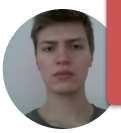


Dec 16, 2017 · 103 min read

The Ultimate Intermediate Ruby on Rails Tutorial: Let's Create an Entire App!

There are plenty tutorials online which show how to create your first app. This tutorial will go a step further and explain line-by-line how to create a more complex Ruby On Rails application.

Throughout the whole tutorial, I will gradually introduce new techniques and concepts. The idea is that with every new section you should learn something new.

The following topics will be discussed throughout this guide:

- Ruby On Rails basics
- Refactoring: helpers, partials, concerns, design patterns

- Testing: TDD/BDD (RSpec & Capybara), Factories (Factory Girl)
- Action Cable
- Active Job
- CSS, Bootstrap, JavaScript, jQuery

So what the app is going to be about?

It's going to be a platform where you could search and meet like-minded people.

Main functionalities which the app will have:

- Authentication (with Devise)
- Ability to publish posts, and search and categorize them
- Instant messaging (popup windows and a separate messenger), with the ability to create private and group conversations.
- Ability to add users to contacts
- Real time notifications

You can see how the complete application is going to look.

And you can find the complete project's source code on GitHub.

Table of Contents

Single post
Specific branches
Service objects
Create a new post

4. **Instant messaging**
   Private conversation
   Contacts
   Group conversation
   Messenger

5. **Notifications**
   Contact requests
   Conversations

Prerequisites

I will try to explain every line of code and how I came up with the solutions. I think it is entirely possible for a total beginner to complete this guide. But keep in mind that this tutorial covers some topics which are beyond the basics.

So if you are a total beginner, it's going to be harder, because your learning curve is going to be pretty steep. I will provide links to resources where you could get some extra information about every new concept we touch.

Ideally, it's best if you are aware of the fundamentals of:

- HTML, CSS, Bootstrap, JavaScript, jQuery
- Ruby, Ruby On Rails
- Git

Setup

I assume that you have already set up your basic Ruby On Rails development environment. If not, check RailsInstaller.

I had been developing on Windows 10 for a while. At first it was okay, but after some time I got tired of overcoming mystical obstacles which were caused by Windows. I had to keep figuring out hack ways to make my applications work. I've realized that it isn't worth my time. Overcoming those obstacles didn't give me any valuable skills or

knowledge. I was just spending my time by duct taping Windows 10 setup.

So I switched to a virtual machine instead. I chose to use Vagrant to create a development environment and PuTTY to connect to a virtual machine. If you want to use Vagrant too, this is the tutorial which I found useful.

Create a new app

We are going to use PostgreSQL as our database. It is a popular choice among Ruby On Rails community. If you haven't created any Rails apps with PostgreSQL yet, you may want to check this tutorial.

Once you are familiar with PostgreSQL, navigate to a directory where you keep your projects and open a command line prompt.

To generate a new app run this line:

```
rails new collabfield --database=postgresql
```

`Collabfield`, that's how our applications is going to be called. By default Rails uses SQlite3, but since we want to use PostgreSQL as our database, we need to specify it by adding:

```
--database=postgresql
```

Now we should've successfully generated a new application.

Navigate to a newly created directory by running the command:

```
cd collabfield
```

And now we can run our app by entering:

```
rails s
```

We just started our app. Now we should be able to see what we got so far. Open a browser and go to http://localhost:3000. If everything went well, you should see the Rails signature welcome page.

Layout

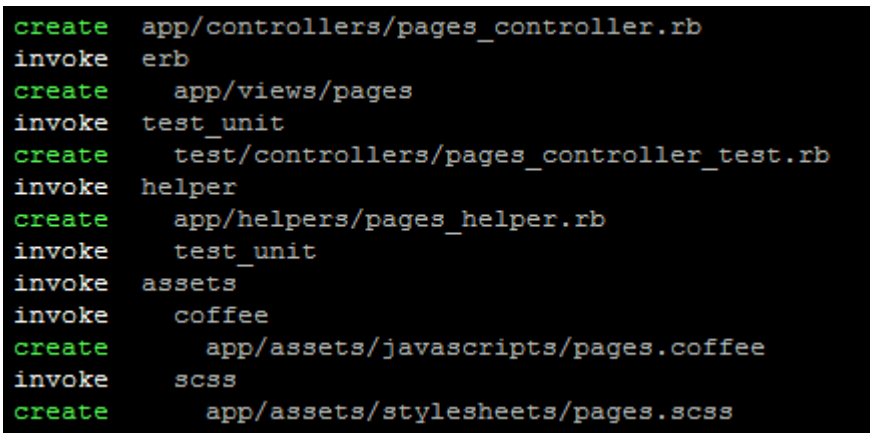
Time to code. Where should we start? Well, we can start wherever we want to. When I build a new website, I like to start by creating some kind of basic visual structure and then build everything else around that. Let's do just that.

Home page

When we go to http://localhost:3000, we see the Rails welcome page. We're going to switch this default page with our own home page. In order to do that, generate a new controller called `Pages` . If you are not familiar with Rails controllers, you should skim through the Action Controller to get an idea what the Rails controller is. Run this line in your command prompt to generate a new controller.

```
rails g controller pages
```

This rails generator should have created some files for us. The output in the command prompt should look something like this:



We are going to use this `PagesController` to manage our special and static pages. Now open the Collabfield project in a text editor. I use Sublime Text, but you can use whatever you want to.

Open a file `pages_controller.rb`

```
app/controllers/pages_controller.rb
```

We'll define our home page here. Of course we could define home page in a different controller and in different ways. But usually I like to define the home page inside the `PagesController`.

When we open `pages_controller.rb`, we see this:

```
1  class PagesController < ApplicationController
2  end
```

controllers/pages_controller.rb

It's an empty class, named `PagesController`, which inherits from the `ApplicationController` class. You can find this class's source code in `app/controllers/application_controller.rb`.

All our controllers, which we will create, are going to inherit from `ApplicationController` class. Which means that all methods defined inside this class are going to be available across all our controllers.

We'll define a public method named `index`, so it can be callable as an action:

```
class PagesController < ApplicationController

  def index
  end

```

controllers/pages_controller.rb

As you may have read in the Action Controller, routing determines which controller and its public method (action) to call. Let's define a route, so when we open our root page of the website, Rails knows which controller and its action to call. Open a `routes.rb` file in `app/config/routes.rb`.

If you don't know what Rails routes is, it is a perfect time to get familiar by reading the Rails Routing.

Insert this line:

```
root to: 'pages#index'
```

Your `routes.rb` file should look like this:

```
Rails.application.routes.draw do
  root to: 'pages#index'
end
```

config/routes.rb

Hash symbol `#` in Ruby represents a method. As you remember an action is just a public method, so `pages#index` says "call the `PagesController` and its public method (action) `index`."

If we went to our root path http://localhost:3000, the index action would be called. But we don't have any templates to render yet. So let's

create a new template for our `index` action. Go to `app/views/pages` and create an `index.html.erb` file inside this directory. Inside this file we can write our regular HTML+ Embedded Ruby code. Just write something inside the file, so we could see the rendered template in the browser.

```
<h1>Home page</h1>
```

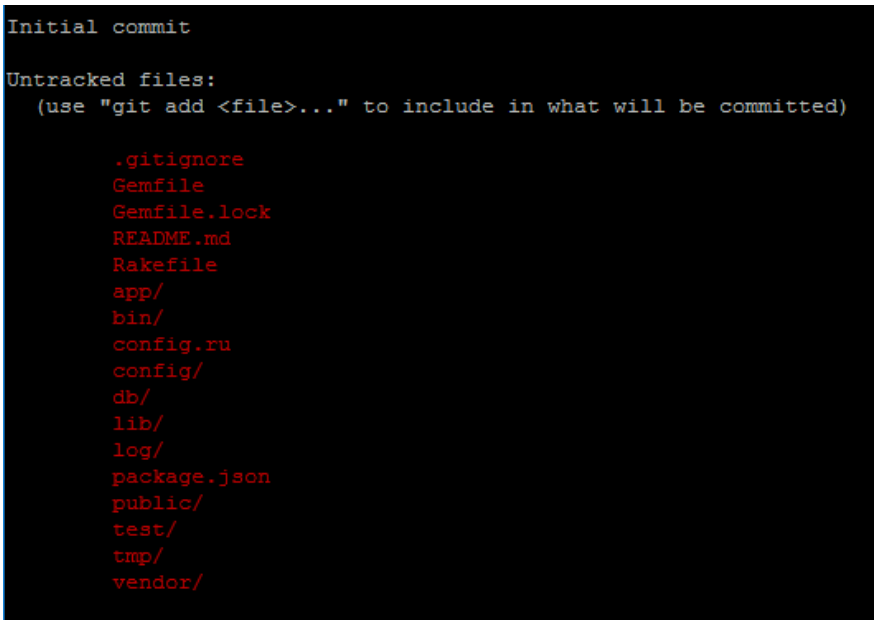
Now when we go to http://localhost:3000, we should see something like this instead of the default Rails information page.

Now we have a very basic starting point. We can start introducing new things to our website. I think it's time to create our first commit.

In your command prompt run:

```
git status
```

And you should see something like this:



If you don't already know, when we generate a new application, a new local git repository is initialized.

Add all current changes by running:

```
git add -A
```

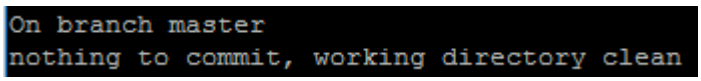
Then commit all changes by running:

```
git commit -m "Generate PagesController. Initialize Home
page"
```

If we ran this:

```
git status
```

We would see that there's nothing to commit, because we just successfully committed all changes.



Bootstrap

For the navigation bar and the responsive grid system we're going to use the Bootstrap library. In order to use this library we have to install the
bootstrap-sass gem. Open the `Gemfile` in your editor.

```
collabfield/Gemfile
```

Add a `bootstrap-sass` gem to the Gemfile. As the documentation says, you have to ensure that `sass-rails` gem is present too.

```
...
gem 'bootstrap-sass', '~> 3.3.6'
gem 'sass-rails', '>= 3.2'
...
```

Save the file and run this to install newly added gems:

```
bundle install
```

If you are still running the application, restart the Rails server to make sure that new gems are available. To restart the server simply shutdown it by pressing `Ctrl + C` and run `rails s` command again to boot the server.

Go to `assets` to open the `application.css` file:

`app/assets/stylesheets/application.css`

Below all the commented text add this:

```
...
@import "bootstrap-sprockets";
@import "bootstrap";
```

Now change the `application.css` name to `application.scss`. This is necessary in order to use Bootstrap library in Rails, also it allows us to use Sass features.

We want to control the order in which all `.scss` files are rendered, because in the future we might want to create some Sass variables. We want to make sure that our variables are going to be defined before we use them.

To accomplish it, remove those two lines from the `application.scss` file:

```
*= require_self
*= require_tree .
```

We're almost able to use Bootstrap library. There's a one more thing which we have to do. As the bootstrap-sass docs says, Bootstrap JavaScript is dependent on jQuery library. To use jQuery with Rails, you have to add jquery-rails gem.

```
gem 'jquery-rails'
```

Run…

```
bundle install
```

…again, and restart the server.

Last step is to require Bootstrap and jQuery in the application's JavaScript file. Go to `application.js`

```
app/assets/javascripts/application.js
```

Then add the following lines in the file:

```
//= require jquery
//= require bootstrap-sprockets
```

Commit the changes:

```
git add -A
git commit -m "Add and configure bootstrap gem"
```

Navigation bar

For the navigation bar we'll use Bootstrap's navbar component as the starting point and then quite modify it. We will store our navigation bar inside a partial template.

We're doing this because it's better to keep every component of the app in separate files. It allows to test and manage app's code much easier. Also we can reuse those components in other parts of the app, without duplicating the code.

Navigate to:

```
views/layouts
```

Create a new file:

```
_navigation.html.erb
```

For partials we use underscore prefix, so the Rails framework can distinguish it as a partial. Now copy and paste navbar component from Bootstrap docs and save the file. To see the partial on the website, we have to render it somewhere. Navigate to `views/layouts/application.html.erb` . This is the default file where everything gets rendered.

Inside the file we see the following method:

```
<%= yield %>
```

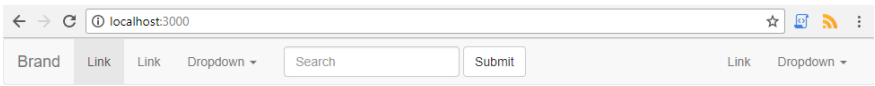
It renders the requested template. To use ruby syntax inside the HTML file, we have to wrap it around with `<% %>` (embedded ruby allows us to do that). To quickly learn the differences between ERB syntax, checkout this StackOverflow answer.

In Home page section we set the route to recognize the root URL. So whenever we send a GET request to go to a root page, `PagesController'sindex` action gets called. And that respective action (in this case the `index` action) responds with a template which gets rendered with the `yield` method. As you remember, our template for a home page is located at `app/views/pages/index.html.erb` .

Since we want to have a navigation bar across all pages, we'll render our navigation bar inside the default `application.html.erb` file. To render a partial file , simply use the `render` method and pass partial's path as an argument. Do it just above the `yield` method like this:

```
...
<%= render 'layouts/navigation' %>
<%= yield %>
...
```

Now go to http://localhost:3000 and you should be able to see the navigation bar.



Home page

As mentioned above, we're going to modify this navigation bar. First let's remove all `<li>` and `<form>` elements. In the future we'll create our own elements here. The `_navigation.html.erb` file should look like this now.

```
<nav class="navbar navbar-default">
  <div class="container-fluid">
    <!-- Brand and toggle get grouped for better mobil
    <div class="navbar-header">
      <button type="button" class="navbar-toggle colla
        <span class="sr-only">Toggle navigation</span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
      </button>
      <a class="navbar-brand" href="#">Brand</a>
    </div>

    <!-- Collect the nav links, forms, and other conte
    <div class="collapse navbar-collapse" id="bs-examp
      <ul class="nav navbar-nav">
```

layouts/_navigation.html.erb

We have a basic responsive navigation bar now. It's a good time to create a new commit. In command prompt run the following commands:

```
git add -A
git commit -m "Add a basic navigation bar"
```

We should change the navigation bar's name from `Brand` to `collabfield`. Since `Brand` is a link element, we should use a `link_to` method to generate links. Why? Because this method allows us to easily generate URI paths. Open a command prompt and navigate to the project's directory. Run the the following command:

```
rails routes
```

This command outputs our available routes, which are generated by the `routes.rb` file. As we see:

```
Prefix Verb URI Pattern Controller#Action
  root GET  /           pages#index
```

Currently, we have only one route, the one which we've defined before. If you look at the given routes, you can see a `Prefix` column. We can use those prefixes to generate a path to a wanted page. All we have to do is use a prefix name and add `_path` to it. If we wrote the `root_path`, that would generate a path to the root page. So let's use the power of `link_to` method and routes.

Replace this line:

```
<a class="navbar-brand" href="#">Brand</a>
```

With this line:

```
<%= link_to 'collabfield', root_path, class: 'navbar-brand'
%>
```

Remember that whenever you don't quite understand how a particular method works, just Google it and you will probably find its documentation with an explanation. Sometimes documentations are poorly written, so you might want to Google a little bit more and you might find a blog or a StackOverflow answer, which would help.

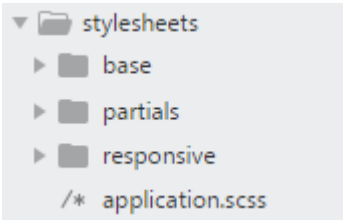
In this case we pass a string as our first argument to add the `<a>` element's value, the second argument is needed for a path, this is where routes helps us to generate it. The third argument is optional, which is accumulated inside the options hash. In this case we needed to add `navbar-brand` class to keep our Bootstrap powered navigation bar to function.

Let's do another commit for this small change. In the upcoming section we'll start changing our app's design, starting from the navigation bar.

```
git add -A
git commit -m "Change navigation bar's brand name from Brand
to collabfield"
```

Style sheets

Let me introduce you how I structure my style sheet files. From what I know there aren't any strong conventions on how to structure your style sheets in Rails. Everyone is doing it slightly differently.



This is how I usually structure my files.

- **Base directory**—This is where I keep Sass variables and styles which are used throughout the whole app. For instance default font sizes and default elements' styles.
- **Partials**—Most of my styles go there. I keep all styles for separate components and pages in this directory.
- **Responsive**—Here I define different style rules for different screen sizes. For example, styles for a desktop screen, tablet screen, phone screen, etc.

First, let's create a new repository branch by running this:

```
git checkout -b "styles"
```

We've just created a new git branch and automatically switched to it. From now on this is how we're going to implement new changes to the code.

The reason for doing this is that we can isolate our currently functional version (master branch) and write a new code inside a project's copy, without being afraid to damage anything.

Once we are complete with the implementation, we can just merge changes to the `master` branch.

