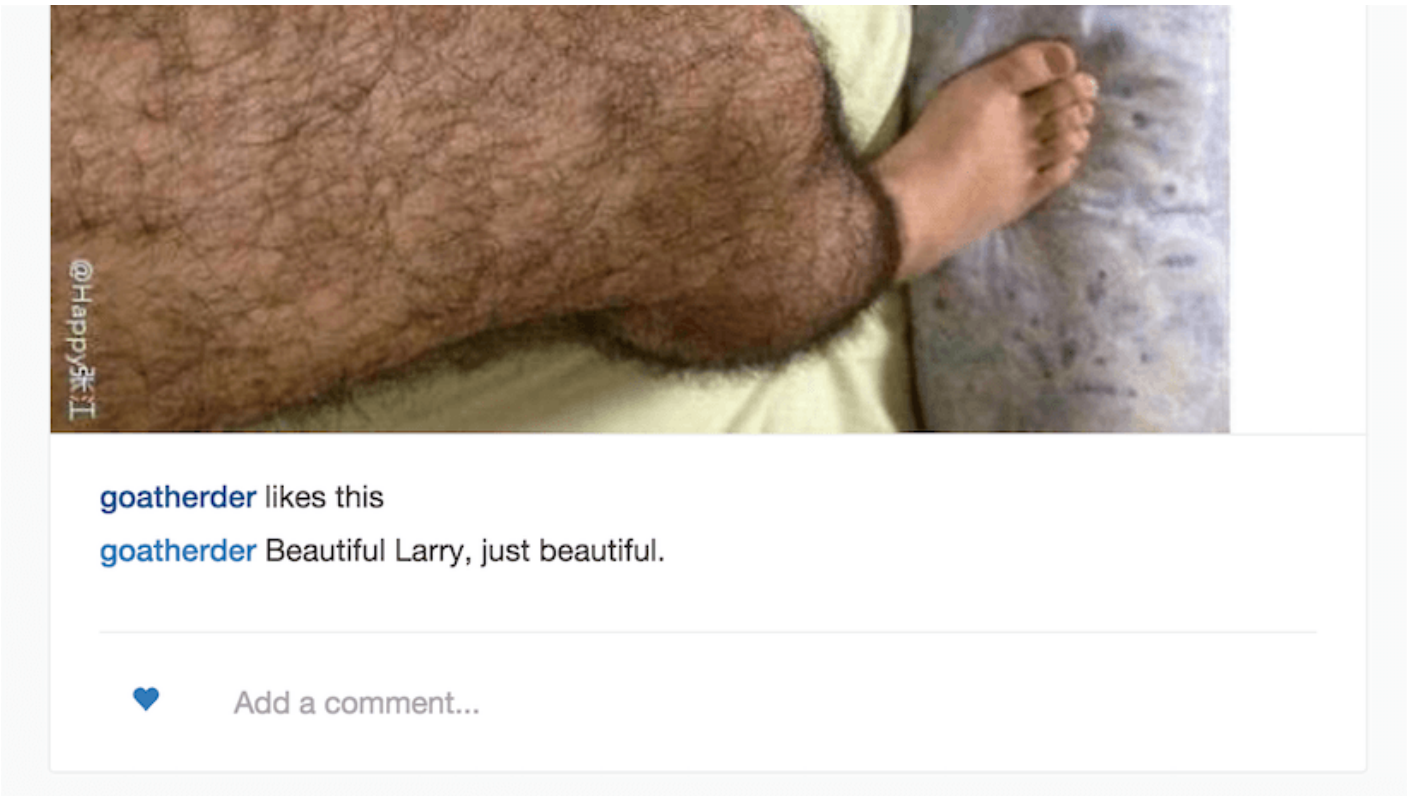
```
def liked_post(post)
  return 'glyphicon-heart' if current_user.voted_for? post
  'glyphicon-heart-empty'
end
```

Last but not least, in order to get the `voted_for?` method in the above code working, you'll need to add `acts_as_voter` to your `app/models/user.rb` file. Here's how my user.rb file looks:

```
class User < ActiveRecord::Base
  acts_as_voter
  validates :user_name, presence: true, length: { minimum: 4,
maximum: 12 }

  devise :database_authenticatable, :registerable,
         :recoverable, :rememberable, :trackable, :validatable

  has_many :posts, dependent: :destroy
  has_many :comments, dependent: :destroy

  has_attached_file :avatar, styles: { medium: '152x152#' }
  validates_attachment_content_type :avatar, content_type:
/\Aimage\/.*\Z/
end
```

This is explained in the acts_as_votable docs and is just giving us some handy methods to add to our toolbox when dealing with our likes.

So, refresh your dashboard now and you should be greeted with some nice, solid (albeit *blue*) hearts.

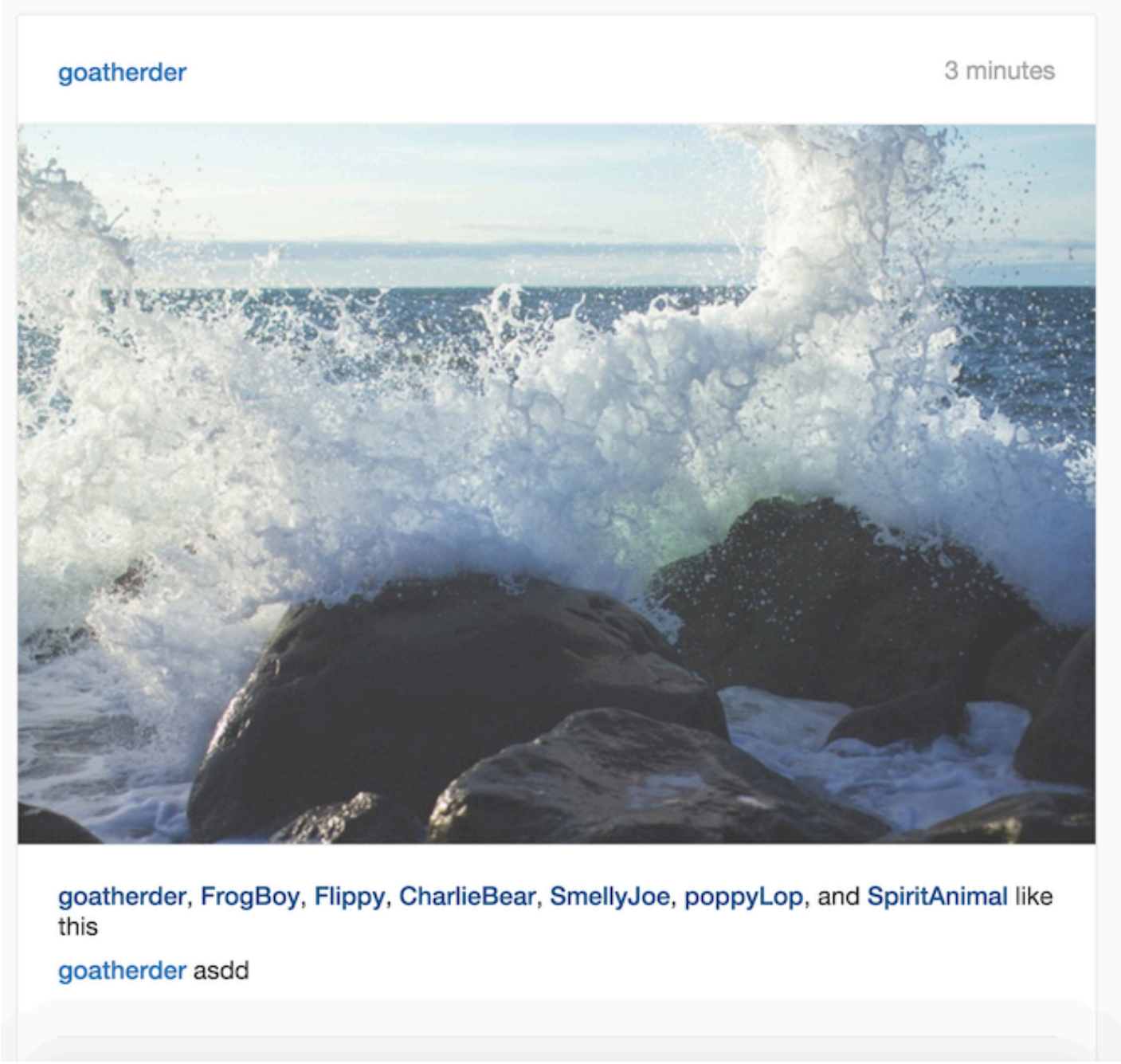



Also, if you like a post that you haven't previously liked now, you'll get a lovely transition from empty to solid love-heart. We're getting there!