Start by creating few directories:

```
app/assets/stylesheets/partials/layout
```

Inside the layout directory create a file `navigation.scss` and inside the file add:

```
.navbar-default, .navbar-toggle:focus, .collapsed, butt
  background: $navbarColor !important;
  border: none;
  a {
    color: white !important;
  }
```

app/assets/stylesheets/partials/layout/navigation.scss

With these lines of code we change navbar's background and links color. As you may have noticed, `a` selector is nested inside another declaration block. Sass allows us to use this functionality. `!important` is used to strictly override default Bootstraps styles. The last thing which you may have noticed is that instead of a color name, we use a Sass variable. The reason for this is that we are going to use this color multiple times across the app. Let's define this variable.

First create a new folder:

```
app/assets/stylesheets/base
```

Inside the base directory create a new file `variables.scss`. Inside the file define a variable:

```
$navbarColor: #323738;
```

If you tried to go to http://localhost:3000, you wouldn't notice any style changes. The reason for that is that in the Bootstrap section we removed these lines:

```
*= require_self
*= require_tree .
```

from `application.scss`, to not automatically import all style files.

This means that now we have to import our newly created files to the main `application.scss` file. The file should look like this now:

```
// ...default comments

// Bootstrap
@import "bootstrap-sprockets";
@import "bootstrap";

// Variables
@import "base/variables";
```

app/assets/stylesheets/application.scss

The reason for importing `variables.scss` file at the top is to make sure that the variables are defined before we use them.

Add some more CSS at the top of the `navigation.scss` file:

```
nav {
  .navbar-header {
    width: 100%;
    button, .navbar-brand {
      transition: opacity 0.15s;
    }
    button {
      margin-right: 0;
    }
    button:hover, .navbar-brand:hover {
```

app/assets/stylesheets/partials/layout/navigation.scss

Of course you can put this code at the bottom of the file if you want to. Personally, I order and group CSS code based on CSS selectors' specificity. Again, everyone is doing it slightly differently. I put less specific selectors above and more specific selectors below. So for instance Type selectors go above Class selectors and Class selectors go above ID selectors.

Let's commit changes:

```
git add -A
git commit -m "Add CSS to the navigation bar"
```

We want to make sure that the navigation bar is always visible, even when we scroll down. Right now we don't have enough content to scroll down, but we will in the future. Why don't we give this feature to the navigation bar right now?

To do that use Bootstrap class `navbar-fixed-top`. Add this class to the `nav` element, so it looks like this:

```
<nav class="navbar navbar-default navbar-fixed-top">
```

Also we want to have `collabfield` to be to the Bootstrap Grid System's left side boundaries. Right now it is to the viewport's left side boundaries, because our class is currently `container-fluid`. To change that, change the class to `container`.

It should look like this:

```
<div class="container">
```

Commit the changes:

```
git add -A
git commit -m "
- in _navigation.html.erb add navbar-fixed-top class to nav.
- Replace container-fluid class with container"
```

If you go to http://localhost:3000, you see that the `Home page` text is hidden under the navigation bar. That's because of the `navbar-fixed-top` class. To solve this issue, push the body down by adding the following to `navigation.scss`:

```
body {
 margin-top: 50px;
}
```

At this stage the app should look like this:

Commit the change:

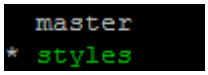
```
git add -A
git commit -m "Add margin-top 50px to the body"
```

As you remember, we've created a new branch before and switched to it. It's time to go back to the `master` branch.

Run the command:

```
git branch
```

You can see the list of our branches. Currently we're in the `styles` branch.



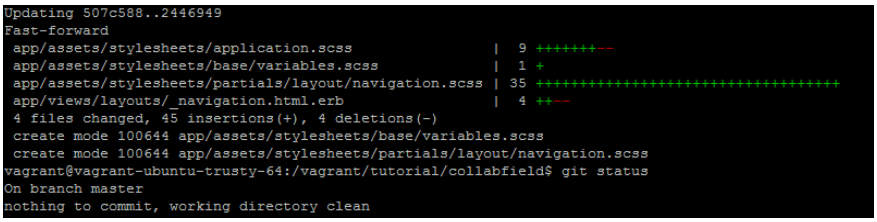
To switch back to the `master` branch, run:

```
git checkout master
```

To merge our all changes, which we did in the `styles` branch, simply run:

```
git merge styles
```

The command merged those two branches and now we can see the summary of changes we made.



We don't need `styles` branch anymore, so we can delete it:

```
git branch -D styles
```

Posts

It's almost a right time to start implementing the posts functionality. Since our app goal is to let users meet like-minded people, we have to make sure that posts' authors can be identified. To achieve that, authentication system is required.

Authentication

For an authentication system we are going to use the devise gem. We could create our own authentication system, but that would require a lot of effort. We'll choose an easier route. Also it's a popular choice among Rails community.

Start by creating a new branch:

```
git checkout -b authentication
```

Just like with any other gem, to set it up we'll follow its documentation. Fortunately, it's very easy to set up.

Add to your `Gemfile`

```
gem 'devise'
```

Then run commands:

```
bundle install
rails generate devise:install
```

You probably see some instructions in the command prompt. We won't use mailers in this tutorial, so no further configuration is needed.

At this point, if you don't know anything about Rails models, you should get familiar with them by skimming through Active Record and Active Modeldocumentations.

Now let's use a devise generator to create a `User` model.

```
rails generate devise User
```

Initialize a database for the app by running:

```
rails db:create
```

Then run this command to create new tables in your database:

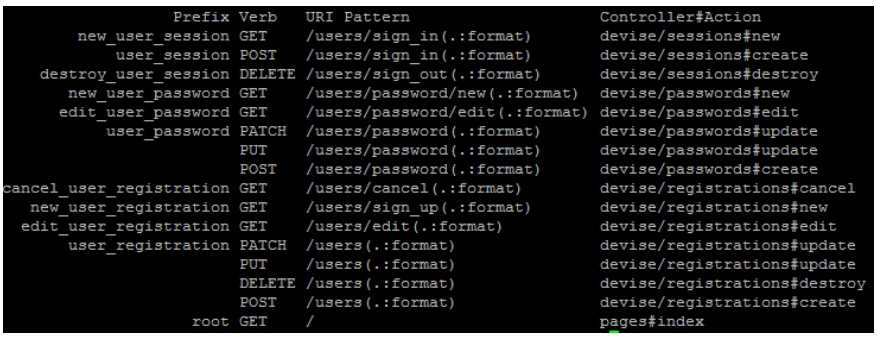
```
rails db:migrate
```

That's it. Technically our authentication system is set up. Now we can use Devise given methods and create new users. Commit the change:

```
git add -A
git commit -m "Add and configure the Devise gem"
```

By installing Devise gem, we not only get the back-end functionality, but also default views. If you list your routes by running:

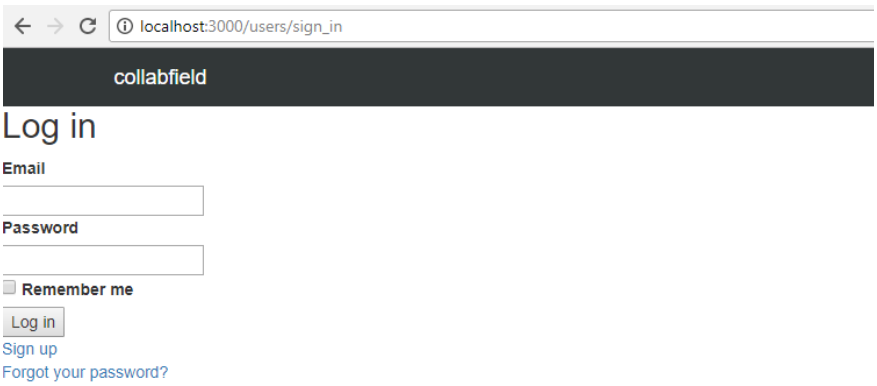
```
rails routes
```



```
                  Prefix Verb   URI Pattern                    Controller#Action
        new_user_session GET    /users/sign_in(.:format)       devise/sessions#new
            user_session POST   /users/sign_in(.:format)       devise/sessions#create
    destroy_user_session DELETE /users/sign_out(.:format)      devise/sessions#destroy
       new_user_password GET    /users/password/new(.:format)  devise/passwords#new
      edit_user_password GET    /users/password/edit(.:format) devise/passwords#edit
          user_password PATCH  /users/password(.:format)      devise/passwords#update
                        PUT    /users/password(.:format)      devise/passwords#update
                        POST   /users/password(.:format)      devise/passwords#create
cancel_user_registration GET    /users/cancel(.:format)        devise/registrations#cancel
   new_user_registration GET    /users/sign_up(.:format)       devise/registrations#new
  edit_user_registration GET    /users/edit(.:format)          devise/registrations#edit
      user_registration PATCH  /users(.:format)               devise/registrations#update
                        PUT    /users(.:format)               devise/registrations#update
                        DELETE /users(.:format)               devise/registrations#destroy
                        POST   /users(.:format)               devise/registrations#create
                  root GET    /                              pages#index
```

You can see that now you have a bunch of new routes. Remember, we only had a one root route until now. If something seems to be confusing, you can always open devise docs and get your answers. Also don't forget that a lot of same questions come to other people' s minds. There's a high chance that you'll find the answer by Googling too.

Try some of those routes. Go to localhost:3000/users/sign_in and you should see a sign in page.



If you went to localhost:3000/users/sign_up, you would see a sign up page too. God Damn! as Noob Noob says. If you look at the `views` directory, you see that there isn't any Devise directory, which we could modify. As Devise docs says, in order to modify Devise views, we've to generate it with a devise generator. Run

```
rails generate devise:views
```

If you check the `views` directory, you'll see a generated devise directory inside. Here we can modify how sign up and login pages are going to look like. Let's start with the login page, because in our case this is going to be a more straightforward implementation. With the registration page, due to our wanted feature, an extra effort will be required.

Login page

Navigate to and open `app/views/devise/sessions/new.html.erb` .

This is where the login page views are stored. There's just a login form inside the file. As you may have noticed, the `form_for` method is used to generate this form. This is a handy Rails method to generate forms. We're going to modify this form's style with bootstrap. Replace all file's content with:

```
<%= bootstrap_form_for(resource,
                       as: resource_name,
                       url: session_path(resource_name

    <%= f.email_field :email,
                      autofocus: true,
                      class: 'form-control',
                      placeholder: 'email' %>

    <%= f.password_field  :password,
                          autocomplete: "off",
                          class: 'form-control',
                          placeholder: 'password' %>

```

views/devise/sessions/new.html.erb

Nothing fancy is going here. We just modified this form to be a bootstrap form by changing the method's name to `bootstrap_form_for` and adding `form-control` classes to the fields.

Take a look how arguments inside the methods are styled. Every argument starts in a new line. The reason why I did this is to avoid having long code lines. Usually code lines shouldn't be longer than 80 characters, it improves readability. We're going to style the code like that for the rest of the guide.

If we visit localhost:3000/users/sign_in, we'll see that it gives us an error:

```
undefined method 'bootstrap_form_for'
```

In order to use bootstrap forms in Rails we've to add a bootstrap_form gem. Add this to the `Gemfile`

```
gem 'bootstrap_form'
```

Then run:

```
bundle install
```

At this moment the login page should look like this:

Commit changes:

```
git add -A
git commit -m "Generate devise views, modify sign in form
and add the bootstrap_form gem."
```

To give the bootstrap's grid system to the page, wrap login form with the bootstrap container.

```
<div class="container">
  <div class="row">
    <div class="col-sm-6 col-sm-offset-3">
      <h2 class="text-center">Log in</h2>

      <!-- PASTE LOGIN FORM HERE -->

    </div>
```

views/devise/sessions/new.html.erb

The width of the login form is 6 columns out of 12. And the offset is 3 columns. On smaller devices the form will take full screen's width. That's how the bootstrap gridworks.

Let's do another commit. Quite a minor change, huh? But that's how I usually do commits. I implement a definite change in one area and then commit it. I think doing it this way helps to track changes and understand how the code has evolved.

```
git add -A
git commit -m "wrap login form in the login page with a
boostrap container"
```

It would be better if we could just reach the login page by going to `/login` instead of `/users/sign_in`. We have to change the route. To do that we need to know where the action, which gets called when we go to login page, is located. Devise controllers are located inside the gem itself. By reading Devise docs we can see that all controllers are located inside the `devise` directory. Not really surprised by the discovery, to be honest U_U. By using the `devise_scope` method we can simply change the route. Go to `routes.rb` file and add

```
devise_scope :user do
  get 'login', to: 'devise/sessions#new'
end
```

Commit the change:

```
git add -A
git commit -m "change route from /users/sign_in to /login"
```

For now, leave the login page as it is.

Sign up page

If we navigated to localhost:3000/users/sign_up, we would see the default Devise sign up page. But as mentioned above, the sign up page will require some extra effort. Why? Because we want to add a new `:name` column to the `users` table, so a User object could have the `:name` attribute.

We're about to do some changes to the `schema.rb` file. At this moment, if you aren't quite familiar with schema changes and migrations, I recommend you to read through Active Record Migrations docs.

First, we have to add an extra column to the `users` table. We could create a new migration file and use a `change_table` method to add an extra column. But we are just at the development stage, our app isn't deployed yet. We can just define a new column straight inside the `devise_create_users` migration file and then recreate the database. Navigate to `db/migrate` and open the `*CREATION_DATE*_devise_create_users.rb` file and add `t.string :name, null: false, default: ""` inside the `create_table` method.

Now run the commands to drop and create the database, and run migrations.

```
rails db:drop
rails db:create
rails db:migrate
```

We added a new column to the users table and altered the `schema.rb` file.

To be able to send an extra attribute, so the Devise controller would accept it, we've to do some changes at the controller level. We can do changes to Devise controllers in few different ways. We can use devise generator and generate controllers. Or we can create a new file, specify the controller and the methods that we want to modify. Both ways are good. We are going to use the latter one.

Navigate to `app/controllers` and create a new file `registrations_controller.rb`. Add the following code to the file:

```
class RegistrationsController < Devise::RegistrationsC

  private

  def sign_up_params
    params.require(:user).permit( :name,
                                  :email,
                                  :password,
                                  :password_confirmati
  end

  def account_update_params
    params.require(:user).permit( :name,
```

controllers/registrations_controller.rb

This code overwrites the `sign_up_params` and `account_update_params` methods to accept the `:name` attribute. As you see, those methods are in the Devise `RegistrationsController`, so we specified it and altered its methods. Now inside our routes we have to specify this controller, so these methods could be overwritten. Inside `routes.rb` change

```
devise_for :users
```

to

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

Commit the changes.

```
git add -A
git commit -m "
- Add the name column to the users table.
- Include name attribute to sign_up_params and
account_update_params
  methods inside the RegistrationsController"
```

Open the `new.html.erb` file:

```
app/views/devise/registrations/new.html.erb
```

Again, remove everything except the form. Convert the form into a bootstrap form. This time we add an additional name field.

```
1  <%= bootstrap_form_for(resource,
2                         :as => resource_name,
3                         :url => registration_path(resou
4
5    <%= f.text_field :name,
6                     placeholder: 'username (will be sho
7                     class: 'form-control' %>
8    <%= f.text_field :email,
9                     placeholder: 'email',
10                    class: 'form-control' %>
11   <%= f.password_field :password,
12                        placeholder: 'password',
```

views/devise/registrations/new.html.erb

Commit the change.

```
git add -A
git commit -m "
```

```
Delete everything from the signup page, except the form.
Convert form into a bootstrap form. Add an additional name
field"
```

Wrap the form with a bootstrap container and add some text.

```
<div class="container" id="sign-up-form">
  <div class="row">
    <h1>Get in touch with like-minded people</h1>
    <h3>Create, study, accomplish goals together</h3>

    <div class="col-sm-offset-4 col-sm-4">
      <h3>Sign up <small>it's free!</small></h3>

        <!-- PASTE THE FORM HERE -->
```

views/devise/registrations/new.html.erb

Commit the change.

```
git add -A
git commit -m "
Wrap the sign up form with a bootstrap container.
Add informational text inside the container"
```

Just like with the login page, it would be better if we could just open a sign up page by going to `/signup` instead of `users/sign_up`. Inside the `routes.rb` file add the following code:

```
devise_scope :user do
  get 'signup', to: 'devise/registrations#new'
end
```

Commit the change.

```
git add -A
git commit -m "Change sign up page's route from
/users/sign_up to /signup"
```

Let's apply a few style changes before we move on. Navigate to `app/assets/sytlesheets/partials` and create a new `signup.scss` file. Inside the file add the following CSS:

```
#sign-up-form {
  margin-top: 100px;
  h1 {
    font-size: 36px !important;
    font-size: 3.6rem !important;
  }
  text-align: center;
```

assets/stylesheets/partials/signup.scss

Also we haven't imported files from the `partials` directory inside the `application.scss` file. Let's do it right now. Navigate to the `application.scss` and just above the `@import partials/layout/*`, import all files from the `partials` directory. `Application.scss` should look like this

```
...

// Partials - main css files
@import "partials/*";
```

assets/stylesheets/application.scss

Commit the changes.

```
git add -A
git commit -m "
- Create a signup.scss and add CSS to the sign up page
- Import all files from partials directory to the
application.scss"
```

Add few other style changes to the overall website look. Navigate to `app/assets/stylesheets/base` and create a new `default.scss` file. Inside the file add the following CSS code:

```
* {
  box-sizing: border-box;
}

html {
  font-size: 62.5%;
}

body {
  background: $backgroundColor;
  font-size: 14px;
  font-size: 1.4rem;
}

h1 {
  font-size: 24px;
  font-size: 2.4rem;
}

i {
  width: 26px;
}
```

assets/stylesheets/base/default.scss

Here we apply some general style changes for the whole website. `font-size` is set to `62.5%`, so `1 rem` unit could represent `10px`. If you don't know what the rem unit is, you may want to read this tutorial. We don't want to see a label text on bootstrap forms, that's why we set this:

```
.control-label {
  display: none;
}
```

You may have noticed that the `$backgroundColor` variable is used. But this variable isn't set yet. So let's do it by opening `variables.scss` file and adding this:

```
$backgroundColor: #f0f0f0;
```

The `default.scss` file isn't imported inside the `application.scss`. Import it below variables, the `application.scss` file should look like this:

```
...

// Variables
@import "base/variables";

// Default styles
@import "base/default";
```

assets/stylesheets/application.scss

Commit the changes.

```
git add -A
git commit -m "
Add CSS and import CSS files inside the main file

- Create a default.scss file and add CSS
- Define $backgroundColor variable
- Import default.scss file inside the application.scss"
```

Navigation bar update

Right now we have three different pages: home, login and signup. It is a good idea to connect them all together, so users could navigate through the website effortlessly. We'll put links to signup and login pages on the navigation bar. Navigate to and open the `_navigation.html.erb` file.

```
app/views/layouts/_navigation.html.erb
```

We're going to add some extra code here. In the future we will add even more code here. This will lead to a file with lots of code, which is hard to manage and test. In order to handle a long code easier, we're going to start splitting larger code into smaller chunks. To achieve that, we'll use partials. Before adding extra code, let's split the current `_navigation.html.erb` code into partials already.