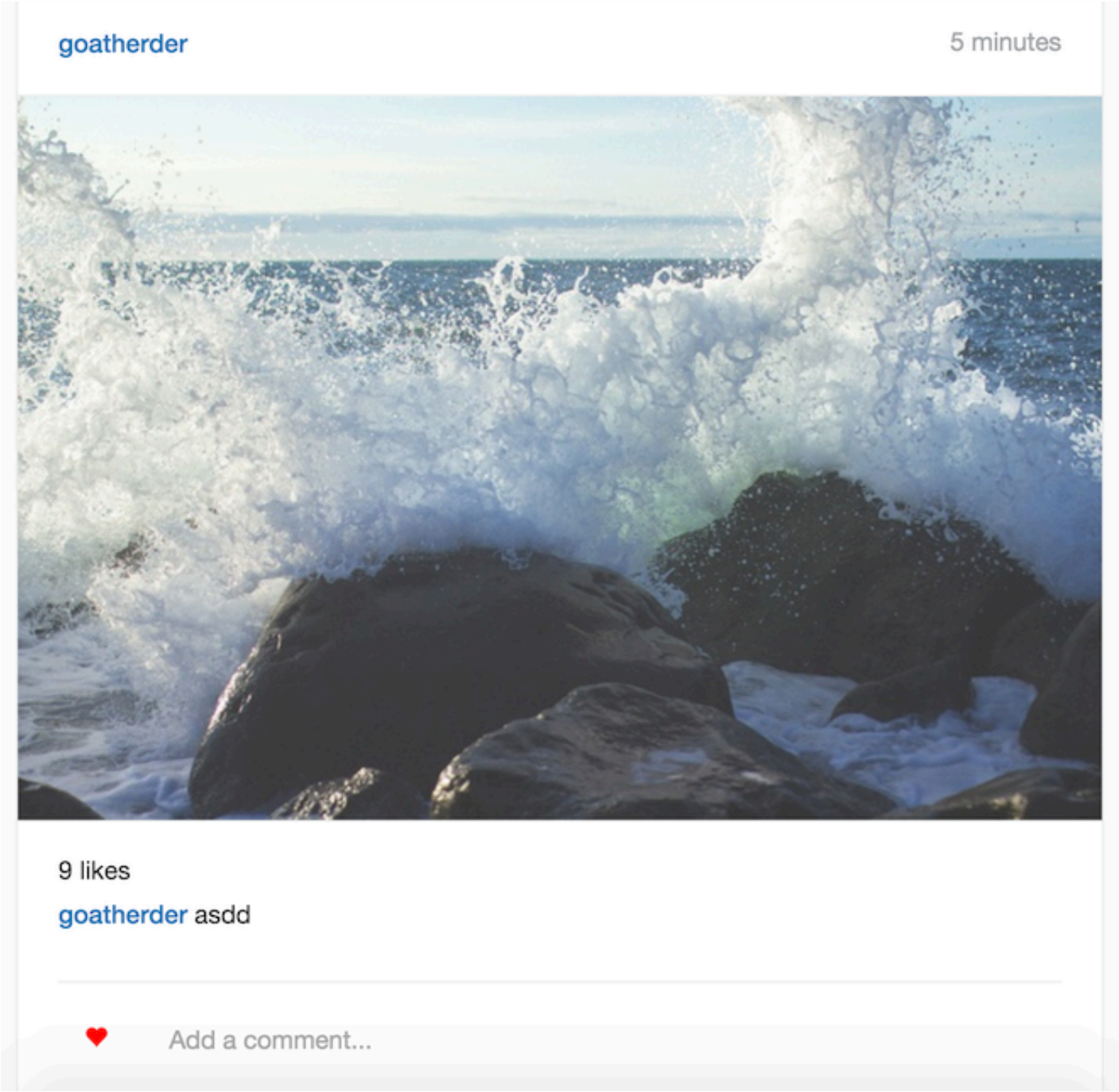
## Displaying Likes in Two Ways

As mentioned earlier in the article, Instagram displays their like counts in two ways.

When there's only a few likes, list all the names under the image:




When there's lots of likes, just list the like count:




We'll do the same. At the moment, we're listing out the names with our helper methods BUT we're also listing the names to infinity, meaning that the names will continue forever, making our application look ridiculous. Let's fix it.

First, let's display a like count for a post if it has more than 8 seperate likes. First though, try it yourself! Think about it like this, for the sake of your helper methods:

1. Your view calls a `display_likes` helper method (yet this is a different name than we used in step one).
2. If the post has less than or equal to 8 likes, call our previous `list_likers` method.
3. If the post has more than 8 likes, call a new method, `count_likers`, which

will display the like count, followed with 'likes'.

Instagram will make that like count a clickable list of likers but we're not too worried about that for the sake of this tutorial (at this stage).


---

Here's how I tidied up my Posts Helper and created some extra logic for displaying the two different forms of 'likers'.  I'll post the whole helper file here for the sake of context:

```ruby
module PostsHelper
  def display_likes(post)
    votes = post.votes_for.up.by_type(User)
    return list_likers(votes) if votes.size <= 8
    count_likers(votes)
  end

  def liked_post(post)
    return 'glyphicon-heart' if current_user.voted_for? post
    'glyphicon-heart-empty'
  end

  private

  def list_likers(votes)
    user_names = []
    unless votes.blank?
      votes.voters.each do |voter|
        user_names.push(link_to voter.user_name,
                                profile_path(voter.user_name),
                                class: 'user-name')
      end
      user_names.to_sentence.html_safe + like_plural(votes)
    end
  end

  def count_likers(votes)
    vote_count = votes.size
    vote_count.to_s + ' likes'
  end

  def like_plural(votes)
    return ' like this' if votes.count > 1
    ' likes this'
  end
end
```

There HAVE been some method name changes in the revision above for the sake of clarity so please take note of that. But really, the logic is as simple as I mention above the goat. Check for the like count, call the appropriate method. Beautiful!

## Instantly Adding our Like

So, now that clicking the like button makes the like heart turn solid red, we now want our name added to the list of likers instantly or to have the 'like count' incremented by one.

Let's create this feature now.

We'll have to think of three scenarios:

1. There are currently no likes on the post.
2. There are currently 8 or less likes on the post.
3. There are currently more than 8 likes on the post.

We'll have to cater for each to make sure the experience is lovely for our users. But wait, what if we could just re-render that whole partial 'likes' area with our updated information? That would then call our helper method and take care of all of that for us!

Open up the `app/views/posts/like.js.erb` file you created earlier and add the following code. I'll explain how it works afterwards.

```
$("#like_<%= @post.id %>").removeClass('glyphicon-heart-
empty').addClass('glyphicon-heart');
$("#likes_<%= @post.id %>").html("<%= j (render partial:
'posts/likes', locals: { post: @post} ) %>");
```

First things first, do you notice the reference to the div id `#likes_<%= @post.id %>`? Well, you're going to have to add it to your views.

Update your `app/views/posts/_likes.html.haml` view as per below:

```
.likes{id: "likes_#{post.id}"}
  = display_likes post
```

Remember, all this does is give us a *very specific location* for javascript to do it's thang. Each `post.id` is unique, so we know that jQuery will always be looking at the right post when it updates.

Now, in the jQuery above, all we're doing is this:

- Finding the appropriate post 'likes' id element.
- Changing the html of that element to the rendered `_likes.html.haml` partial WITH the new information it'll have from our AJAX call.

To summarise: Our user-name will be added to the list!

Now, this isn't an ideal solution for this problem for one simple reason: We're making extra calls to the server every time we like a post (or dislike a post). How else could we achieve the same effect?

Just create a facade of adding names to the list with javascript. Rather than re-render that whole element with the partial every single time, just have some javascript running that will append the `current_user`'s user name to the likers list.

It'd be much more efficient and I recommend you try to create it yourself.

## Unliking Likes

I'm going to leave this to you. No, seriously I am. Want me to give you some answers below the goat? Nope. It's all on you.

I will give you some really great hints though:

- Could we expand our `liked_post` helper method to actually include the `link_to` helper as well? That way, if the user has already liked a post, we could change the link to point to the unlike_post_path, rather than the like_post_path.
- What if we added some extra logic to the javascript file, so that not only will clicking on our new link adjust the class of our little heart, but also changed the link itself?
- What would the logic in a `unlike.js.erb` file look like? Probably pretty similar to that found in the `like.js.erb` file, no?
- What would the new `unlike` action look like in the posts controller? Pretty similar to the `like` action I imagine, just using some different `acts_as_votable` methods...

And that's it. Some simple steps to making a brand new `unlike` feature. Go forth and create something cool using what you've learnt so far by building the likes. I assure you it's very, very similar and you're awesome enough to do it.

## Hearts are Red

Now lets very simply style our little heart so that it looks a little nicer for our users (they're very picky). At the moment, our heart is the default blue colour found in our app for all links, and we also get an ugly little underline on the heart when hovering with our mouse.

Jump on into your `app/assets/stylesheets/application.scss` file and add the following code to the bottom:

```
.like {
  text-decoration: none;
  color: red;
}

.like-button a:hover, a:focus {
  text-decoration: none;
  color: red;
}
```

You'll have to add the `like-button` class to your `app/views/posts/_post.html.haml` file like so:

```
.comment-like-form.row
  .col-sm-1.like-button
    =link_to '', like_post_path(post.id), remote: true,
                                          id: "like_#{post.id}",
                                          class: "like glyphicon
#{liked_post post}"
```

And the little love heart will now look a little better. Does it look Instagram good? No, no it doesn't. But it looks better than it did!

## The End of Likes

So we've now added another big feature to Photogram, liking each other's posts. We've only got a little more functionality to add at this point, notifications and creating the follower / following relationships.

Another question that has been asked is this, "Should we just use a javascript front-end framework for an application like this, where we're playing with javascript a lot?". And the answer is almost certainly yes.

Instagram itself uses ReactJS for the front-end, so why bother using rails and jQuery?

Because it's great to see what can be done with Rails itself and it helps piece together the Rails puzzle. The javascript in the series so far hasn't been *too* awful and messy so for a project of this scope, so in the grand scheme of things it's

probably fine.

Until next chapter my friend, stay frosty and know that you're awesome.

# Chapter 6 - Ninety Nine Nice Notifications

## I'm Living the Dream

As a result of my incredible life, millions of dollars and luxury lifestyle, a HUGE amount of attractive people are commenting on my Photogram posts and I want to be notified of it!

If I don't have any knowledge of people attempting to be *social* with me, what the hell's the point of this *social network?*

**I have no idea.**

Luckily for me, I have developed a very particular set of skills. Skills I have acquired over a very long career. Skills that make me a nightmare for features like this.

Let's build a notifications feature in Photogram (or any Rails app at all).

## What will this feature look like?

This feature is going to be more Facebook than Instagram in that we're going to have a little icon in our top navbar that we can click, giving us a small paginated list of notifications. Once we've had a glance at the notifications, we can either click one to be taken to the appropriate comment *or* we can click a 'View all notifications' link that will take us to a separate page with a complete list of notifications.

Also, we want to make sure that our notifications can be classed as **read** or **unread**. We don't want to annoy our users with old notifications, that would be the worst.

Now, why would a user be notified? There are a few scenarios:

- Someone comments on one of our rad photos.
- Someone likes one of our posts.

We could also create a feature where we get notified if someone makes a comment in a thread we've also commented in but this is one of my least favourite Facebook notification features, so let's not encourage it any further. If you want to challenge yourself, try to build it yourself. I'd recommend using a Subscription

model and creating a many to many relationship between Users & Posts *through* subscriptions.

Sounds good, time to get nerdy.

## The Notification Model

First, let's generate a model for our notifications. We're going to want to reference a few things in our model:

- The user who's made the new comment or like.
- The user who owns the post that has been affected.
- The post which the user is commenting or liking.
- The id of the comment.
- The type of notification it is, a comment or like.
- Whether or not the notification has been read.

I believe in you. Yes you. I think that you are wonderful enough to generate this model by yourself in the terminal. Go forth and generate (but don't migrate just yet).


---

How'd you go? Of course you went great! Here's how I'd do it (your way was probably better). I just tapped the following away in my terminal:

```
rails g model Notification user:references
subscribed_user:references post:references identifier:integer
type:string read:boolean
```

Once that's complete, make sure your migration file looks something like the following:

```
class CreateNotifications < ActiveRecord::Migration
  def change
    create_table :notifications do |t|
      t.references :user, index: true
      t.references :notified_by, index: true
      t.references :post, index: true
      t.integer :identifier
      t.string :notice_type
      t.boolean :read, default: false

      t.timestamps null: false
    end
    add_foreign_key :notifications, :users
    add_foreign_key :notifications, :users, column:
:notified_by_id
    add_foreign_key :notifications, :posts
  end
end
```

Once you've perused the above and you're happy that your migration matches, you can jump back in your terminal and migrate.

```
rake db:migrate
```

Nice work!

Let's hook up the internals of our models now by stating the relationships in the model classes. Open the `app/models/notification.rb` file and add the User class name to the `notified_by` line.

```
belongs_to :notified_by, class_name: 'User'
belongs_to :user
belongs_to :post
```

Alright, so each notification is referencing a `notified_by` user in the User model, a regular old `user` (who we'll attach the notification to) and a regular old `post`. Let's quickly add a validation to the model as well.

```
validates :user_id, :notified_by_id, :post_id, :identifier,
:notice_type, presence: true
```

Now it's time to sort out our other two affected models. First, let's adjust the User model. Add this line to the relationships area.

```
has_many :notifications, dependent: :destroy
```

And the same to our Post model.

```
has_many :notifications, dependent: :destroy
```

Now that we have our model sorted, you'd think that we'd generate a controller but first, let's see how we'll slot our notifications into our current system.

Open up your comments controller at `app/controllers/comments_controller.rb`. Pan your eyes to the create action and check out the current flow. We build a comment based on the comment params and assign a user id. If that comment saves, we go ahead and respond with some javascript that updates the user's page. Let's add another line to create a notification.

Underneath the `if @comment.save` line, add the following:

```
if @comment.save
  create_notification @post, @comment
  respond_to do |format|
    format.html { redirect_to root_path }
    format.js
  end
```

Now that `create_notification` method doesn't exist yet, so underneath the `private` line in your controller (we don't want this method called outside this controller), add our new method.

Give it a go yourself now. We're going to have method that takes the post as an argument. In that method, all we're we're doing is creating a new notification object, by assigning it's attributes from the post argument and the current_user. Also, make sure a user *isn't* notified if they're commenting on their own post. That'd be annoying.

Good luck!


---

Here's my brand new `create_notification` method:

```
def create_notification(post)
    return if post.user.id == current_user.id
    Notification.create(user_id: post.user.id,
                        notified_by_id: current_user.id,
                        post_id: post.id,
                        comment_id: comment.id,
                        notice_type: 'comment')
end
```

How easy is that!

Let's test it now.  Get your Photogram server running, log in and comment on an existing post, making sure the logged in user and the creator of the post aren't the same user.  Once you've done that, jump into your terminal and make sure that the notifications are being created.

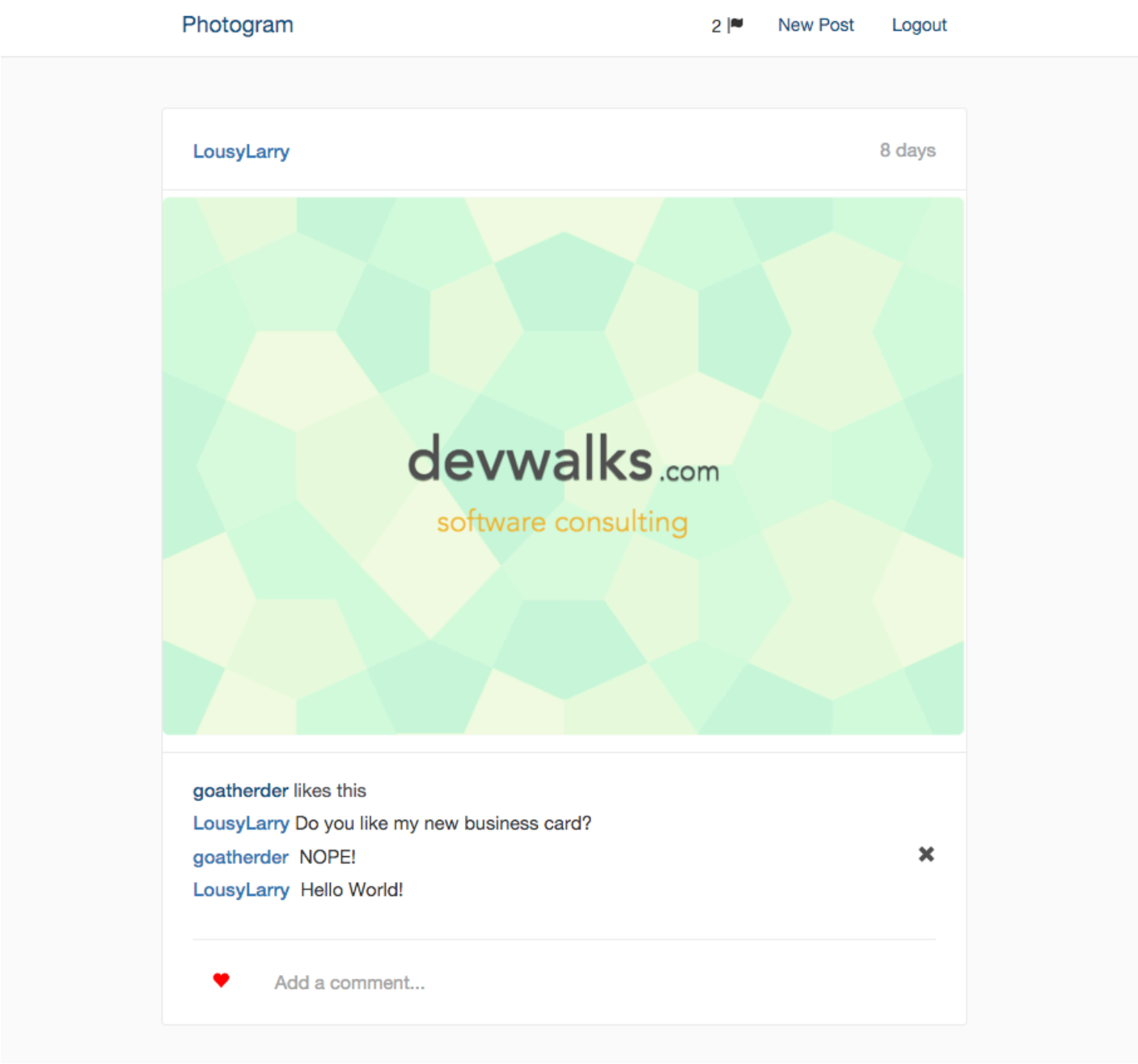
In your terminal:

```
rails c #to run the console
n = Notification.last # The notification should appear!
user = User.where(email: 'ben@devwalks.com').first # How I found
my user who should be notified
user.notifications # Should list out all the appropriate
notifications!
```

Hopefully you're starting to get an idea on how we'll implement this feature in our UI now that you can see how easy it is to access notifications associated to a specific user.