Let me quickly introduce you how our navigation bar is going to work. We'll have two major parts. On one part elements will be shown all the time, no matter what the screen size is. On the other part of the navigation bar, elements will be shown only on bigger screens and collapsed on the smaller ones.

This is how the structure inside the `.container` element will look like:

```
<div class="row">

  <!-- Elements visible all the time -->
  <div class="col-sm-7">
  </div><!-- col-sm-7 -->

  <!-- Elements collapses on smaller devices -->
  <div class="col-sm-5">
```

layouts/_navigation_html.erb

Inside the `layouts` directory:

```
app/views/layouts
```

Create a new `navigation` directory. Inside this directory create a new partial `_header.html.erb` file.

```
app/views/layouts/navigation/_header.html.erb
```

From the `_navigation.html.erb` file cut the whole `.navbar-header` section and paste it inside the `_header.html.erb` file. Inside the `navigation` directory, create another partial file named `_collapsible_elements.html.erb` .

```
app/views/layouts/navigation/_collapsible_elements.html.erb
```

From the `_navigation.html.erb` file cut the whole `.navbar-collapse` section and paste it inside the `_collapsible_elements.html.erb`. Now let's render those two partials inside the `_navigation.html.erb` file. The file should look like this now.

```
<nav class="navbar navbar-default navbar-fixed-top">
  <div class="container">
    <div class="row">

      <!-- Elements visible all the time -->
      <div class="col-sm-7">
        <%= render 'layouts/navigation/header' %>
      </div><!-- col-sm-7 -->

      <!-- Elements collapses on smaller devices -->
      <div class="col-sm-5">
        <%= render 'layouts/navigation/collapsible ele
```

layouts/_navigation_html.erb

If you went to http://localhost:3000 now, you wouldn't notice any difference. We just cleaned our code a little bit and prepared it for a further development.

We are ready to add some links to the navigation bar. Navigate to and open the `_collapsible_elements.html.erb` file again:

```
app/views/layouts/_collapsible_elements.html.erb
```

Let's fill this file with links, replace the file's content with:

```
<!-- Collect the nav links, forms, and other content f
<div class="collapse navbar-collapse navbar-right" id=
  <ul class="nav navbar-nav ">
    <% if user_signed_in? %>
      <li class="dropdown pc-menu">
        <a id="user-settings" class="dropdown-toggle"
          <span id="user-name"><%= current_user.name %
          <span class="caret"></span>
        </a>

        <ul class="dropdown-menu" role="menu">
          <li><%= link_to 'Edit Profile', edit_user_re
          <li><%= link_to 'Log out', destroy_user_sess
        </ul>
      </li>

      <li class="mobile-menu">
        <%= link_to 'Edit Profile', edit_user_registra
      </li>
```

layouts/navigation/_collapsible_elements.html.erb

Let me briefly explain to you what is going on here. First, at the second line I changed the element's `id` to `navbar-collapsible-content`. This is required in order to make this content collapsible. It's a bootstrap's functionality. The default `id` was `bs-example-navbar-collapse-1`. To trigger this this function there's the button with the `data-target` attribute inside the `_header.html` file. Open `views/layouts/navigation/_header.html.erb` and change `data-target` attribute to `data-target="#navbar-collapsible-content"`. Now the button will trigger the collapsible content.

Next, inside the `_collapsible_elements.html.erb` file you see some `if else` logic with the `user_signed_in?` Devise method. This will show different links based on if a user is signed in, or not. Leaving logic, such as `if else` statements inside views isn't a good practice. Views should be pretty "dumb" and just spit the information out, without "thinking" at all. We will refactor this logic later with Helpers.

The last thing to note inside the file is `pc-menu` and `mobile-menu` CSS classes. The purpose of these classes is to control how links are displayed on different screen sizes. Let's add some CSS for these classes. Navigate to `app/assets/stylesheets` and create a new directory `responsive`. Inside the directory create two files,

`desktop.scss` and `mobile.scss`. The purpose of those files is to have different configurations for different screen sizes. Inside the `desktop.scss` file add:

```
@media screen and (min-width: 767px) {
  .mobile-menu {
    display: none !important;
  }
}
```

assets/styleshseets/responsive/desktop.scss

Inside the `mobile.scss` file add:

```
@media screen and (max-width: 767px) {
  .pc-menu {
    display: none !important;
  }
}
```

assets/styleshseets/responsive/mobile.scss

If you aren't familiar with CSS media queries, read this. Import files from the `responsive` directory inside the `application.scss` file. Import it at the bottom of the file, so the `application.scss` should look like this:

```
...

// Media queries for a responsive design
@import "responsive/*";
```

app/assets/stylesheets/application.scss

Navigate to and open `navigation.scss` file

```
app/assets/stylesheets/partials/layout/navigation.scss
```

and do some stylistic tweaks to the navigation bar by adding the following inside the `nav` element's selector:

```
.col-sm-5, .col-sm-7 {
  padding: 0;
}
```

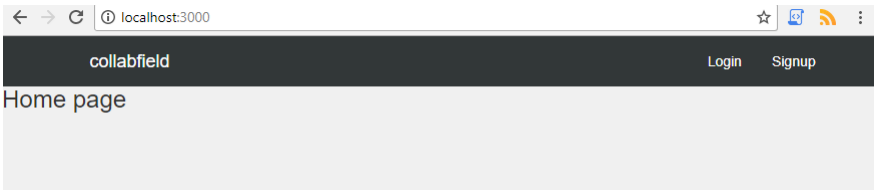
assets/styleshseets/partials/layout/navigation.scss

And outside the `nav` element, add the following CSS code:

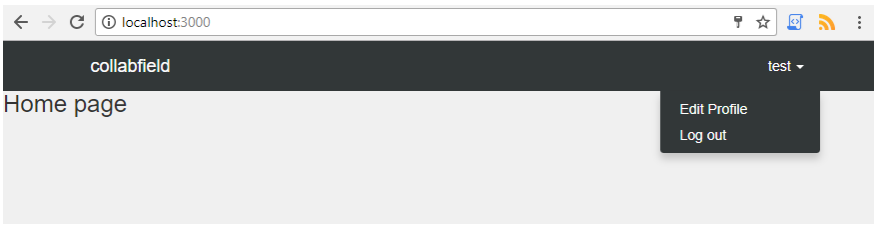
```
.pc-menu {
  margin-right: 10px;
}

.mobile-menu {
  i {
    color: white;
  }
  ul {
    padding: 0px;
  }
  a {
    display: block;
    padding: 10px 0px 10px 25px !important;
  }
  a:hover {
    background: white !important;
    color: black !important;
    i {
      color: black;
    }
  }
}

.icon-bar {
  background-color: white !important;
}
```

assets/styleshseets/partials/layout/navigation.scss

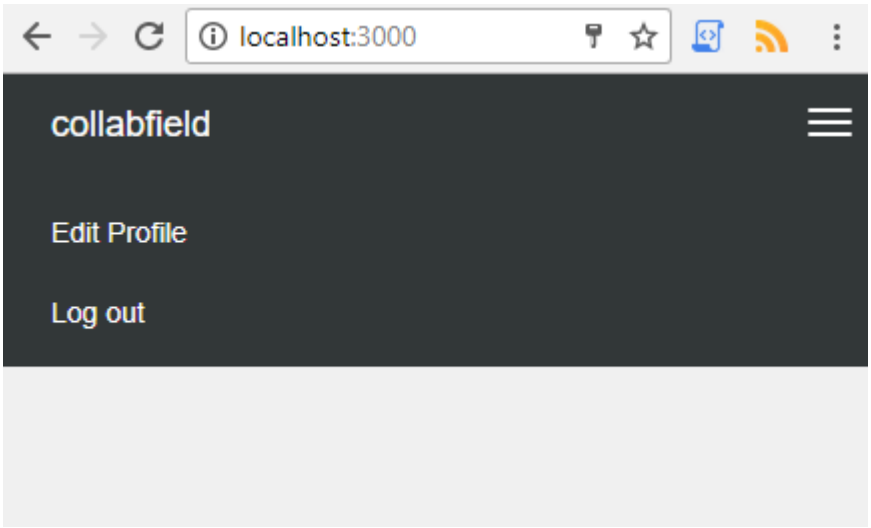
At this moment, our application should look like this when a user is not logged in:



Like this when a user is logged in:



And like this when the screen size is smaller:



Commit the changes.

```
git add -A
git commit -m "
Update the navigation bar

- Add login, signup, logout and edit profile links on the
navigation bar
- Split _navigation.scss code into partials
- Create responsive directory inside the stylesheets
directory and add CSS.
- Add CSS to tweak navigation bar style"
```

Now we have a basic authentication functionality. It satisfies our needs. So let's merge `authentication` branch with the `master` branch.

```
git checkout master
git merge authentication
```

We can see the summary of changes again. Authentication branch is not needed anymore, so delete it.

```
git branch -D authentication
```

Helpers

When we were working on the `_collapsible_elements.html.erb` file, I mentioned that Rails views is not the right place for logic. If you look inside the `app` directory of the project, you see there's the directory called `helpers`. We'll extract logic from Rails views and put it inside the `helpers` directory.

```
app/views/pages
```

Let's create our first helpers. Firstly, create a new branch and switch to it.

```
git checkout -B helpers
```

Navigate to the `helpers` directory and create a new `navigation_helper.rb` file

```
app/helpers/navigation_helper.rb
```

Inside helper files, helpers are defined as modules. Inside the `navigation_helper.rb` define the module.

```
module NavigationHelper
end
```

app/helpers/navigation_helper.rb

By default Rails loads all helper files to all views. Personally I do not like this, because methods' names from different helper files might clash. To override this default behavior open the `application.rb` file

```
config/application.rb
```

Inside the `Application` class add this configuration

```
config.action_controller.include_all_helpers = false
```

Now helpers are available for corresponding controller's views only. So if we have the `PagesController`, all helpers inside the `pages_helper.rb` file will be available to all view files inside the `pages` directory.

We don't have the `NavigationController`, so helper methods defined inside the `NavigationHelper` module won't be available anywhere. The navigation bar is available across the whole website. We can include the `NavigationHelper` module inside the `ApplicationHelper`. If you aren't familiar with loading and including files, read through this article to get an idea what is going to happen.

Inside the `application_helper.rb` file, require the `navigation_helper.rb` file. Now we have an access to the `navigation_helper.rb` file's content. So let's inject `NavigationHelper` module inside the `ApplicationHelper` module by using an `include` method. The `application_helper.rb` should look like this:

```
require 'navigation_helper.rb'

module ApplicationHelper
  include NavigationHelper
```

helpers/application_helper.rb

Now `NavigationHelper` helper methods are available across the whole app.

Navigate to and open the `_collapsible_elements.html.erb` file

```
app/views/layouts/navigation/_collapsible_elements.html.erb
```

We're going to split the content inside the `if else` statements into partials. Create a new `collapsible_elements` directory inside the `navigation` directory.

```
app/views/layouts/navigation/collapsible_elements
```

Inside the directory create two files: `_signed_in_links.html.erb` and `_non_signed_in_links.html.erb` . Now cut the content from `_collapsible_elements.html.erb` file's `if else` statements and paste it to the corresponding partials. The partials should look like this:

```
<li class="dropdown pc-menu">
  <a id="user-settings" class="dropdown-toggle" data-t
    <span id="user-name"><%= current_user.name %></spa
    <span class="caret"></span>
  </a>

  <ul class="dropdown-menu" role="menu">
    <li><%= link_to 'Edit Profile', edit_user_registra
    <li><%= link_to 'Log out', destroy_user_session_pa
  </ul>
</li>

```

layouts/navigation/collapsible_elements/_signed_in_links.html.erb

```
<li ><%= link_to 'Login', login_path %></li>
<li ><%= link_to 'Signup', signup_path %></li>
```

layouts/navigation/collapsible_elements/_non_signed_in_links.html.erb

Now inside the `_collapsible_elements.html.erb` file, instead of `if else` statements, add the `render` method with the `collapsible_links_partial_path` helper method as an argument. The file should look like this

```
<!-- Collect the nav links, forms, and other content fo
<div class="collapse navbar-collapse navbar-right" id="
  <ul class="nav navbar-nav ">
    <%= render collapsible_links_partial_path %>
  </ul>
```

layouts/navigation/_collapsible_elements.html.erb

`collapsible_links_partial_path` is the method we are going to define inside the `NavigationHelper`. Open `navigation_helper.rb`

```
app/helpers/navigation_helper.rb
```

and define the method inside the module. The `navigation_helper.rb` file should look like this:

```
module NavigationHelper

  def collapsible_links_partial_path
    if user_signed_in?
      'layouts/navigation/collapsible_elements/signed_
    else
      'layouts/navigation/collapsible_elements/non_sig
    end
```

app/helpers/navigation_helper.rb

The defined method is pretty straightforward. If a user is signed in, return a corresponding partial's path. If a user is not signed in, return another partial's path.

We've created our first helper method and extracted logic from views to a helper method. We're going to do this for the rest of the guide, whenever we encounter logic inside a view file. By doing this we're making a favor to ourselves, testing and managing the app becomes much easier.

The app should look and function the same.

Commit the changes.

```
git add -A
git commit -m "Configure and create helpers

- Change include_all_helpers config to false
- Split the _collapsible_elements.html.erb file's content
into
  partials and extract logic from the file into partials"
```

Merge the `helpers` branch with the `master`

```
git checkout master
git merge
helpershttps://gist.github.com/domagude/419bba70cb97e27f4ea0
4fe37820194a#file-rails_helper-rb
```

Testing

At this point the application has some functionality. Even thought there aren't many features yet, but we already have to spend some time by manually testing the app if we want to make sure that everything works. Imagine if the application had 20 times more features than it has now. What a frustration would be to check that everything works fine, every time we did code changes. To avoid this frustration and hours of manual testing, we'll implement automated tests.

Before diving into tests writing, allow me to introduce you how and what I test. Also you can read through A Guide to Testing Rails Applications to get familiar with default Rails testing techniques.

What I use for testing

- **Framework:** RSpec
  When I started testing my Rails apps, I used the default Minitestframework. Now I use RSpec. I don't think there's a good or a bad choice here. Both frameworks are great. I think it depends on a personal preference, which framework to use. I've heard that RSpec is a popular choice among Rails community, so I've decided to give it a shot. Now I am using it most of the time.

- **Sample data:** factory_girl
  Again, at first I tried the default Rails way—fixtures, to add sample data. I've found that it's a different case than it is with testing frameworks. Which testing framework to choose is probably a personal preference. In my opinion it's not the case with sample data. At first fixtures were fine. But I've noticed that after apps become larger, controlling sample data with fixtures becomes tough. Maybe I used it in a wrong way. But with factories everything was nice and peaceful right away. No matter if an app is smaller or bigger—the effort to set sample data is the same.

- **Acceptance tests:** Capybara
  By default Capybara uses rack_test driver. Unfortunately, this driver doesn't support JavaScript. Instead of the default Capybara's driver, I chose to use poltergeist. It supports JavaScript and in my case it was the easiest driver to set up.

What I test

I test all logic which is written by me. It could be:

- Helpers
- Models
- Jobs
- Design Patterns
- Any other logic written by me

Besides logic, I wrap my app with acceptance tests using Capybara, to make sure that all app's features are working properly by simulating a user's interaction. Also to help my simulation tests, I use request tests to make sure that all requests return correct responses.

That's what I test in my personal apps, because it fully satisfies my needs. Obviously, testing standards could be different from person to person and from company to company.

Controllers, views and gems weren't mentioned, why? As many Rails developers say, controllers and views shouldn't contain any logic. And I agree with them. In this case there isn't much to test then. In my opinion, user simulation tests are enough and efficient for views and controllers. And gems are already tested by their creators. So I think

that simulation tests are enough to make sure that gems work properly too.

How I test

Of course I try to use TDD approach whenever is possible. Write a test first and then implement the code. In this case the development flow becomes more smoother. But sometimes you aren't sure how the completed feature is going to look like and what kind of output to expect. You might be experimenting with the code or just trying different implementation solutions. So in those cases, test first and implementation later approach doesn't really work.

Before (sometimes after, as discussed above) every piece of logic I write, I write an isolated test for it a.k.a. unit test. To make sure that every feature of an app works, I write acceptance (user simulation) tests with Capybara.

Set up a test environment

Before we write our first tests, we have to configure the testing environment.

Open the `Gemfile` and add those gems to the test group

```
gem 'rspec-rails', '~> 3.6'
gem 'factory_girl_rails'
gem 'rails-controller-testing'
gem 'headless'
gem 'capybara'
gem 'poltergeist'
gem 'database_cleaner'
```

As discussed above, `rspec` gem is a testing framework, `factory_girl` is for adding sample data, `capybara` is for simulating a user's interaction with the app and `poltergeist` driver gives the JavaScript support for your tests.

You can use another driver which supports JavaScript if it's easier for you to set up. If you decide to use `poltergeist` gem, you will need PhantomJS installed. To install PhantomJS read poltergeist docs.

`headless` gem is required to support headless drivers. `poltergeist` is a headless driver, that's why we need this gem. `rails-controller-`

`testing` gem is going to be required when we will test requests and responses with the requests specs. More on that later.

`database_cleaner` is required to clean the test database after tests where JavaScript was executed. Normally the test database cleans itself after each test, but when you test features which has some JavaScript, the database doesn't clean itself automatically. It might change in the future, but at the moment, of writing this tutorial, after tests with JavaScript are executed, the test database isn't cleaned automatically. That's why we have to manually configure our test environment to clean the test database after each JavaScript test too. We'll configure when to run the `database_cleaner` gem in just a moment.

Now when the purpose of these gems is covered, let's install them by running:

```
bundle install
```

To initialize the `spec` directory for the RSpec framework run the following:

```
rails generate rspec:install
```

Generally speaking, spec means a single test in RSpec framework. When we run our specs, it means that we run our tests.

If you look inside the `app` directory, you will notice a new directory called `spec` . This is where we're going to write tests. Also you may have noticed a directory called `test` . This is where tests are stored when you use a default testing configuration. We won't use this directory at all. You can simply remove it from the project c(x_X)b.

As mentioned above, we have to set up the `database_cleaner` for the tests which include JavaScript. Open the `rails_helper.rb` file

```
spec/rails_helper.rb
```

Change this line

```
config.use_transactional_fixtures = true
```

to

```
config.use_transactional_fixtures = false
```

and below it add the following code:

spec/rails_helper.rb

I took this code snippet from this tutorial.

The last thing we've to do is to add some configurations. Inside the `rails_helper.rb` file's configurations, add the following lines

```
require 'capybara/poltergeist'
require 'factory_girl_rails'
require 'capybara/rspec'

config.include Devise::Test::IntegrationHelpers, type
config.include FactoryGirl::Syntax::Methods
Capybara.javascript_driver = :poltergeist
```

spec/rails_helper.rb

Let's breakdown the code a little bit.

With `require` methods we load files from the new added gems, so we could use their methods below.

```
config.include Devise::Test::IntegrationHelpers, type:
:feature
```

This configuration allows us to use `devise` methods inside `capybara` tests. How did I come up with this line? It was provided inside the Devise docs.

```
config.include FactoryGirl::Syntax::Methods
```

This configuration allows to use `factory_girl` gem's methods. Again, I found this configuration inside the gem's documentation.

```
Capybara.javascript_driver = :poltergeist
Capybara.server = :puma
```

Those two configurations are required in order to be able to test JavaScript with `capybara`. Always read the documentation first, when you want to implement something you don't know how to.

The reason why I introduced you with most of the testing gems and configurations at once and not gradually, once we meet a particular problem, is to give you a clear picture what I use for testing. Now you can always come back to this section and check majority of the configurations in one place. Rather than jumping from one place to another and putting gems with configurations like puzzle pieces together.

Let's commit the changes and finally get our hands dirty with tests.

```
git add -A
git commit -m "
Set up the testing environment

- Remove test directory
- Add and configure rspec-rails, factory_girl_rails,
  rails-controller-testing, headless, capybara, poltergeist,
  database_cleaner gems"
```

Helper specs

About each type of specs (tests), you can find general information by reading rspec docs and its gem docs. Both are pretty similar, but you can find some differences between each other.

Create and switch to a new branch:

```
git checkout -b specs
```

So far we've created only one helper method. Let's test it.

Navigate to `spec` directory anhttps://gist.github.com/domagude/3c42ba6ccf31bf1c50588c59277a9146#file-navigation_helper_spec-rbd create a new directory called `helpers` .

```
spec/helpers
```

Inside the directory, create a new file `navigation_helper_spec.rb`

```
spec/helpers/navigation_helper_spec.rb
```

Inside the file, write the following code:

```ruby
require 'rails_helper'

RSpec.describe NavigationHelper, :type => :helper do

```

spec/helpers/navigation_helper_spec.rb

`require 'rails_helper'` gives us an access to all testing configurations and methods. `:type => :helper` treats our tests as helper specs and provides us with specific methods.

That's how the `navigation_helper_spec.rb` file should look like when the `collapsible_links_partial_path` method is tested.

```
require 'rails_helper'

RSpec.describe NavigationHelper, :type => :helper do

  context 'signed in user' do
    before(:each) { helper.stub(:user_signed_in?).and_

    context '#collapsible_links_partial_path' do
      it "returns signed_in_links partial's path" do
        expect(helper.collapsible_links_partial_path).
          eq 'layouts/navigation/collapsible_elements/
        )
      end
    end
  end

  context 'non-signed in user' do
    before(:each) { helper.stub(:user_signed_in?).and_

```

spec/helpers/navigation_helper_spec.rb

To learn more about the `context` and `it`, read the basic structure docs. Here we test two cases—when a user is logged in and when a user is not logged in. In each context of `signed in user` and `non-signed in user`, we have before hooks. Inside the corresponding context, those hooks (methods) run before each our tests. In our case, before each test we run the stub method, so the `user_signed_in?` returns whatever value we tell it to return.

And finally, with the expect method we check that when we call `collapsible_links_partial_path` method, we get an expected return value.

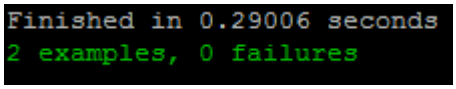
To run all tests, simply run:

```
rspec spec
```

To run specifically the `navigation_helper_spec.rb` file, run:

```
rspec spec/helpers/navigation_helper_spec.rb
```

If the tests passed, the output should look similar to this:



Commit the changes.

```
git add -A
git commit -m "Add specs to NavigationHelper's
collapsible_links_partial_path method"
```

Factories

Next, we'll need some sample data to perform our tests. `factory_girl` gem gives us ability to add sample data very easily, whenever we need it. Also it provides a good quality docs, so it makes the overall experience pretty pleasant. The only object we can create with our app so far is the `User` . To define the user factory, create a `factories` directory inside the `spec` directory.

```
spec/factories
```

Inside the `factories` directory create a new file `users.rb` and add the following code:

```
1  FactoryGirl.define do
2    factory :user do
3      sequence(:name) { |n| "test#{n}" }
4      sequence(:email) { |n| "test#{n}@test.com" }
5      password '123456'
6      password_confirmation '123456'
```

spec/factories/users

Now within our specs, we can easily create new users inside the test database, whenever we need them, using `factory_girl` gem's methods. For the comprehensive guide how to define and use factories, checkout the `factory_girl` gem's docs.

Our defined factory, `user`, is pretty straightforward. We defined the values, `user` objects will have. Also we used the `sequence` method. By reading docs, you can see that with every additional `User` record, `n` value gets incremented by one. I.e. the first created user's name is going to be `test0`, the second one's `test1`, etc.

Commit the changes.

```
git add -A
git commit -m "add a users factory"
```

Feature specs

In the feature specs we write code which simulates a user's interaction with an app. Feature specs are powered by the `capybara` gem.

Good news is that we've everything set up and ready to write our first feature specs. We're going to test the login, logout and signup functionalities.

Inside the `spec` directory, create a new directory called `features`.

```
spec/features
```

Inside the `features` directory, create another directory called `user`.

```
spec/features/user
```

Inside the `user` directory, create a new file called `login_spec.rb`

```
spec/features/user/login_spec.rb
```

That's how the login test looks like:

```
require "rails_helper"

RSpec.feature "Login", :type => :feature do
  let(:user) { create(:user) }

  scenario 'user navigates to the login page and succe
    user
    visit root_path
    find('nav a', text: 'Login').click
    fill_in 'user[email]', with: user.email
    fill_in 'user[password]', with: user.password
```

spec/features/user/login_spec.rb

With this code we simulate a visit to the login page, starting from the home page. Then we fill the form and submit it. Finally, we check if we have the `#user-settings` element on the navigation bar, which is available only for signed in users.

`feature` and `scenario` are part of the Capybara's syntax. `feature` is the same as `context` / `describe` and `scenario` is the same as `it` . More info you can find in Capybara's docs, Using Capybara With Rspec.

`let` method allows us to write memorized methods which we could use across all specs within the context, the method was defined.

Here we also use our created `users` factory and the `create` method, which comes with the `factory_girl` gem.

`js: true` argument allows to test functionalities which involves JavaScript.

As always, to see if a test passes, run a specific file. In this case it is the `login_spec.rb` file:

```
rspec spec/features/user/login_spec.rb
```

Commit the changes.

```
git add -A
git commit -m "add login feature specs"
```

Now we can test the logout functionality. Inside the `user` directory, create a new file named `logout_spec.rb`

```
spec/features/user/logout_spec.rb
```

The implemented test should look like this:

spec/features/user/logout_spec.rb

The code simulates a user clicking the logout button and then expects to see non-logged in user's links on the navigation bar.

`sign_in` method is one of the Devise helper methods. We have included those helper methods inside the `rails_helper.rb` file previously.

Run the file to see if the test passes.

Commit the changes.

```
git add -A
git commit -m "add logout feature specs"
```

The last functionality we have is ability to sign up a new account. Let's test it. Inside the `user` directory create a new file named `sign_up_spec.rb`. That's how the file with the test inside should look like:

```
require "rails_helper"

RSpec.feature "Sign up", :type => :feature do
  let(:user) { build(:user) }

  scenario 'user navigates to sign up page and success
    visit root_path
    find('nav a', text: 'Signup').click
    fill_in 'user[name]', with: user.name
    fill_in 'user[email]', with: user.email
    fill_in 'user[password]', with: user.password
    fill in 'user[password confirmation]', with: user.
```

spec/features/user/sign_up_spec.rb

We simulate a user navigating to the signup page, filling the form, submitting the form and finally, we expect to see the `#user-settings` element which is available only for logged in users.

Here we use the Devise's `build` method instead of `create`. This way we create a new object without saving it to the database.

We can run the whole test suite and see if all tests pass successfully.

```
rspec spec
```

```
Finished in 3.73 seconds
5 examples, 0 failures
```

Commit the changes.

```
git add -A
git commit -m "add sign up features specs"
```

We're done with our first tests. So let's merge the `specs` branch with the `master`.

```
git checkout master
git merge specs
```

```
```

Specs branch isn't needed anymore. Delete it q__o.

```
git branch -D specs
```

Main feed

On the the home page we're going to create a posts feed. This feed is going to display all type of posts in a card format.

Start by creating a new branch:

```
git checkout -b main_feed
```

Generate a new model called `Post`.

```
rails g model post
```

Then we'll need a `Category` model to categorize the posts:

```
rails g model category
```

Now let's create some associations between `User`, `Category` and `Post` models.

Every post is going to belong to a category and its author (user). Open the models' files and add the associations.

```
class Post < ApplicationRecord
  belongs_to :user
  belongs_to :category
end
```

```
class User < ApplicationRecord
  ...
  has_many :posts, dependent: :destroy
end
```

```
class Category < ApplicationRecord
  has_many :posts
end
```

The `dependent: :destroy` argument says, when a user gets deleted, all posts what the user has created will be deleted too.

Now we've to define data columns and associations inside the migrations files.

db/migrate/CREATION_DATE_create_posts.rb

db/migrate/CREATION_DATE_create_categories.rb

Now run the migration files:

```
rails db:migrate
```

Commit the changes:

```
git add -A
git commit -m "
- Generate Post and Category models.
- Create associations between User, Post and Category
models.
- Create categories and posts database tables."
```

Specs

We can test the newly created models. Later we'll need sample data for the tests. Since a post belongs to a category, we also need sample data for categories to set up the associations.

Create a `category` factory inside the `factories` directory.

```
spec/factories/categories.rb
```

spec/factories/categories.rb

Create a `post` factory inside the `factories` directory

```
spec/factories/posts.rb
```

spec/factories/posts.rb

As you see, it's very easy to set up an association for factories. All we had to do to set up `user` and `category` associations for the `post` factory, is to write factories' names inside the `post` factory.

Commit the changes.

```
git add -A
git commit -m "Add post and category factories"
```

For now we'll only test the associations, because that's the only thing we wrote yet inside the models.

Open the `post_spec.rb`

```
spec/models/post_spec.rb
```

Add specs for the associations, so the file should look like this:

spec/models/post_spec.rb

We use the `described class` method to get the current context's class. Which is basically the same as writing `Post` in this case. Then we use

`reflect on association` method to check that it returns a correct association.

Do the same for other models.

spec/models/category_spec.rb

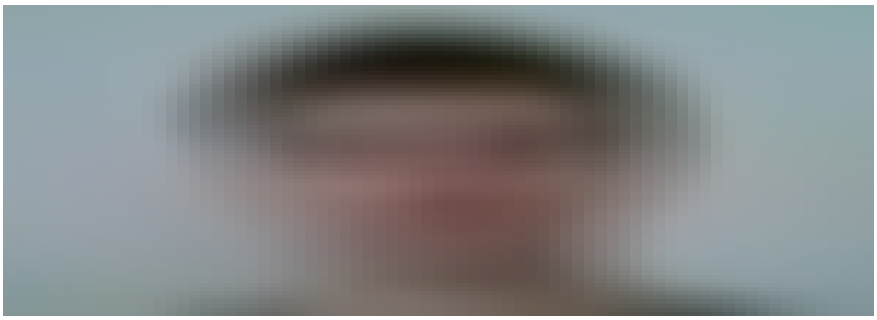
spec/models/user_spec.rb

Commit the changes.

```
git add -A
git commit -m "Add specs for User, Category, Post models' associations"
```

Home page layout

Currently the home page has nothing inside, only the dummy text "Home page". It's time to create its layout with bootstrap. Open the home page's view file `views/pages/index.html.erb` and replace the file's content with the following code to create the page's layout:



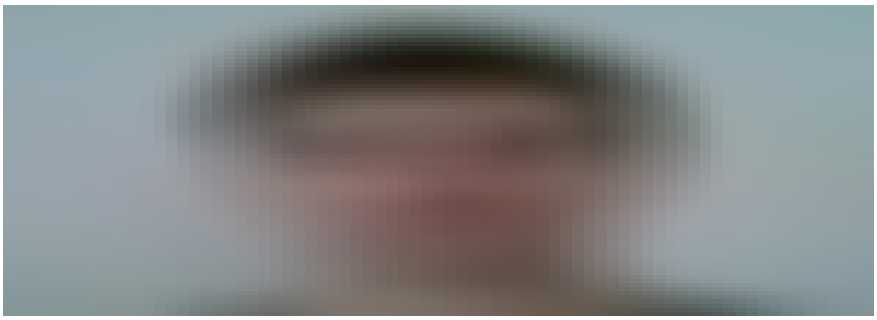
views/pages/index.html.erb

Now add some CSS to define elements' style and responsive behavior.