Let's start building out the UI for this feature now.

## Showing Notifications




We're going to add a preview of our available notifications to our horizontal navbar at the top of the page, as per Facebook. Open the navbar partial file at `app/views/layouts/_navbar.html.haml` and add the following underneath the `user_signed_in?` helper:

```
- if user_signed_in?
  %li
    .dropdown.notification-dropdown
      %button.btn.btn-default.dropdown-toggle{ type: 'button',
id: 'dropdownMenu1', data: { toggle: 'dropdown' }, 'aria-
haspopup': true, 'aria-expanded': true }
        5
        %span.glyphicon.glyphicon-flag
      %ul.dropdown-menu{ 'aria-labelledby': 'dropdownMenu1' }
        %li.dropdown-header.text-center Notifications
        %li
          %a{ href: '#' } Notification 1
        %li
          %a{ href: '#' } Notification 2
        %li
          = link_to "New Post", new_post_path
```

Reload Photogram and you'll see a pretty basic demo of what we're going to have at the end of the day. Fix up the styling quickly by adding the following SCSS to your `app/assets/stylesheets/application.scss` file:

```
.notification-dropdown {
  margin-top: 10px;
  button {
    border: none;
  }
}
```

Now, rather than pollute our navbar with extra code, let's move our notifications dropdown box into its own partial. Cut everything notification related under the list tag and paste it into a new file under `views/notifications` (you'll have to create the folder) called `_notificiation_dropdown.html.haml`. Now, add the render method to the list tag like so:

```
- if user_signed_in?
  %li
    = render 'notifications/notification_dropdown'
  %li
    = link_to "New Post", new_post_path
```

Alright. Time to get dynamic.

## Adding Real Data & Notifications

First, let's change our placeholder '5' in our notifications dropdown with the actually count of ALL of the current user's notifications (we'll change this in future to show only the unread notifications).

Change the line with the lone '5' to instead show the following:

```
= current_user.notifications.count
```

Refresh your Photogram feed and at the very least, the number will have changed. If, like me, you're showing 0 notifications, log in as a different user in a private browser and make a few comments on the posts belonging to the original user.

Head on back to the original browser window and refresh. If everything worked as it should've you'll be blessed with a number higher than 0.

How good is this? We're accessing our notifications just as we should. Let's now list ALL of the notifications we have for the current user.

And I want you to try first. Here's how you'll have to do it.

- Create a new partial called `_notification.html.haml` in your `views/notifications` folder. This partial will be in control of each `<li>` item in the drop-down menu.
- Loop through your notifications in the `_notification_dropdown.html.haml` partial. You'll have to create a `_notification.html.haml` partial that will act as the placeholder for each individual notification.
- You'll want each notification to reference the user name of the user who created the notification and the type of notification (comment or like).
- Use a `link_to` helper and actually link each notification to the appropriate post.

Good luck my friend, I believe in you.


---

Alright!  Here's how I achieved our simple notification system.

Here's my complete `_notification_dropdown.html.haml` file:

```
.dropdown.notification-dropdown
  %button.btn.btn-default.dropdown-toggle{ type: 'button', id:
'dropdownMenu1', data: { toggle: 'dropdown' }, 'aria-haspopup':
true, 'aria-expanded': true }
    = current_user.notifications.count
    %span.glyphicon.glyphicon-flag
  %ul.dropdown-menu{ 'aria-labelledby': 'dropdownMenu1' }
    %li.dropdown-header.text-center Notifications
    = render current_user.notifications.order('created_at DESC')
```

Notice I've just replaced our `<li>` tags with the render method.

I then created the `_notification.html.haml` partial I mentioned above.

```
%li
  = link_to "#{notification.notified_by.user_name} has #
{notification.notice_type}ed on your post",
post_path(notification.post)
```

It's pretty simple, really.  In our string, we access details about the notification, hack together a plural version of 'comment' and link to the `post_path` of the appropriate post.

Give it a go now and implement it into your own application.  Click around and make sure it's all working as you'd expect.

## Marking notifications as read

I'm super happy that I can now navigate to posts that I have notifications for, but my notifications are now plagued with old notifications I don't care about any more! How can we mark a notification as 'read' after we've navigated to the post?

Here's how:

Create a notification controller with a `link_through` action via the Rails generator in your terminal.

```
rails g controller Notifications link_through
```

Open up your new `app/controllers/notifications_controller.rb` and adjust the `link_through` action as below:

```
def link_through
  @notification = Notification.find(params[:id])
  @notification.update read: true
  redirect_to post_path @notification.post
end
```

We're retrieving a notification via the `:id` url param (more on this next) and then setting the `read` attribute to true. Once this is complete we redirect the user to the appropriate post.

Now, how do we get our `:id` param? We'll add it to the route for this action. A route was generated for our action automatically but we'll adjust it now. Open up the `config/routes.rb` file and change the generated line:

```
get 'notifications/:id/link_through', to:
'notifications#link_through',
                                    as: :link_through
```

This gives us our `:id` param we need to find the appropriate notification and also gives us a handy name to use in our `link_to` helper methods in our views.

Last but not least, let's adjust our `link_to` helper method in our `_notification.html.haml` view so that it's linking to our new action, rather than the old `post_path`.

```
= link_to "#{notification.notified_by.user_name} has #
{notification.notice_type}ed on your post",
link_through_path(notification)
```

Lovely!

If we want to only show our unread notifications in our drop-down list, we can simply adjust our `render` call in the `app/views/notifications/_notification_dropdown.html.haml` file to:

```
= render current_user.notifications.where(read: false)
```

Don't forget to change the notification count line as well if you decide to go this way!

As for how you display your notifications, I'll leave it to you! Scary huh.

If you'd like to show both your read *and* unread notifications in the drop-down menu but style them differently (Facebook style), you can simply add a special class to the notifications that are read and style them appropriately! If you need more help with this, leave a comment below and I will happily guide you in the right direction.

## Pagination & viewing all notifications

Eventually, we're going to have *a lot* of notifications (don't be humble, you're super popular). This will be an issue with our current drop down box so let's quickly fix that now.

We can only show the last x amount of notifications by simply adding the `last` method to the end of our render helper method.

```
= render current_user.notifications.order('created_at
DESC').last(5)
```

Make that number whatever you want. BUT, now that we're only showing a limited amount, how do we view all of our notifications?

Let's handle it Facebook style, by creating a whole new 'index' page of our notifications.

At the bottom of your notifications drop-down view, add a couple more lines:

```
%li.divider{ role: 'separator' }
    %li.text-center= link_to 'View All', notifications_path
```

If you were to refresh your dashboard now, you'll get an error because we don't yet have a route called 'notifications'. Let's fix that now. Open the `routes.rb` file and add this line:

```
get 'notifications', to: 'notifications#index'
```

This alone will let us refresh our dashboard, BUT, if you try to click the 'View All' link now, you'll be faced with an error screen letting you know that there is no such action as `index` in the notifications controller.

Believe in yourself and fix that now. If you've been following along with the tutorial so far, we've built PLENTY of index pages just like this!

Make sure you add an instance variable within the index action that references ALL of the current users notifications (for the moment). While you're going, add an `index.html.haml` view to your `views/notifications/` folder and hook it up to your instance variable. List out all of the notifications (even if it looks terrible, just get the data into the view as a big old list).

Pro-tip: Don't forget to use our `_notification.html.haml` partial file in your index view.


---

Alright, loyal reader! Let's compare solutions.

In my Notifications Controller, I added the following lines:

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

And promply added the below line to my brand new `views/notifications/index.html.haml` file:

```
= render @notifications
```