Inside the `stylesheets/partials` directory create a new file `home_page.scss`

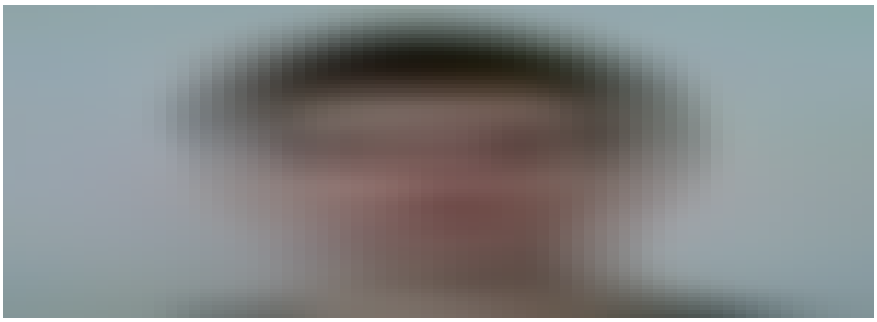
```
assets/stylesheets/partials/home_page.scss
```

In the file add the following CSS:



assets/stylesheets/partials/home_page.scss

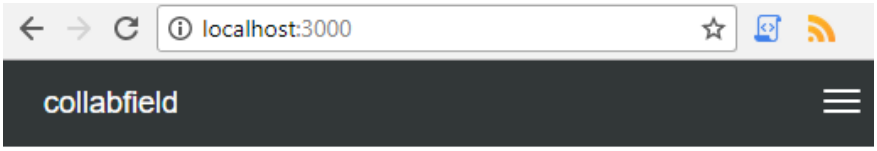
Inside the `mobile.scss` file's `max-width: 767px` media query add:



assets/stylesheets/responsive/mobile.scss

Now the home page should look like this on bigger screens

and like this on the smaller screens



Commit the changes.

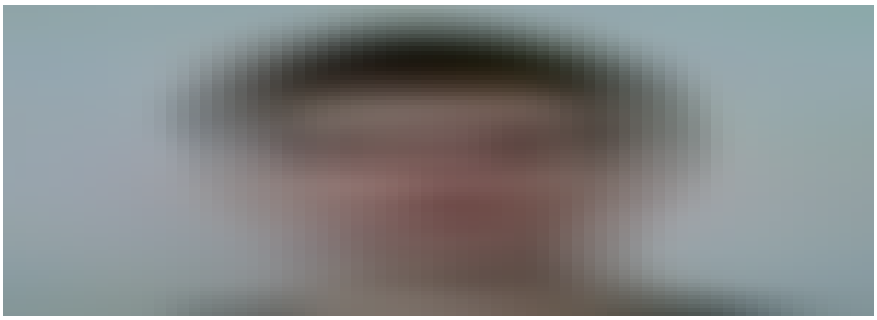
```
git add -A
git commit -m "
- Add the bootstrap layout to the home page
- Add CSS to make home page layout's stylistic and
responsive design changes"
```

Seeds

To display posts on the home page, at first we need to have them inside the database. Creating data manually is boring and time consuming. To automate this process, we'll use seeds. Open the `seeds.rb` file.

```
db/seeds.rb
```

Add the following code:



db/seeds.rb

As you see, we create `seed_users`, `seed_categories` and `seed_posts` methods to create `User`, `Category` and `Post` records inside the development database. Also the faker gem is used to generate dummy text. Add `faker` gem to your `Gemfile`

```
gem 'faker'
```

and

```
bundle install
```

To seed data, using the `seeds.rb` file, run a command

```
rails db:seed
```

Commit the changes.

```
git add -A
git commit -m "
- Add faker gem
- Inside the seeds.rb file create methods to generate
  User, Category and Post records inside the development
database"
```

Rendering the posts

To render the posts, we'll need a `posts` directory inside the views.

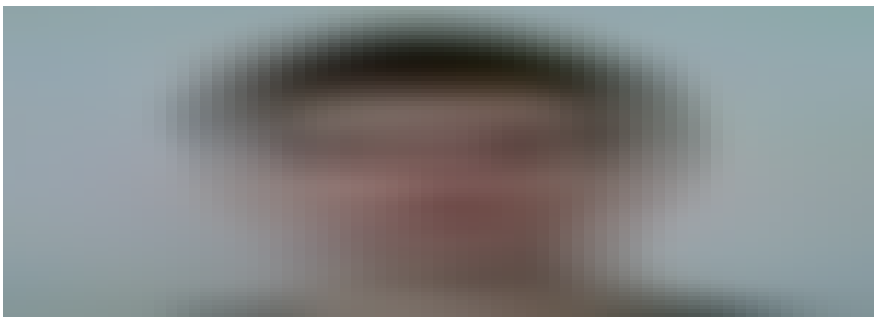
Generate a new controller called `Posts`, so it will automatically create a `posts` directory inside the views too.

```
rails g controller posts
```

Since in our app the `PagesController` is responsible for the homepage, we'll need to query data inside the `pages_controller.rb` file's `index` action. Inside the `index` action retrieve some records from the `posts` table. Assign the retrieved records to an instance variable, so the retrieved objects are going to be available inside the home page's views.

- If you aren't familiar with ruby variables, read this guide.
- If you aren't familiar with retrieving records from the database in Rails, read the Active Record Query Interface guide.

The `index` action should look something like this right now:



controllers/pages_controller.rb

Navigate to the home page's template

```
views/pages/index.html.erb
```

and inside the `.main-content` element add

```
<%= render @posts %>
```

This will render all posts, which were retrieved inside the `index` action. Because `post` objects belong to the `Post` class, Rails automatically tries to render the `_post.html.erb` partial template which is located

```
views/posts/_post.html.erb
```

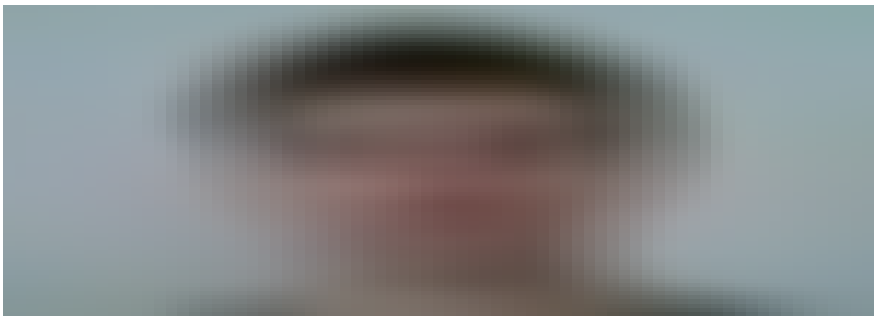
We haven't created this partial file yet, so create it and add the following code inside:

```
1  <div class="col-sm-3 single-post-card" id=<%= post_pat
2    <div class="card">
3      <div class="card-block">
4        <h4 class="post-text">
5          <%= truncate(post.title, :length => 60) %>
6        </h4>
7        <div class="post-content">
8          <div class="posted-by">Posted by <%= post.user
9          <h3><%= post.title %></h3>
10         <p><%= post.content %></p>
11         <%= link_to "I'm interested", post_path(post_i
```

views/posts/_post.html.erb

I've used a bootstrap card component here to achieve the desired style. Then I just stored post's content and its path inside the element. Also I added a link which will lead to the full post.

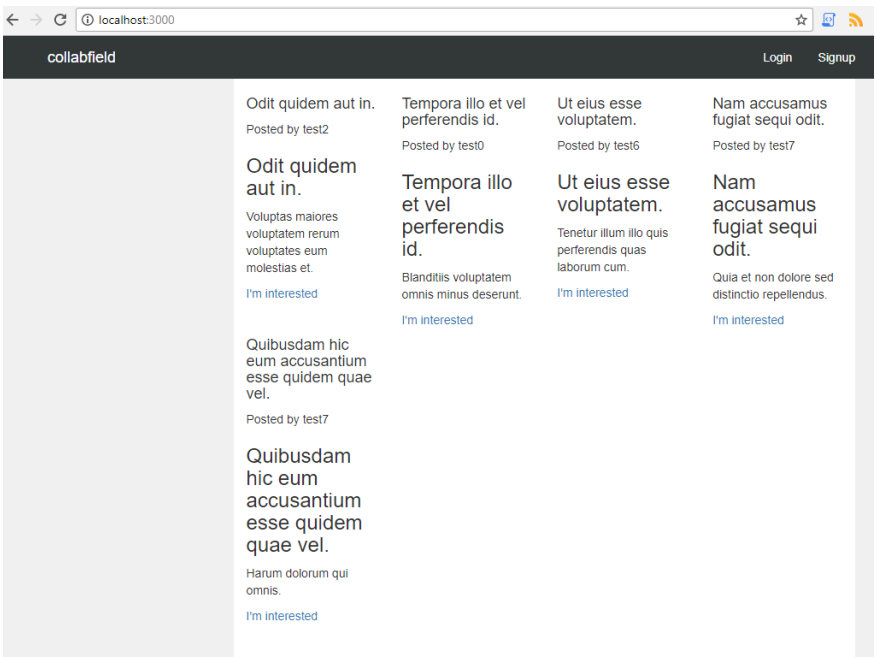
So far we didn't define any routes for posts. We need them right now, so let's declare them. Open the `routes.rb` file and add the following code inside the routes:



routes.rb

Here I've used a resources method to declare routes for `index`, `show`, `new`, `edit`, `create`, `update` and `destroy` actions. Then I've declared some custom `collection` routes to access pages with multiple `Post` instances. These pages are going to be dedicated for separate branches, we'll create them later.

Restart the server and go to http://localhost:3000. You should see rendered posts on the screen. The application should look similar to this:



Commit the changes.

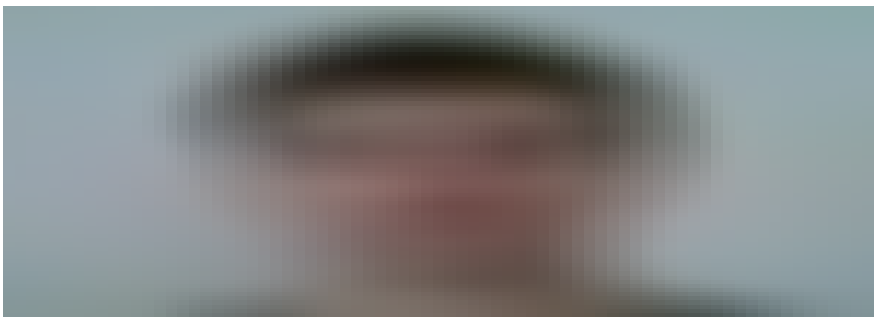
```
git add -A
git commit -m "Display posts on the home page

- Generate Posts controller and create an index action.
  Inside the index action retrieve Post records
- Declare routes for posts
```

```
- Create a _post.html.erb partial inside posts directory
- Render posts inside the home page's main content"
```

To start styling posts, create a new scss file inside the `partials` directory:

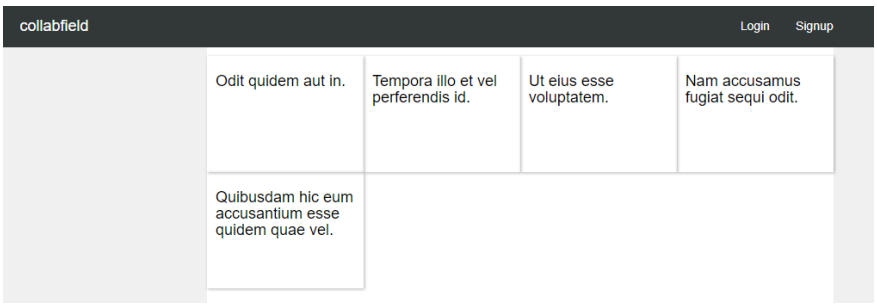
```
assets/stylesheets/partials/posts.scss
```

and inside the file add the following CSS:



assets/stylesheets/partials/posts.scss

The home page should look similar to this:



Commit the change.

```
git add -A
git commit -m "Create a posts.scss file and add CSS to it"
```

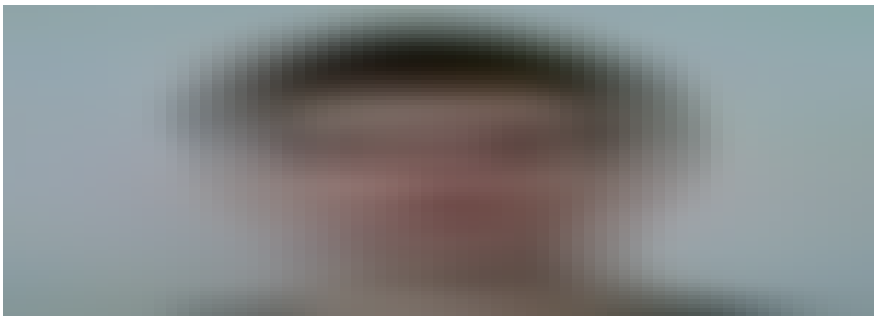
Styling with JavaScript

Currently the site's design is pretty dull. To create contrast, we're going to color the posts. But instead of just coloring it with CSS, let's color them with different color patterns every time a user refreshes the website. To do that we'll use JavaScript. It's probably a silly idea, but it's fun c(o_u)?

Navigate to the `javascripts` directory inside your `assets` and create a new directory called `posts`. Inside the directory create a new file called `style.js`. Also if you want, you can delete by default generated `.coffee`, files inside the `javascripts` directory. We won't use CoffeeScript in this tutorial.

```
assets/javascripts/posts/style.js
```

Inside the `style.js` file add the following code.



assets/javascripts/posts/style.js

With this piece of code we randomly set one of two style modes when a browser gets refreshed, by adding attributes to posts. One style has colored borders only, another style has solid color posts. With every page change and browser refresh we also recolor posts randomly too. Inside the `randomColorSet()` function you can see predefined color schemes.

`mouseenter` and `mouseleave` event handlers are going to be needed in the future for posts in specific pages. There posts' style is going to be different than posts' on the home page. When you'll hover on a post, it will slightly change its bottom border's color. You'll see this later.

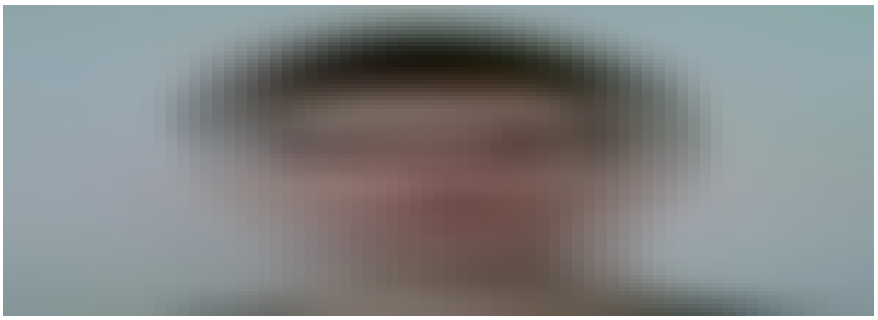
Commit the changes.

```
git add -A
git commit -m "Create a style.js file and add js to create
posts' style"
```

To complement the styling, add some CSS. Open the `posts.scss` file

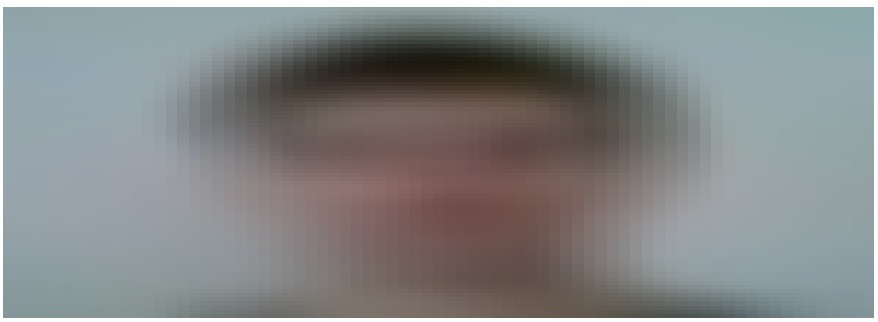
```
assets/stylesheets/partials/posts.scss
```

and add the following CSS:



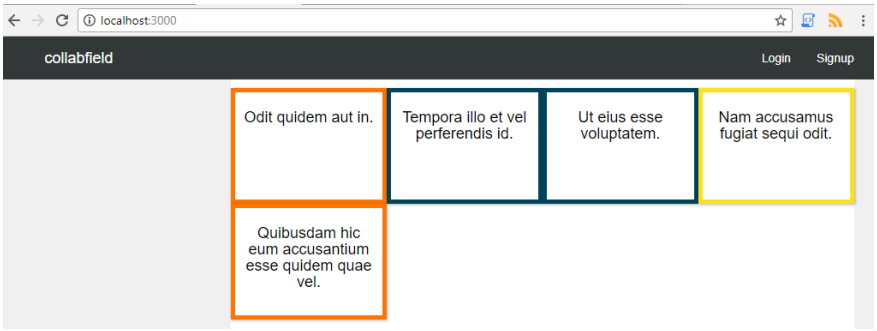
assets/stylesheets/partials/posts.scss

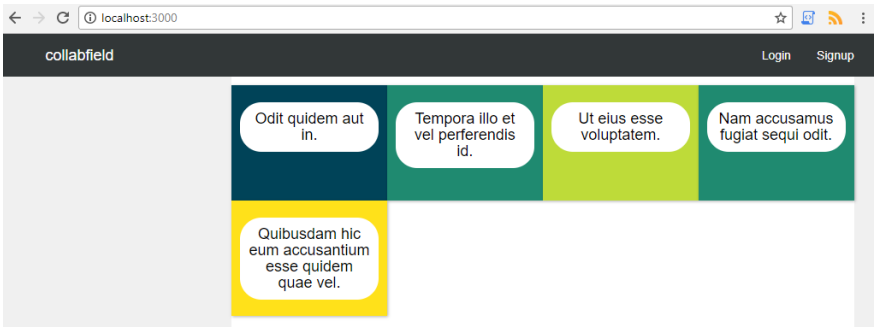
Also inside the `mobile.scss` add the following code to fix too large text issues on smaller screens:

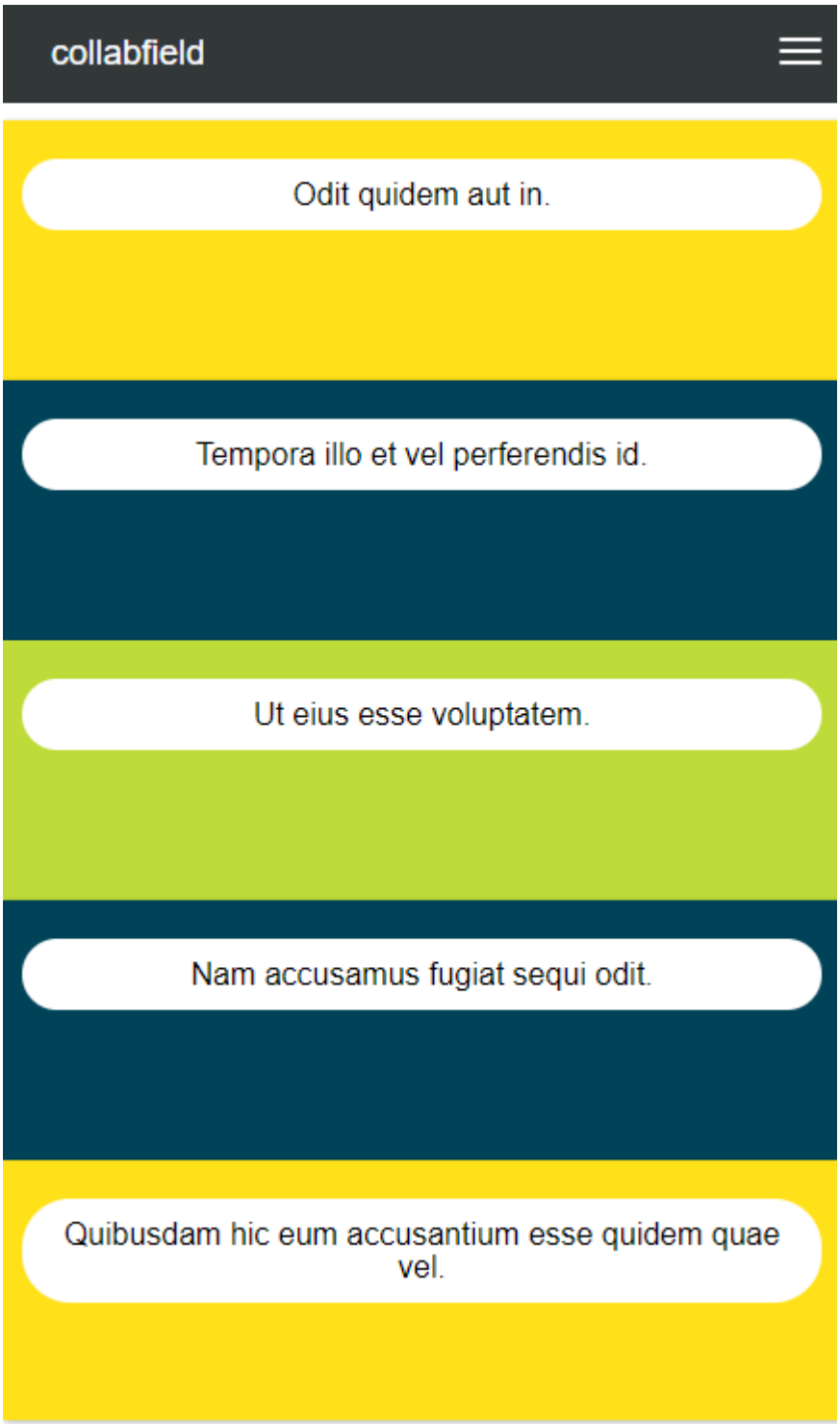


assets/stylesheets/responsive/mobile.scss

The home page should look similar to this right now:


localhost:3000
collabfield
Login
Signup
Odit quidem aut in.
Tempora illo et vel perferendis id.
Ut eius esse voluptatem.
Nam accusamus fugiat sequi odit.
Quibusdam hic eum accusantium esse quidem quae vel.


localhost:3000
collabfield
Login
Signup
Odit quidem aut in.
Tempora illo et vel perferendis id.
Ut eius esse voluptatem.
Nam accusamus fugiat sequi odit.
Quibusdam hic eum accusantium esse quidem quae vel.



Commit the changes

```
git add -A
git commit -m "Add CSS to posts on the home page

- add CSS to the posts.scss file
- add CSS to the mobile.scss to fix too large text issues on
smaller screens"
```

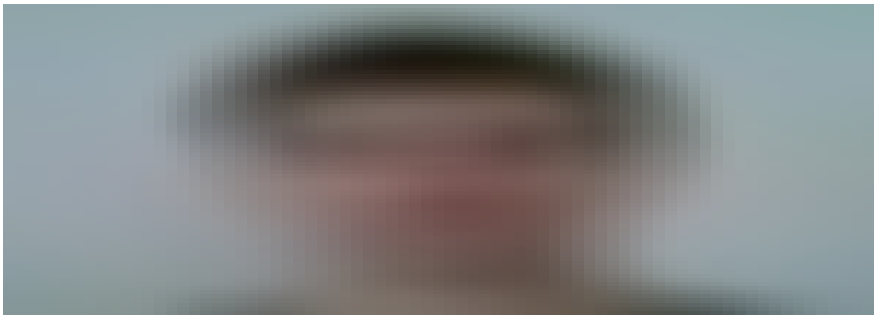
Modal window

I want to be able to click on a post and see its full content, without going to another page. To achieve this functionality I'll use a bootstrap's modal component.

Inside the `posts` directory, create a new partial file `_modal.html.erb`

```
views/posts/_modal.html.erb
```

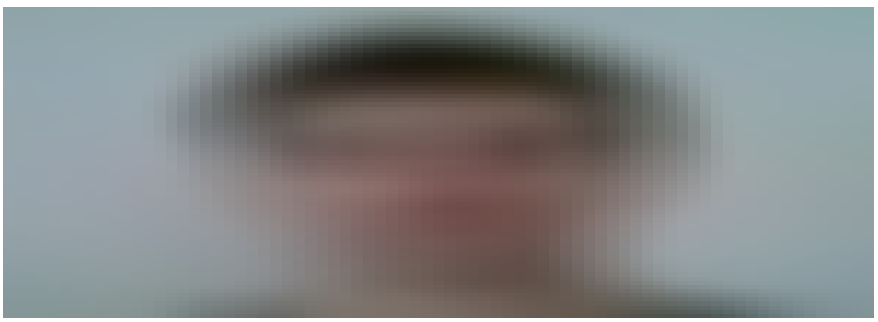
and add the following code:



views/posts/_modal.html.erb

This is just a slightly modified bootstrap's component to accomplish this particular task.

Render this partial at the top of the home page's template.

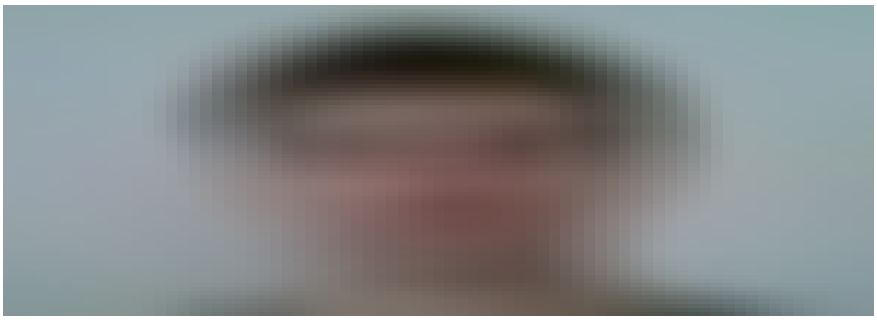


views/pages/index.html.erb

To make this modal window functional, we have to add some JavaScript. Inside the `posts` directory, create a new file `modal.js`

```
assets/javascripts/posts/modal.js
```

Inside the file, add the following code:



assets/javascripts/posts/modal.js

With this js code we simply store selected post's data into variables and fill modal window's elements with this data. Finally, with the last line of code we make the modal window visible.

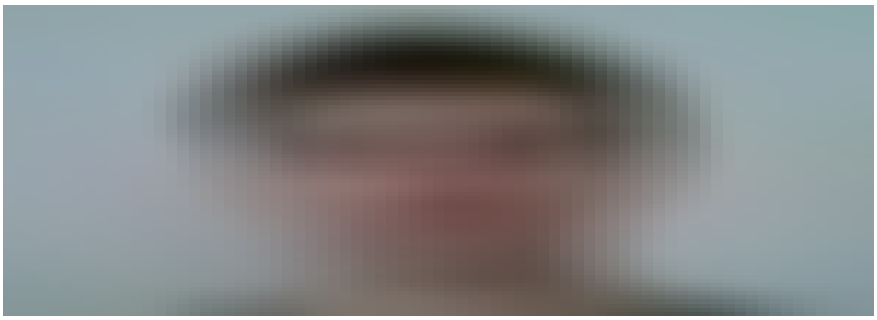
To enhance the modal window's looks, add some CSS. But before adding CSS, let's do a quick management task inside the `stylesheets` directory.

Inside the `partials` directory create a new directory called `posts`

```
assets/stylesheets/partials/posts
```

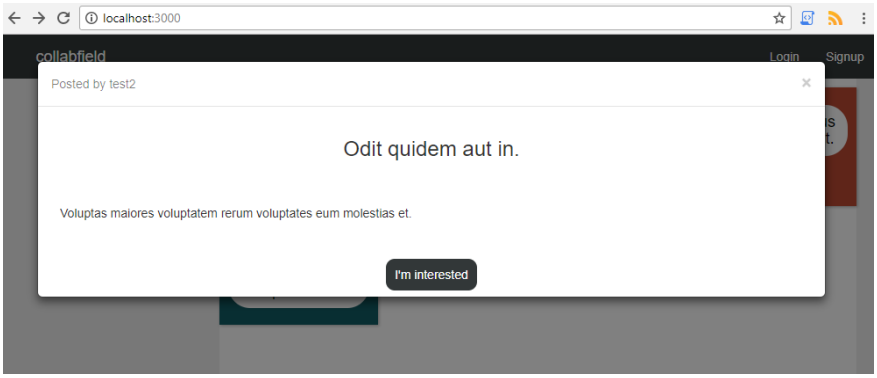
Inside the `posts` directory create a new file `home_page.scss` . Cut all code from the `posts.scss` file and paste it inside the `home_page.scss` file. Delete the `posts.scss` file. We're doing this for a better CSS code management. It's clearer when have few smaller CSS files with a distinguishable purpose, rather than one big file where everything is mashed together.

Also inside the `posts` directory, create a new file `modal.scss` and add the following CSS:



assets/stylesheets/partials/posts/modal.scss

Now when we click on the post, the application should look like this:



Commit the changes.

```
git add -A
git commit -m "Add a popup window to show a full post's
content

- Add bootstrap's modal component to show full post's
content
- Render the modal inside the home page's template
- Add js to fill the modal with post's content and show it
- Add CSS to style the modal"
```

Also merge the `main_feed` branch with the `master`

```
git checkout master
git merge main_feed
```

Get rid of the `main_feed` branch

```
git branch -D main_feed
```

Single post

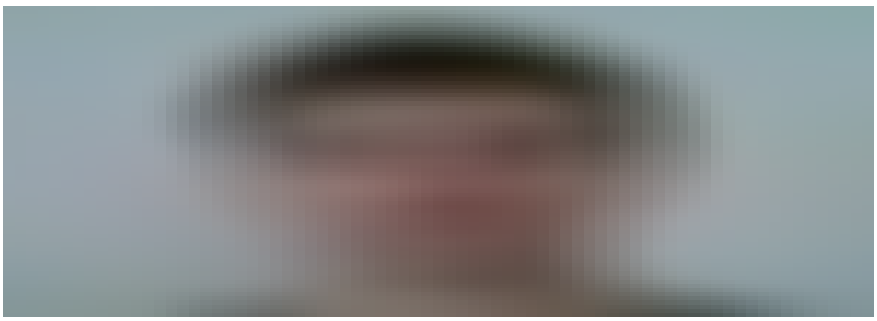
Switch to a new branch

```
git checkout -b single_post
```

Show a single post

If you try to click on the `I'm interested` button, you will get an error. We haven't created a `show.html.erb` template nor we've created a corresponding controller's action. By clicking the button I want to be redirected to a selected post's page.

Inside the `PostsController`, create a `show` action and then query and store a specific post object inside an instance variable:



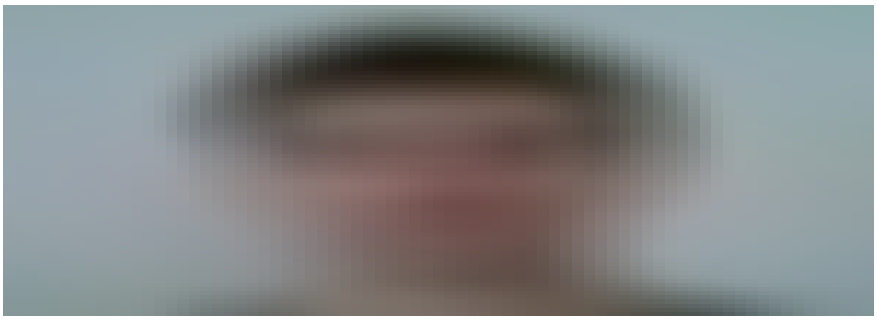
controllers/posts_controller.rb

`I'm interested` button redirects to a selected post. It has a `href` attribute with a path to a post. By sending a `GET` request to get a post, rails calls the `show` action. Inside the `show` action, we've an access to the id param, because by sending a `GET` request to get a specific post, we provided its `id`. I.e. by going to a `/posts/1` path, we would send a request to get a post whose `id` is `1`.

Create a `show.html.erb` template inside the `posts` directory

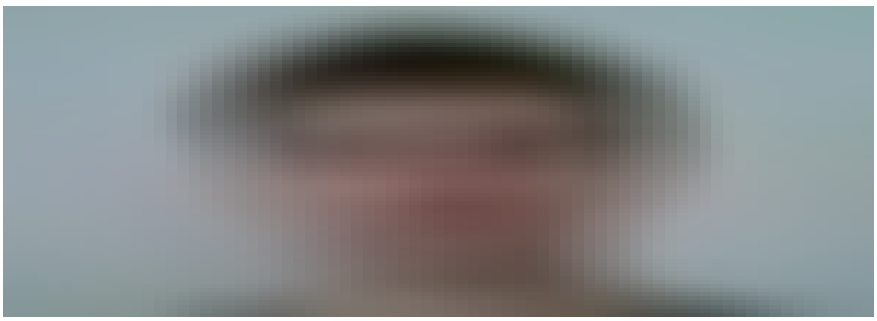
```
views/posts/show.html.erb
```

Inside the file add the following code:



views/posts/show.html.erb

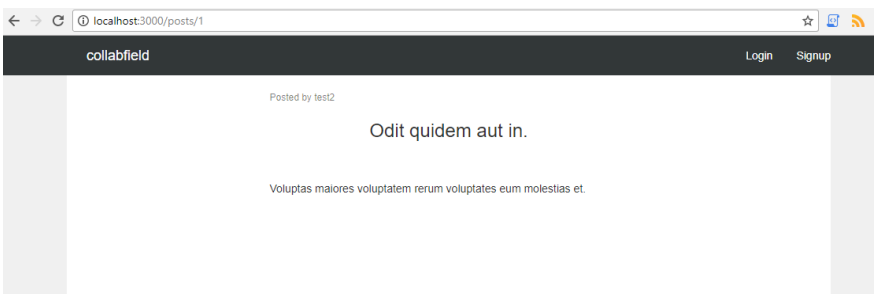
Create a `show.scss` file inside the `posts` directory and add CSS to style the page's look:



assets/stylesheets/partials/posts/show.scss

Here I defined the page's height to be `100vh-50px`, so the page's content is full viewport's height. It allows the container to be colored white across the full browser's height, no matter if there is enough of content inside the element or not. `vh` property means viewport's height, so `100vh` value means that the element is stretched 100% of viewport's height. `100vh-50px` is required to subtract navigation bar's height, otherwise the container would be stretched too much by `50px`.

If you click on the `I'm interested` button now, you will be redirected to a page which looks similar to this:



We'll add extra features to the `show.html.erb` template later. Now commit the changes.

```
git add -A
git commit -m "Create a show template for posts

- Add a show action and query a post to an instance variable
- Create a show.scss file and add CSS"
```

Specs

Instead of manually checking that this functionality, of modal window appearance and redirection to a selected post, works, wrap it all with specs. We're going to use capybara to simulate a user's interaction with the app.