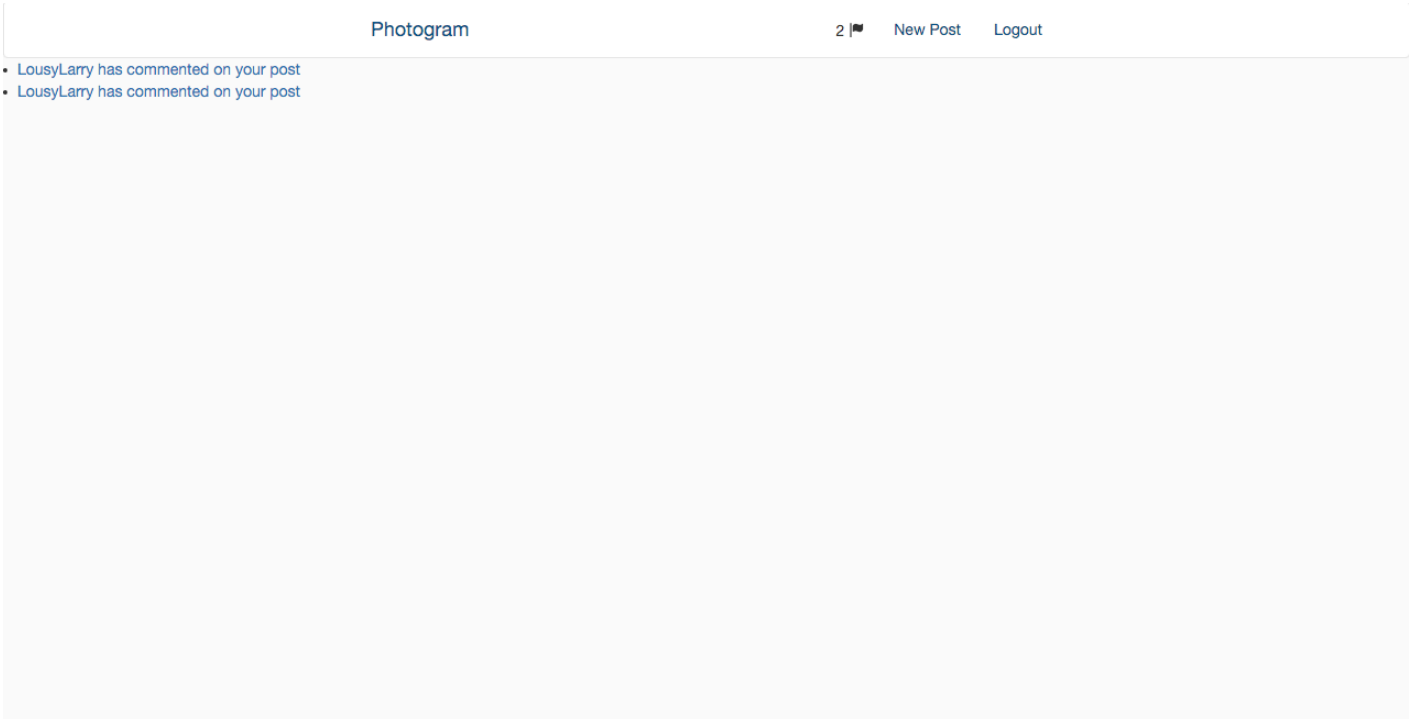
The above line is just a bit of Rails magic that does the exact same thing as this:

```
- @notifications.each do |notification|
  = render 'notification', notification: notification
```

Which version looks nicer? I'll leave that to you to decide.

Navigate to your brand new notification index page via the top dropdown menu and have a quick look.

**Not great:**




Let's make it look a little better.

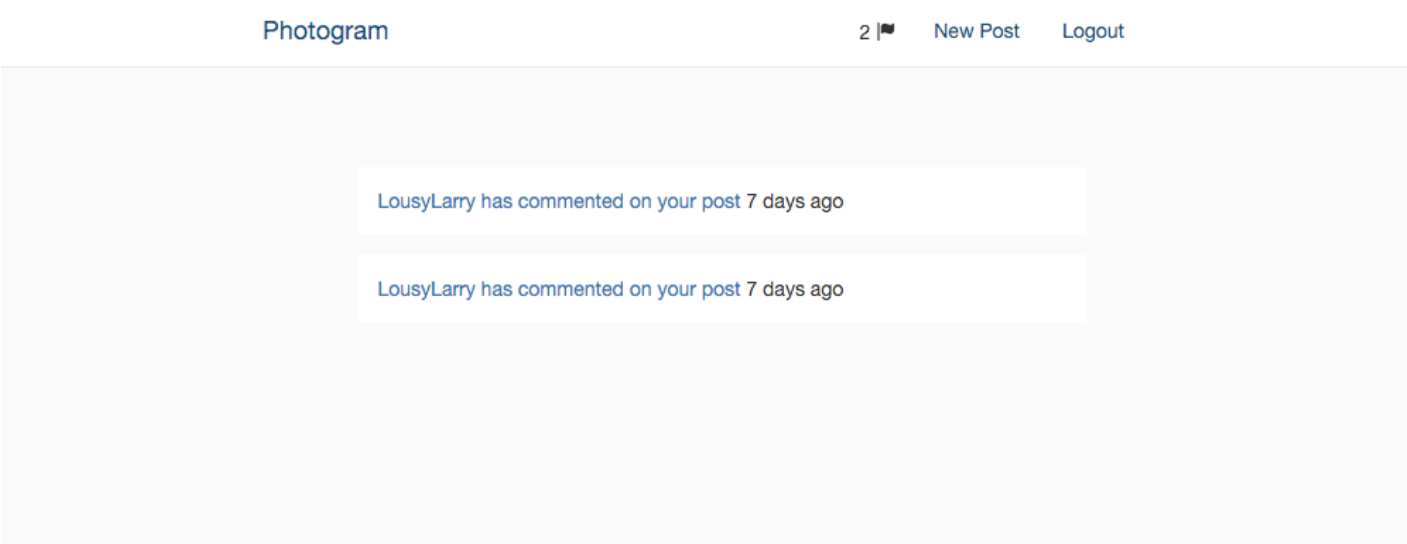
Add some styling to the index view:

```
.container
  .row
    .col-md-6.col-md-offset-3
      %ul.notification-index-list
        = render @notifications
```

And now we'll attack that `notification-index-list` class in our scss file.

```
.notification-index-list {
  margin-top: 5em;
  list-style: none;
  li {
    margin-top: 1em;
    background-color: white;
    padding: 1em;
  }
}
```

If you refresh the page now, it'll look *slightly* better.




And so we have it!  A functional notifications system in Rails.

What would I recommend you do for homework?

- Paginate the notifications in the index view.  You've done it plenty of times before!
- It'd be great to know *when* someone commented on our post.  We could easily add this with the Rails `time_ago_in_words` helper method.
- Perhaps create a new partial for our index page that makes it look a little

nicer. There probably isn't enough in common for the drop-down and index page to share the same partial view.
- How good would it be to mark our notifications as read and unread in our notifications dropdown AND index? I imagine that feature would looks very similar to how we add or remove comments in Photogram....
- Divide our index page into read and unread. You could do this by styling the read and unread items differently or completely separate them into different pages.
- Add notifications for likes! Think about how you did it for comments and just apply the same logic to the 'like' action in the posts controller! There will be some differences though with regards to the `notice_type` and `identifier` parameters of the new Notification.

## It's Great to Build Non-CRUD Rails Features

I even love writing about it!

In the last chapter of this book, let's build a *very* important feature of our social network, the following / follower relationship.

# Chapter 7 - Following Fancy Friends

## Our social network is *very* social

So social, in fact, that every single user follows (and is followed) by everyone else! What a magical social network to be a part of.

**Not everyone is happy though.**

There are grumblings in the Photogram community that users want to *pick and choose* who they follow and therefore who appears in their feed! It might just be crazy enough to work...

## A Brief History of Following

The model relationships that we're going to discuss in this guide can be tricky to understand at first, just because it brings forward what I found to be an awkward way of thinking. Join tables.

Sure, *now* it makes sense to me, but I remember feeling quite overwhelmed the first time I was exposed to this idea and it's possible that you might too. You could also be a genius, in which case, good for you smarty pants!

Check out the Rails Active Record Associations Guide that runs through some of the associations you'll use often. Pay special attention to those parts referencing the `has_many, :through` relationships because that's what we'll be dealing with today.

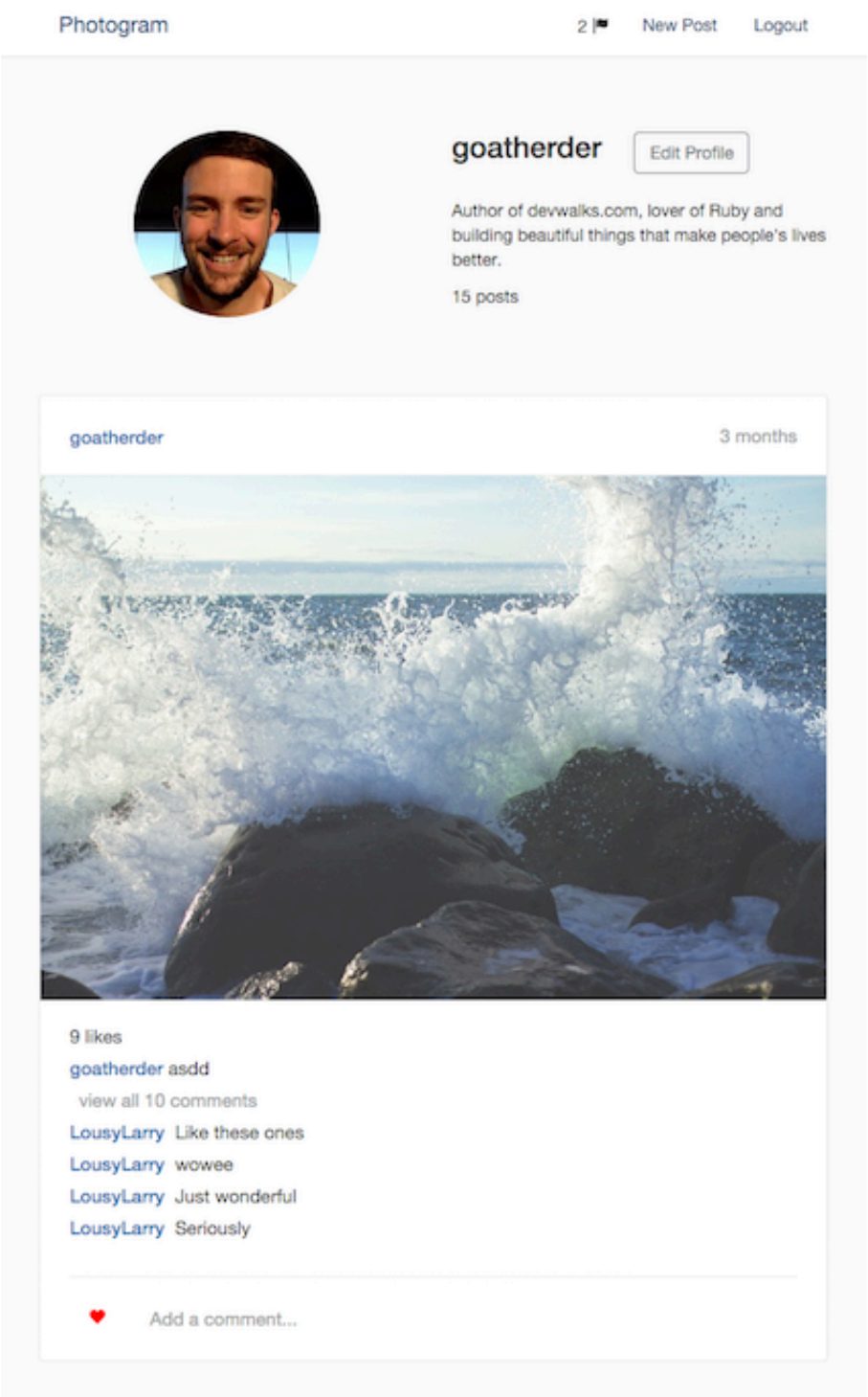
Please note that we're also not going to easily be able to *browse* the whole user base in this guide in order to see who we want and don't want to follow. What we'll do is, list ALL of the posts on Photogram in one dashboard (similar to Instagram's browse feature and exactly what we've been doing up until this point) and have our main news-feed only show the users that we follow.

## Let's Build It!

All talk and no play makes you, the reader, feel not so gay. So let's go forth and build our functionality. We can discuss *theory* more as we go.

## Here's How it's Works

At the moment, our user's profiles look like this:




We can't actually follow or unfollow a user from here, all we can do is gaze upon their wonderful selfies.

Where the 'Edit Profile' button is in the above screenshot (that's my own profile), we'll put a FOLLOW or FOLLOWING button, depending on our existing relationship. On clicking the 'FOLLOW' button, we'll use an AJAX call to a `follow_user` controller action and the new relationship will be created. Upon success, we'll use jQuery to transform the 'FOLLOW' button into a green 'FOLLOWING' button (we'll also change the link at this point to point to

an `unfollow_user` action, just in case our user made a terible mistake and followed a My Little Pony enthusiast).

We'll also adjust our dashboard logic so we're only showing posts from the users we're following and also create a brand new dashboard that emulates our current system of showing EVERYTHING (so we can find new users to follow).

Phew, that's quite a lot to build, let's not waste any more time my friend, greatness awaits us.

## First, the Models & Relationships

It's nice to get the models and relationships set up first. Once these are set up we can easily test relationships in the Rails console that will then convert directly into beautiful logic in our application!

Let's go over what our model relationships will look like:

- A user `has_many` users that they're following (those users whose posts will show up on your primary dashboard).
- A user `has_many` followers who are also users (the followers will see the users posts on their primary dashboard).

*Easy enough right?*

Sure is! The complexity is in how you model this data in the background. Think about it like this. Imagine we *just* had the user model. How would we refer to both the following users and the user's followers (think in terms of referencing id's in the database columns)?

You couldn't have a separate column for each relationship, that'd be madness. That's where a *join* table comes in handy. The join table simple references a model to another model, or in this case, a model back to itself. Let's imagine it this way:

There's a record for a user, whose `id` is 1. The user is following users with the id's of 4, 7 and 12. By creating a separate join table, we can reference a relationship between these users. The join table can have two columns, `following_id`, `follower_id`.

The `follower_id` in the above scenario would be 1 for all three records and the `following_id` would be 4, 7 and 12 respectively. Once the relationships are setup nicely in Rails, we could then simply refer to those users we're following with:

```
@current_user.following
```

In the opposite direction, imagine that the user with the `id` of 1 is followed by users 2, 3 and 10. The `following_id` is 1 in each of the three records and the `follower_id` would be 2, 3, and 10 respectively. To retrieve the records of the user's followers:

```
@current_user.followers
```

If this hasn't sunk in yet, do not fear. It may take time. As I've mentioned previously, this sort of relationship made me *quite* uncomfortable for an extended period of time. Hopefully your brain is better aligned and it'll click for you sooner! Perservere and you'll be ok in the end I promise (I make no actual legally-binding promises).

## Making things happen

Let's generate some migrations for the above relationships. Jump into your terminal and tap away the following:

```
rails g migration CreateFollowJoinTable
```

Once complete, pop into the migrations folder in your project and open up your newly created file.

What you want to do at this point is to create a table that has two integer columns, one called `following_id` and the other called `follower_id`. You also want to make sure that both columns have an index and that the combination of both columns in new records MUST be unique.

Call the new table 'follows'.

Give it a go now! Use your google-foo if you must.


---

Here's my complete migration file for comparison:

```
class CreateFollowJoinTable < ActiveRecord::Migration
  def change
    create_table 'follows' do |t|
      t.integer 'following_id', :null => false
      t.integer 'follower_id', :null => false

      t.timestamps null: false
    end

    add_index :follows, :following_id
    add_index :follows, :follower_id
    add_index :follows, [:following_id, :follower_id], unique:
true
  end
end
```

Starting from the top, we've called our table 'follows', and the created two integer columns that may not be false. We add some timestamps for good measure. Below this, we add an index for model columns and then make sure the combination of the two in a new record must be unique.

Go forth and run the new migration in your terminal:

```
rake db:migrate
```

Perfect! Give yourself a high-five and once complete, we'll finish of our setup.

First, we want a new model called `Follow`. This will reference our `follows` table implicitly and will let us create our user relationships.

In the terminal run:

```
rails g model Follow
```

Open up your new `app/models/follow.rb` file and edit it thusly:

```
class Follow < ActiveRecord::Base
  belongs_to :follower, foreign_key: 'follower_id', class_name:
'User'
  belongs_to :following, foreign_key: 'following_id',
class_name: 'User'
end
```

We're explicitly stating our relationships here, rather than the usual Rails way. We're stating the class that the foreign key is associated with and also giving it name of either 'follower' or 'following'. Finally, let's adjust our `User` model to suit.

Once open, you'll want the new relationships in your user model to look like:

```
  has_many :follower_relationships, foreign_key: :following_id,
class_name: 'Follow'
  has_many :followers, through: :follower_relationships, source:
:follower

  has_many :following_relationships, foreign_key: :follower_id,
class_name: 'Follow'
  has_many :following, through: :following_relationships,
source: :following
```

Alright...

Some pretty wild things are happening here so let's go through them.

Our first `has_many` line gives us the 'follows' relationship objects where the user is being followed. The next line goes the next step and accesses a user's `followers` *through* those relationships (we're accessing the actual user data at this point).

The second two lines is the same but for the 'following' relationships (those who the user is following, in order to see their posts).

It's one of those things that can be easier to understand by playing around in the console.

Make sure you've saved the above relationships in your `User` model and then open up your Rails console. Let's see how this all works.

*The below assumes you have users with the id of 1, 2 and 3. If you don't, just use what you have. You may have to create some new users, either manually or in the console.*

```
# create a new follow relationship where the user
# with the id of 1 follows the user with the id of 2
Follow.create(follower_id: 1, following_id: 2)
# create another for good measure with a different user
Follow.create(follower_id: 1, following_id: 3)
# Now, let's create a relationship where the user with
# the id of 1 is being followed by a user with the
# id of 3
Follow.create(follower_id: 3, following_id: 1)
# Now, let's see how our relationships work.  First,
# find the user who we created relationships for.
u = User.find 1
# List this user's followers
u.followers
# List the users this user is following
u.following
# calling follower_relationships on the user will list
# the relationships we've created in the follows table
# where the user is being followed by other users
u.follower_relationships
# calling following_relationships on the user will list
# the relationships we've created in the follows table
# where the user is following other users
u.following_relationships
```

If you're anything like me, the words 'following' and 'follower' have lost all meaning at this point and you probably need a coffee.

The point is...

The words and labels aren't important (words and labels are incredibly important in programming, but for the sake of understanding these relationships, it's not), it's how we're *linking* our data that's important. By creating relationships in the 'follows' table between two users, we can create the framework we need in order

to build our functionality.

We can make it even nicer to achieve the creation and destruction of relationships by adding some simple methods to our `User` model. It'd be great if we could simply call `current_user.follow user_id` or `current_user.unfollow user_id` within our controller. Let's write the simple methods to achieve that now.

```
def follow(user_id)
  following_relationships.create(following_id: user_id)
end

def unfollow(user_id)
  following_relationships.find_by(following_id: user_id).destroy
end
```

## That was some heavy stuff

And I'm sorry. Only time and practice will help you with these concepts.