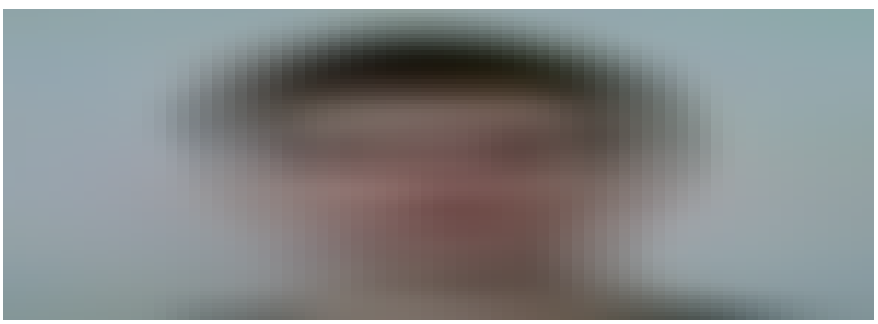
Inside the `features` directory, create a new directory called `posts`

```
spec/features/posts
```

Inside the new directory, create a new file `visit_single_post_spec.rb`

```
spec/features/posts/visit_single_post_spec.rb
```

And add a feature spec inside. The file looks like this:



spec/features/posts/visit_single_post_spec.rb

Here I defined all steps which I would perform manually. I start by going to the home page, click on the post, expect to see the popped up modal window, click on the `I'm interested` button, and finally, expect to be redirected to the post's page and see its content.

By default RSpec matchers `have_selector`, `have_css`, etc. return true if an element is actually visible to a user. So after it was clicked on a post, testing framework expects to see a visible modal window. If you don't care if a user sees an element or not and you just care about an element's presence in the DOM, pass an additional `visible: false` argument.

Try to run the test

```
rspec spec/features/posts/visit_single_post_spec.rb
```

Commit the changes.

```
git add -A
git commit -m "Add a feature spec to test if a user can go
to a
single post from the home page"
```

Merge the `single_post` branch with the `master`.

```
git checkout master
git merge single_post
git branch -D single_post
```

Specific branches

Every post belongs to a particular branch. Let's create specific pages for different branches.

Switch to a new branch

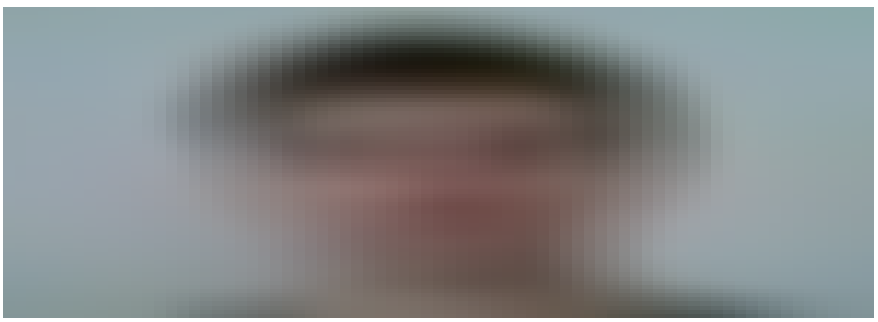
```
git checkout -b specific_branches
```

Home page's side menu

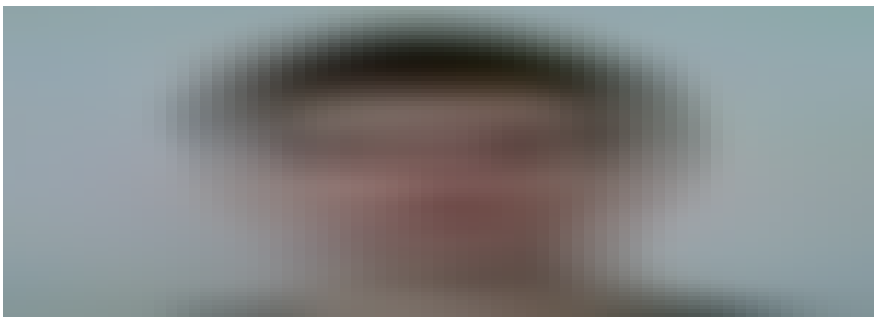
Start by updating the home page's side menu. Add links to specific branches. Open the `index.html.erb` file:

```
views/pages/index.html.erb
```

We are going to put some links inside the `#side-menu` element. Split file's content into partials, otherwise it will get noisy very quickly. Cut `#side-menu` and `#main-content` elements, and paste them into separate partial files. Inside the `pages` directory create an `index` directory, and inside the directory create corresponding partial files to the elements. The files should look like this:

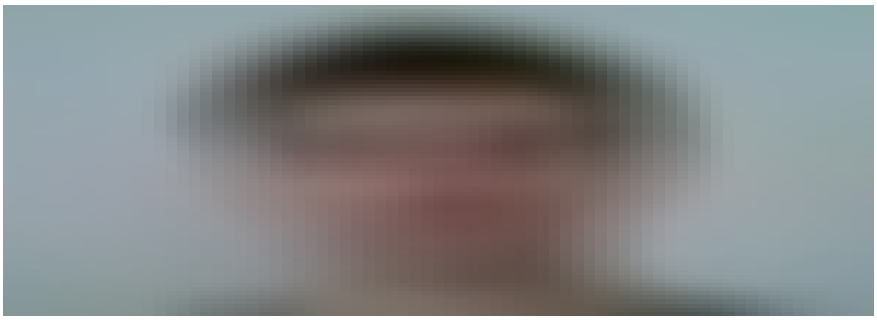


views/pages/index/_side_menu.html.erb



views/pages/index/_main_content.html.erb

Render those partial files inside the home page's template. The file should look like this:

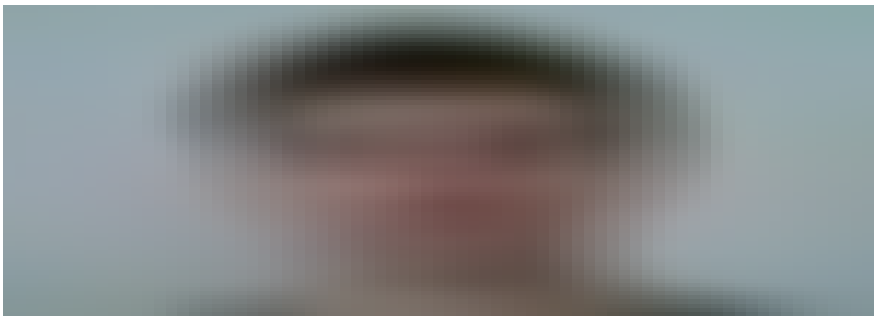


views/pages/index.html.erb

Commit the changes.

```
git add -A
git commit -m "Split home page template's content into
partials"
```

Inside the `_side_menu.html.erb` partial add a list of links, so the file should look like this:



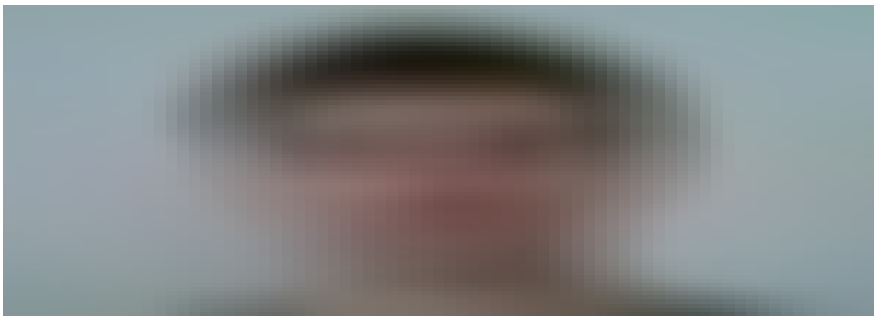
views/pages/index/_side_menu.html.erb

An unordered list was added. Inside the list we render another partial with links. Those links are going to be available for all users, no matter if they are signed in or not. Create this partial file and add the links.

Inside the `index` directory create a `side_menu` directory:

```
views/pages/index/side_menu
```

Inside the directory create a `_no_login_required_links.html.erb` partial with the following code:



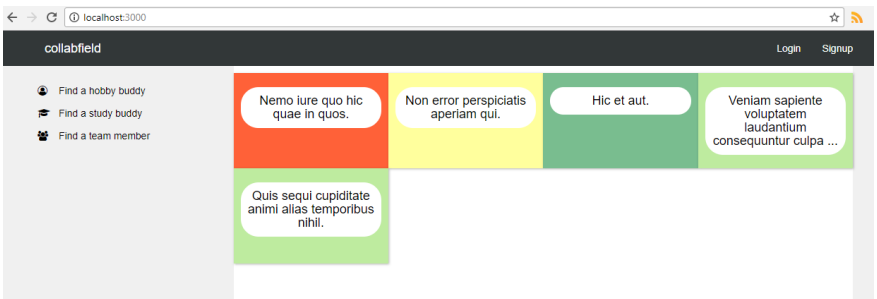
views/pages/index/side_menu/_no_login_required_links.html.erb

Here we simply added links to specific branches of posts. If you are wondering how do we have paths, such as `hobby_posts_path` , etc., look at the `routes.rb` file. Previously we've added nested `collection` routes inside the `resources :posts` declaration.

If you pay attention to `i` elements' attributes, you will notice `fa` classes. With those classes we declare Font Awesome icons. We haven't set up this library yet. Fortunately, it's very easy to set up. Inside the main `application.html.erb` file's `head` element, add the following line

```
<link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/font-
awesome/4.7.0/css/font-awesome.min.css">
```

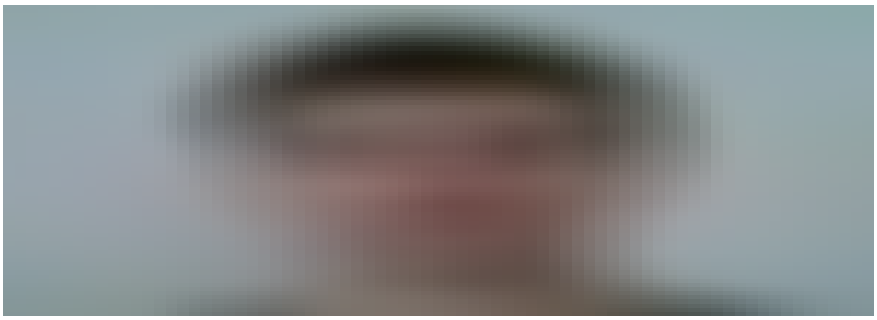
The side menu should be present now.



Commit the changes.

```
git add -A
git commit -m "Add links to the home page's side menu"
```

On smaller screens, where width is between `767px` and `1000px`, bootstrap's container looks unpleasant, it looks overly compressed. So stretch it among those widths. Inside the `mobile.scss` file, add the following code:



assets/stylesheets/responsive/mobile.scss

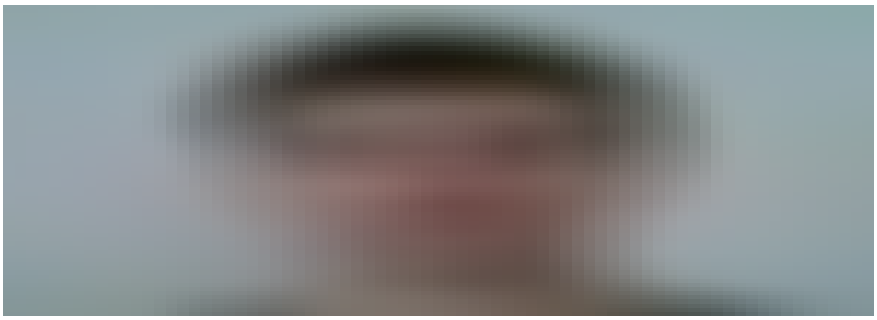
Commit the change.

```
git add -A
git commit -m "set .container width to 100%
when viewport's width is between 767px and 1000px"
```

Branch page

If you try to click on one of those side menu links, you will get an error. We haven't set up actions inside the `PostsController` nor we created any templates for it.

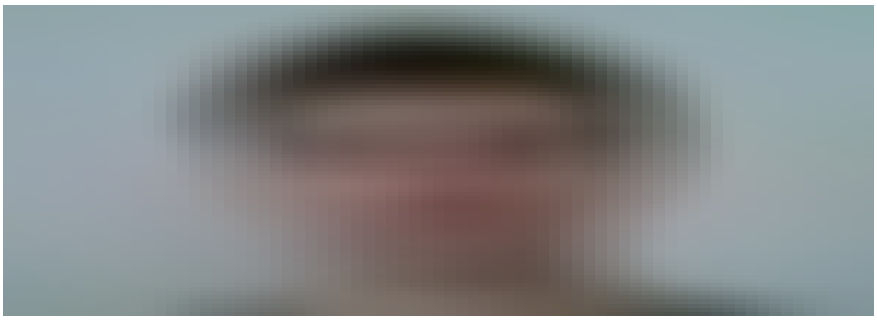
Inside the `PostsController`, define `hobby`, `study`, and `team` actions.



controllers/posts_controller.rb

Inside every action, `posts_for_branch` method is called. This method will return data for the specific page, depending on the action's name.

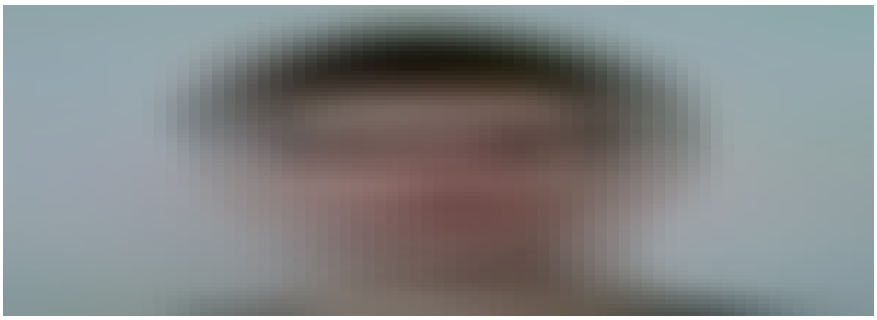
Define the method inside the `private` scope.



contorllers/posts_controller.rb

In the `@categories` instance variable we retrieve all categories for a specific branch. I.e. if you go to the hobby branch page, all categories which belong to the hobby branch will be retrieved.

To get and store posts inside the `@posts` instance variable, `get_posts` method is used and then it is chained with a `paginate` method. `paginate` method comes from will_paginate gem. Let's start by defining the `get_posts` method. Inside the `PostsController`'s `private` scope add:



controllers/posts_controller.rb

Right now `get_posts` method just retrieves any 30 posts, not specific to anything, so we could move on and focus on further development. We'll come back to this method in the near future.

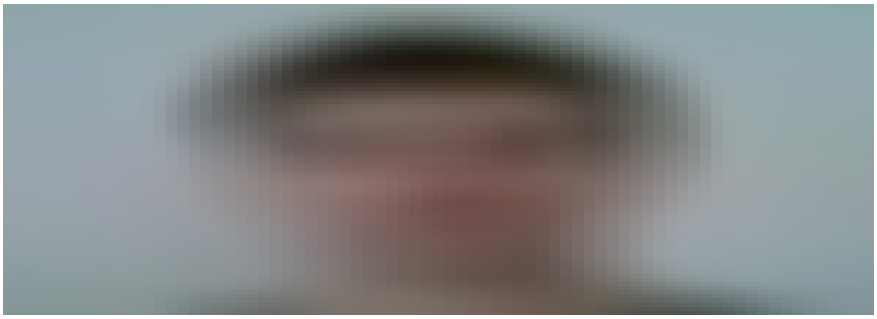
Add the `will_paginate` gem to be able to use pagination.

```
gem 'will_paginate', '~> 3.1.0'
```

run

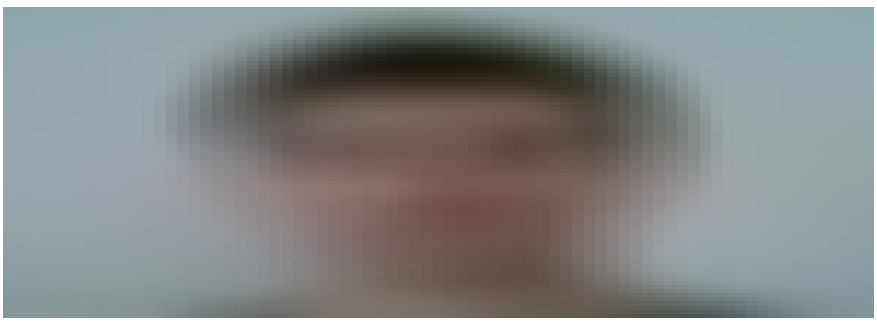
```
bundle install
```

All we miss now is templates. They are going to be similar to all branches, so instead of repeating the code, inside every of those branches, create a partial with a general structure for a branch. Inside the `posts` directory create a `_branch.html.erb` file.