Let's move on anyway and start building some relationships between our uses within the application itself.

First, let's create a 'follow' button that will sit in our user profiles. Within `app/views/profiles/show.html.haml`, add our new `-else` option:

```
%span
  - if @user == current_user
    = link_to 'Edit Profile',
edit_profile_path(@user.user_name),
                                class: 'btn edit-button'
  - else
    = link_to 'Following', unfollow_user_path,
                           remote: true,
                           class: 'btn unfollow-button',
                           id: 'unfollow-button',
                           method: :post
```

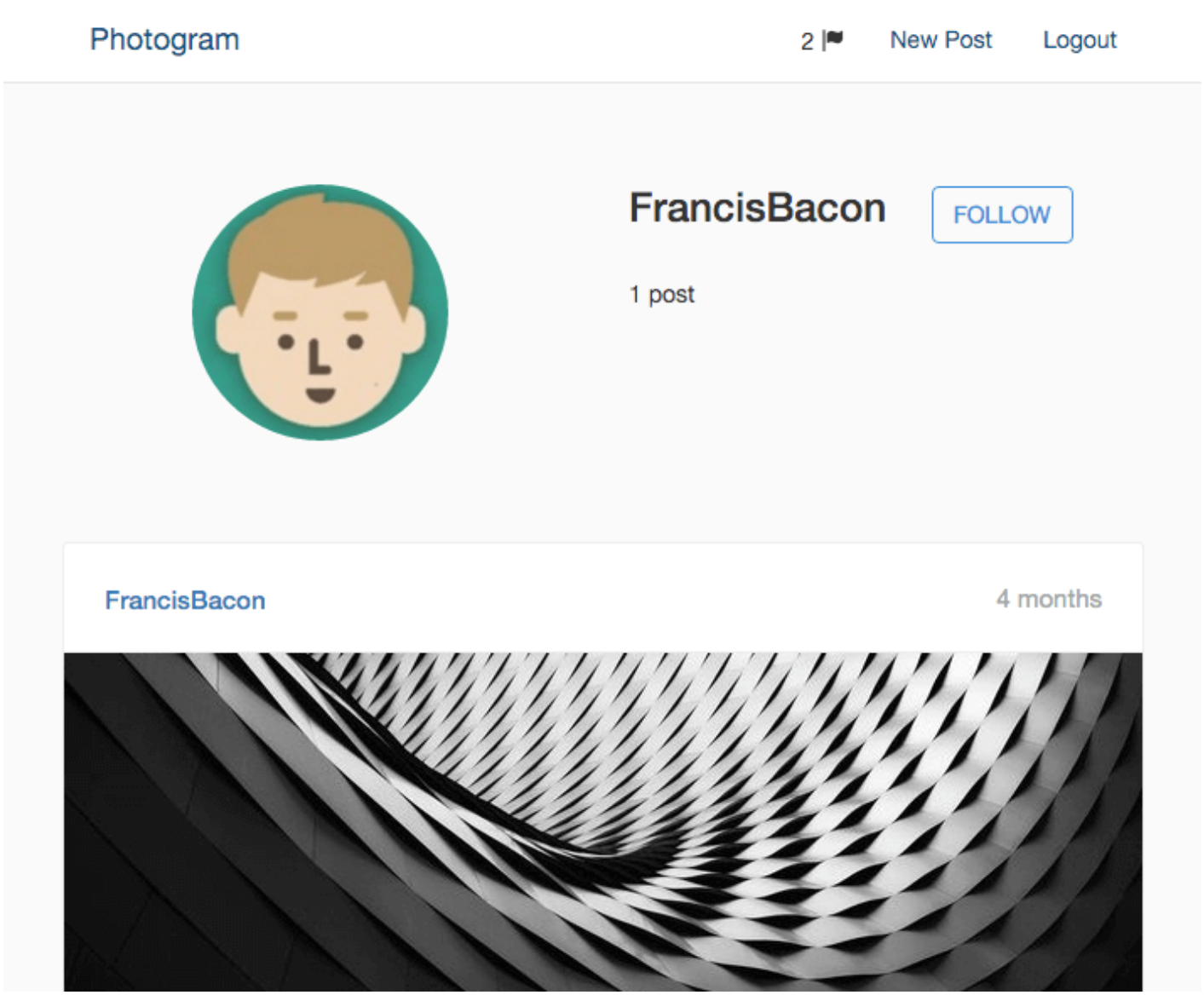
This is only temporary, we'll want to show an 'unfollow' button for those users who we're already following, but this will do for the moment (`follow_user_path` also doesn't exist yet but we'll fix that in a moment as well).

Next, we'll add some styling in our `app/assets/stylesheets/application.scss` file:

```
.follow-button {
  text-transform: uppercase;
  border-color: #4090db;
  color: #4090db;
  margin-left: 20px;
  &:visited, &:focus, &:hover {
    border-color: #2D6396;
    color: #2D6396;
  }
}

.unfollow-button {
  text-transform: uppercase;
  border-color: #66bd2b;
  background-color: #66bd2b;
  color: white;
  margin-left: 20px;
  &:visited, &:focus, &:hover {
    border-color: #47821F;
    background-color: #47821F;
    color: white;
  }
}
```

Note we added the styling for the 'unfollow' button as well (we'll add it soon enough).

Check it out!




Let's make it do something now.

First, we'll need to decide where we want to add the 'follow' and 'unfollow' functionality. It's tempting to use our existing `profiles_controller` but the functionality isn't *really* profile focussed. Let's separate the functionality into its own controller.

```
rails g controller Relationships follow_user unfollow_user
```

We could call this controller 'Following', 'Followers' or so on but relationships seems to make sense. Feel free to call it what makes sense to you.

Generating the controller with the above command, also creates some new routes for us. Quickly adjust them now to give them a name and add a reference to the user name of the user we want to follow or unfollow.

```
post ':user_name/follow_user', to: 'relationships#follow_user',
as: :follow_user
post ':user_name/unfollow_user', to:
'relationships#unfollow_user', as: :unfollow_user
```

Lovely! We've created ourselves a nice looking pair of routes that will do the job perfectly.

Time to add some logic to the `follow_user` and `unfollow_user` actions in the `relationships_controller.rb` file.

You should... give it a go yourself!

**Here's how to think about it:**

1. First, find the user we want to follow or unfollow via the `:user_name` parameter in the url.
2. Create a new relationships between the `current_user` and the user we found in step one using our new User methods.
3. If the method call is successful, respond with some javascript that will change the class of the button to 'btn unfollow-button', the text of the button to 'Unfollow' and the link of the button to either 'unfollow_user' or 'follow_user' depending on which action you're working on.
4. Go ahead and do the same for the `unfollow_user` action.
5. Make sure your profile view is showing the unfollow button for users you already follow and the follow button for users you don't yet follow.

**Here's why you can do this yourself:**

1. You've already implemented something similar to Step One in our previous Profiles article.
2. You've already played with building relationships earlier in this article when we were mucking about in the Rails console.
3. You've responded to saving data with javascript in the Comments AND Likes articles.

Basically, you already know what to do! Seriously though, give it a go yourself. The struggle will make you better. Be better and embrace the struggle. Struggle struggle struggle.


---

How'd you go? Struggle? Let's compare notes...

Here's my `RelationshipsController`:

```
class RelationshipsController < ApplicationController
  def follow_user
    @user = User.find_by! user_name: params[:user_name]
    if current_user.follow @user.id
      respond_to do |format|
        format.html { redirect_to root_path }
        format.js
      end
    end
  end

  def unfollow_user
    @user = User.find_by! user_name: params[:user_name]
    if current_user.unfollow @user.id
      respond_to do |format|
        format.html { redirect_to root_path }
        format.js
      end
    end
  end
end
```

Hopefully this makes sense to you at this point. We find the user who owns the profile and then assign them to the `@user` instance variable. If the `current_user.follow` method is successful, we then respond with some javascript (our link has the property `remote: true`).

We pretty much follow the exact same route for the `unfollow_user` action but call

the `current_user.unfollow` method instead.

Now that the controller actions are responding with javascript, we can create those new javascript responses in our `views/profiles` folder.

Here's `follow_user.js.erb`:

```
$('#follow-button').attr('class', 'btn unfollow-button')
                   .text('Following')
                   .attr('href', "/<%= @user.user_name
%>/unfollow_user")
                   .attr('id', 'unfollow-button');
```

And here's `unfollow_user.js.erb`:

```
$('#unfollow-button').text('Follow')
                     .attr('class', 'btn follow-button')
                     .attr('href', "/<%= @user.user_name
%>/follow_user")
                     .attr('id', 'follow-button');
```

Last but not least, let's add some logic to our profile view in `app/views/profiles/show.html.haml`. Make sure your view looks something like below:

```
%span
- if @user == current_user
  = link_to 'Edit Profile', edit_profile_path(@user.user_name),
                            class: 'btn edit-button'
- else
  - if current_user_is_following(current_user.id, @user.id)
    = link_to 'Following', unfollow_user_path,
                           remote: true,
                           class: 'btn unfollow-button',
                           id: 'unfollow-button',
                           method: :post
  - else
    = link_to 'Follow', follow_user_path,
                        remote: true,
                        class: 'btn follow-button',
                        id: 'follow-button',
                        method: :post
```

As of now, you should have a completely functional Follow / Unfollow button that correctly creates and destroys relationships in our application!

Congrats! If I was there in that room with you I'd give you a high five, followed promptly by a cuddle, followed promptly by our secret handshake. I love our handshake, especially that little flair at the end.

You know the one.

Let's go forth and adjust our news feed now so it only shows the posts of the users who we follow.

## Selective Selfies

Up until now, we've been seeing ALL of the posts ever created by ALL users on our news feed. It was a far too social feature of our social network.

If we check out the `index` action in our `PostsController` we can see why:

```
def index
  @posts = Post.all.order('created_at DESC').page params[:page]
end
```

We're loading ALL of our posts, ordering them and paginating them. Let's add a scope to our `Post` model so we can easily show only the posts for the users we're following.

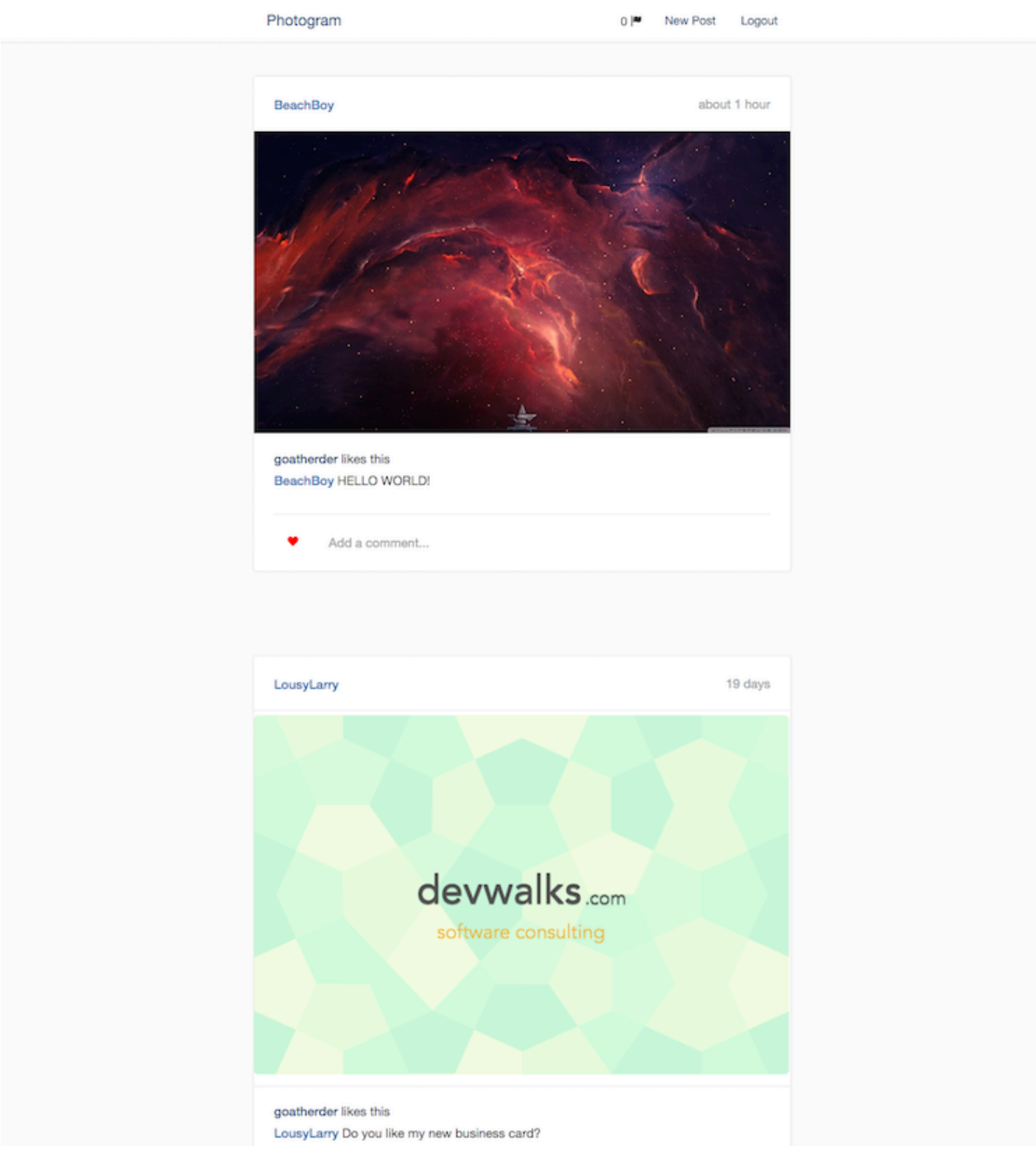
Open the `app/models/post.rb` file and add the following line:

```
scope :of_followed_users, -> (following_users) { where user_id:
following_users }
```

This let's us adjust our `index` action now:

```
def index
    @posts =
Post.of_followed_users(current_user.following).order('created_at
DESC').page params[:page]
end
```

Now, go back to your browser and refresh....




We now have a filtered feed!

## Let's find new users to follow

What we'll do to finish this guide is to make sure that our wonderful users can browse ALL other posts (just as they were before) in order to find new and exciting users to follow.

All we'll do is create a new action in our post controller called `browse`, create a new associated view, and update our navbar to have a link to the browse view.

Go on, give it a go yourself! Here's a hint, this functionality is EXACTLY what we were doing before! The only difference is that you're creating a new controller action and a new view.

Here's one more hint:

The new view and the old index view are going to be *very* similar. What if we created a new partial view that they could both render?


---

Create a new partial view called `_post_dashboard.html.haml`. Copy over all your haml from your index view into the new view.

```
#posts
  = render 'posts'

#paginator.text-center
  = link_to_next_page @posts, 'LOAD MORE', remote: true, id:
'load_more'
```

Now, create a new javascript partial view called `_post_dashboard.js.erb` and copy over all of the logic from the `index.js.erb` file.

```
$('#posts').append("<%= escape_javascript(render 'posts')%>");
$('#paginator').html("<%=
escape_javascript(link_to_next_page(@posts, 'LOAD MORE', remote:
true, id: 'load_more'))%>");
if (!$('#load_more').length) { $('#paginator').remove(); }
```

Wonderful! Now you can delete your old `index.js.erb` file and replace your `index.html.haml` view with this lonely little line:

```
= render 'post_dashboard', posts: @posts
```

So now we've got a new partial and the index view is working just as it was. Let's add our new browse action & view.

First, create the new action in your `posts_controller.rb` file.

```
def browse
  @posts = Post.all.order('created_at DESC').page params[:page]
end
```

That code looks familiar!

Let's now create our new view at `app/views/posts/browse.html.haml`:

```
= render 'post_dashboard', posts: @posts
```

Which also looks familiar...

Let's make sure we have a route for our new view & action. In your `routes.rb` file, add the following line:

```
get 'browse', to: 'posts#browse', as: :browse_posts
```

Last but most certainly not least, let's make it easy for our users to browse all posts. Let's add it to our `app/views/layouts/_navbar.html.haml` view.

```
- if user_signed_in?
  %li
    = render 'notifications/notification_dropdown'
  %li
    = link_to "Browse Posts", browse_posts_path
  %li
    = link_to "New Post", new_post_path
  %li
    = link_to "Logout", destroy_user_session_path, method:
:delete
- else
   %li
     = link_to "Login", new_user_session_path
   %li
     = link_to "Register", new_user_registration_path
```

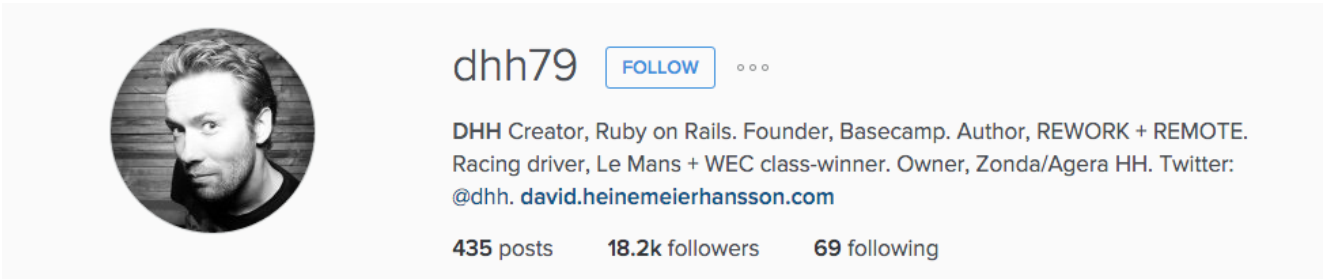
Check it out for yourself. Restart your browser and you should see the feed of ALL of the posts, just as it was back in the old days.

Now let's sit back and reminisce together...

## Homework & the end of an era

There are a few features that would be lovely to have as part of photogram at this point. Some of them easier, some of them trickier. You should definitely try to build them all.

- I'd love to have the follower and following counts shown on each profile. How else can I tell who is popular?




- I'd love for my own posts to be shown on my dashboard too. How could I achieve that?

That should be enough for the moment. Once you've done those couple of things, you'll have a pretty well rounded social network on your hands!