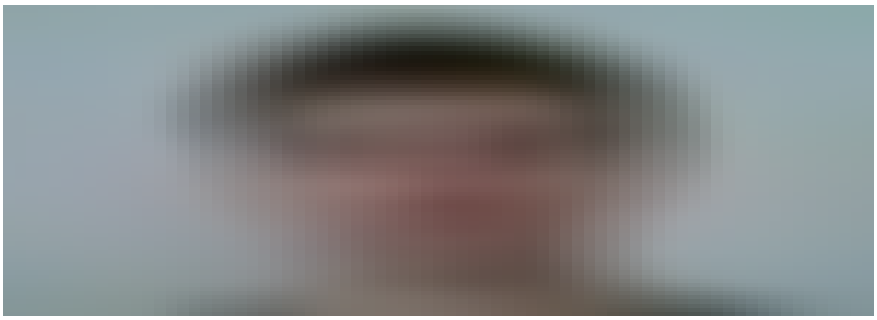
posts/_branch.html.erb

First you see a `page_title` variable being printed on the page. We'll pass this variable as an argument when we'll render the `_branch.html.erb` partial. Next, a `_create_new_post` partial is rendered to display a link, which will lead to a page, where a user could create a new post. Create this partial file inside a new `branch` directory:



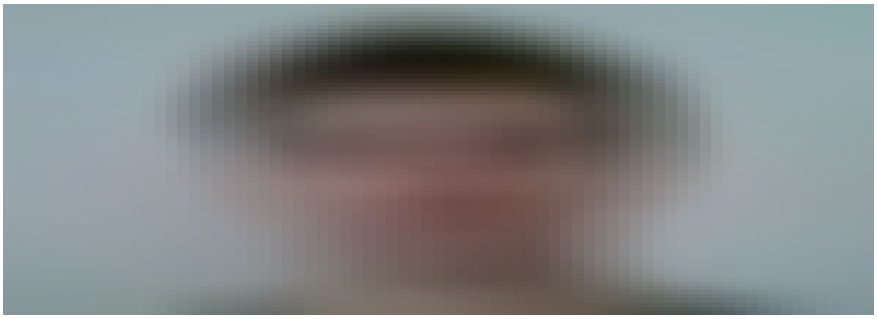
posts/branch/_create_new_post.html.erb

Here we'll use a `create_new_post_partial_path` helper method to determine which partial file to render. Inside the `posts_helper.rb` file, implement the method:

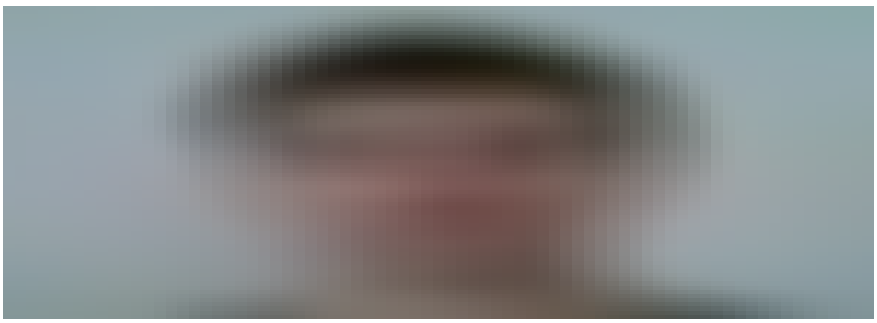


helpers/posts_helper.rb

Also create those two corresponding partials inside a new `create_new_post` directory:

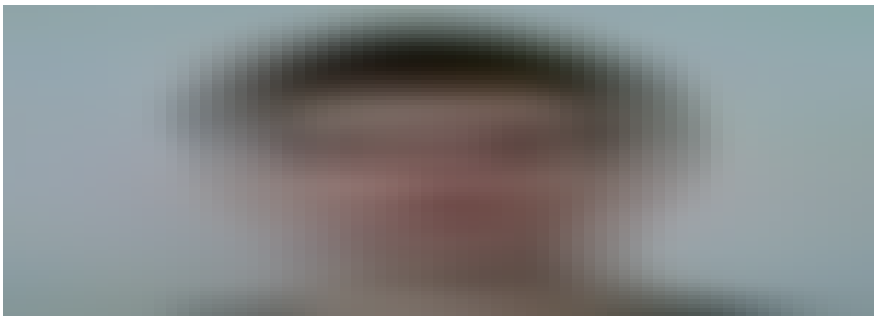


posts/branch/create_new_post/_signed_in.html.erb



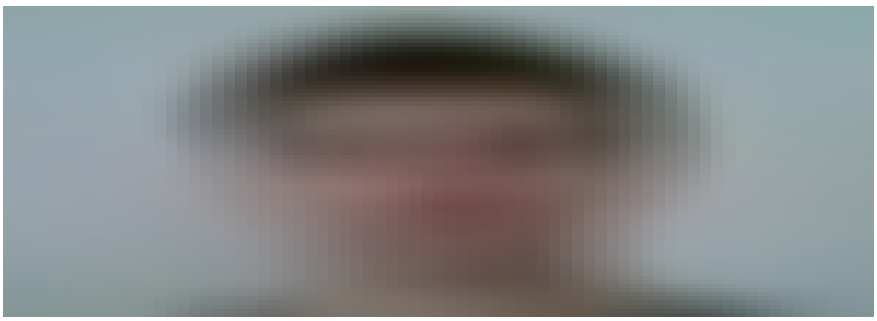
posts/branch/create_new_post/_not_signed_in.html.erb

Next, inside the `_branch.html.erb` file we render a list of categories. Create a `_categories.html.erb` partial file:



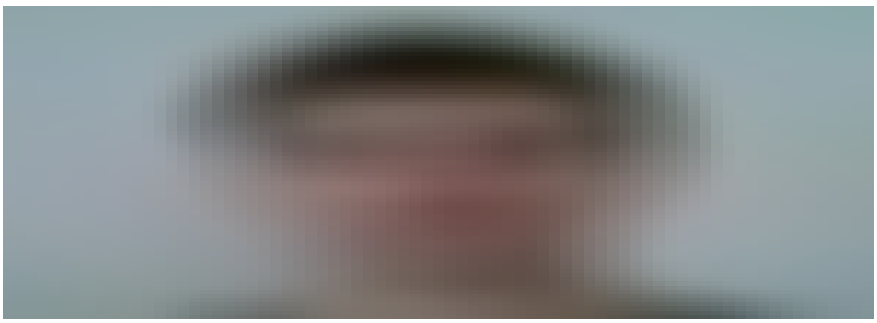
posts/branch/_categories.html.erb

Inside the file, we have a `all_categories_button_partial_path` helper method which determines which partial file to render. Define this method inside the `posts_helper.rb` file:

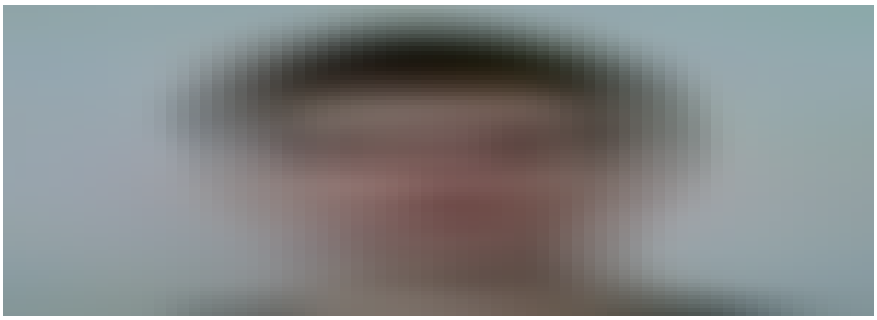


helpers/posts_helper.rb

All categories are going to be selected by default. If the `params[:category]` is empty, it means that none categories were selected by a user, which means that currently the default value `all` is selected. Create the corresponding partial files:



posts/branch/categories/_all_selected.html.erb

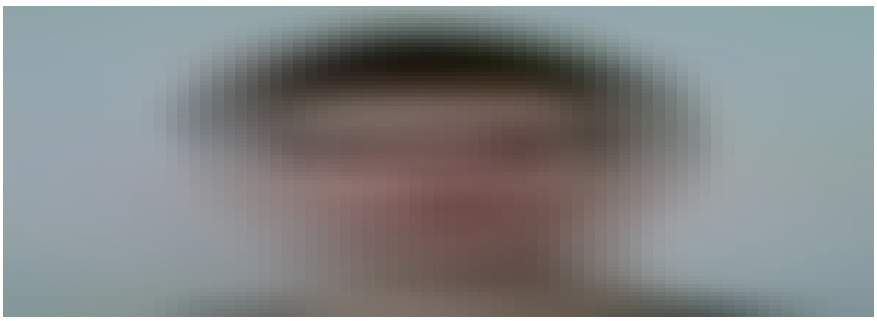


posts/branch/categories/_all_not_selected.html.erb

The send method is used here to call a method by using a string, this allows to be flexible and call methods dynamically. In our case we generate different paths, depending on the current controller's action.

Next, inside the `_branch.html.erb` file we render posts and call the `no_posts_partial_path` helper method. If posts are not found, the method will display a message.

Inside the `posts_helper.rb` add the helper method:



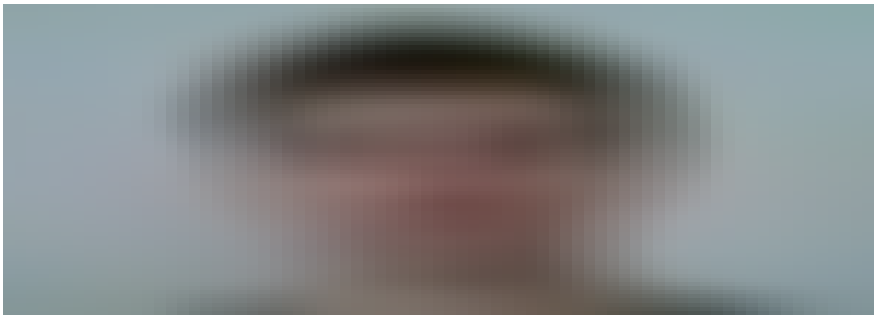
helpers/posts_helper.rb

Here I use a ternary operator, so the code looks a little bit cleaner. If there are any posts, I don't want to show any messages. Since you cannot pass an empty string to the `render` method, I pass a path to an empty partial instead, in occasions where I don't want to render anything.

Create a `shared` directory inside the views and then create an empty partial:

```
views/shared/_empty_partial.html.erb
```

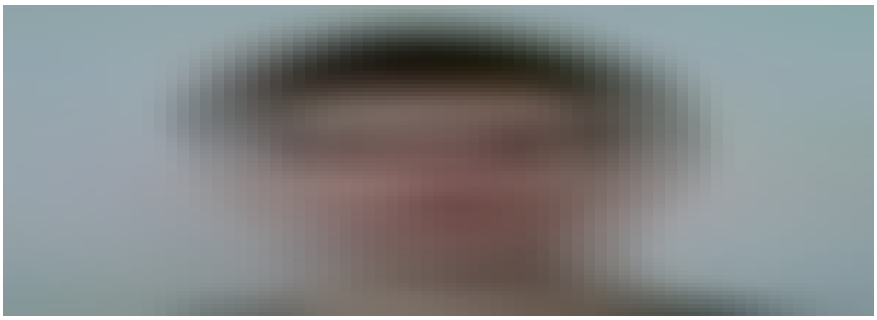
Now create a `_no_posts.html.erb` partial for the message inside the `branch` directory.



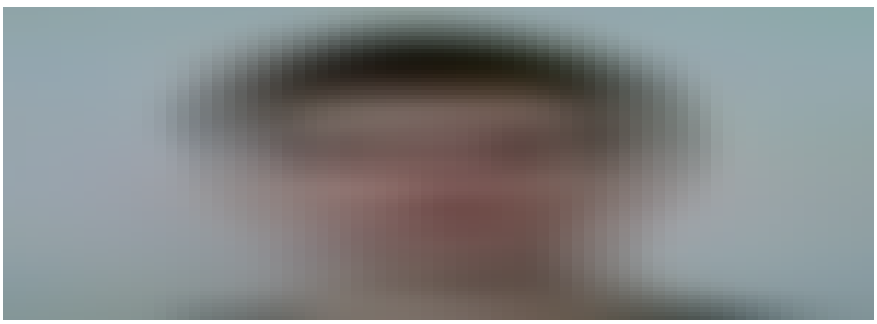
posts/branch/_no_posts.html.erb

Finally, we use the `will_paginate` method from the gem to split posts into multiple pages if there are a lot of posts.

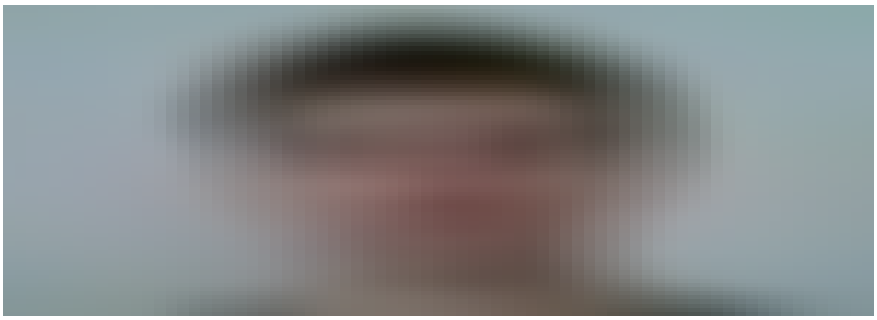
Create templates for `hobby`, `study` and `team` actions. Inside them we'll render the `_branch.html.erb` partial file and pass specific local variables.



posts/hobby.html.erb

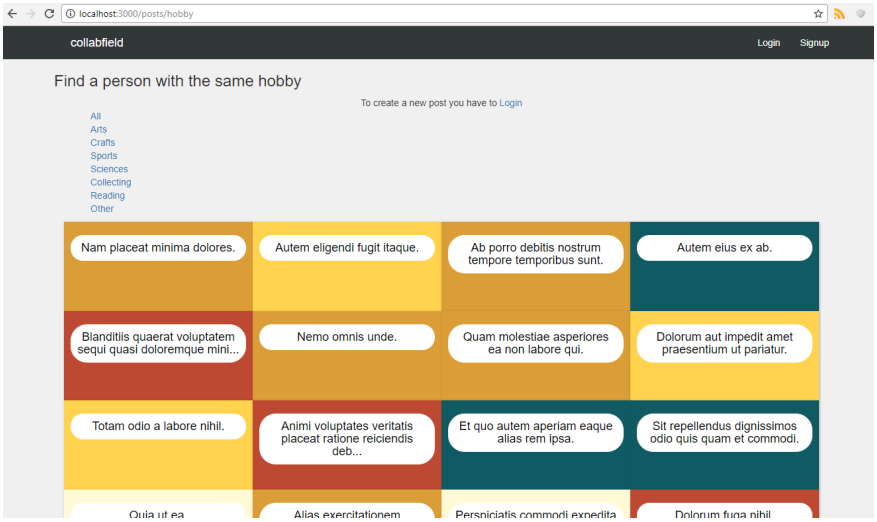


posts/study.html.erb

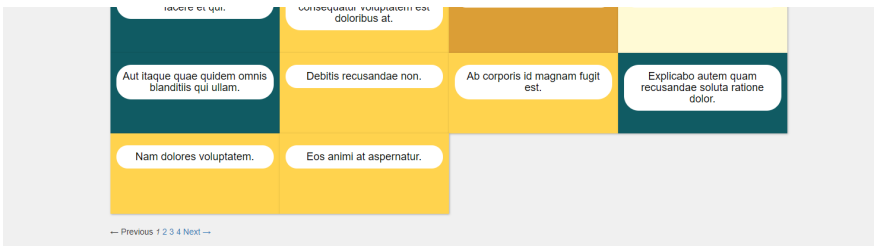


posts/team.html.erb

If you go to any of those branch pages, you will see something like this



Also if you scroll down, you will see that now we have a pagination



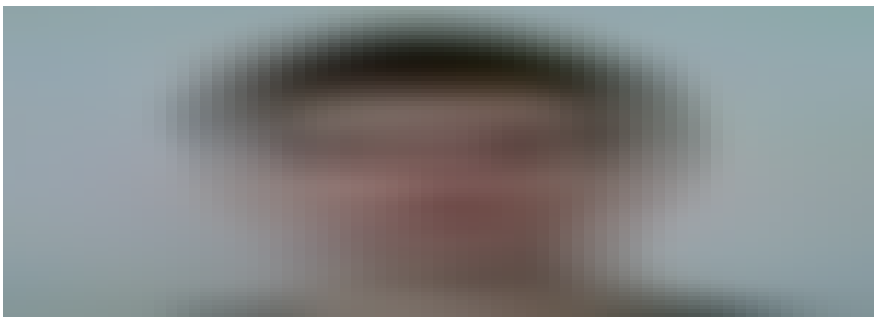
We've done quite a lot of work to create these branch pages. Commit the changes

```
git add -A
git commit -m "Create branch pages for specific posts

- Inside the PostsController define hobby, study and team
actions.
  Define a posts_for_branch method and call it inside these
actions
```

```
- Add will_paginate gem
- Create a _branch.html.erb partial file
- Create a _create_new_post.html.erb partial file
- Define a create_new_post_partial_path helper method
- Create a _signed_in.html.erb partial file
- Create a _not_signed_in.html.erb partial file
- Create a _categories.html.erb partial file
- Define a all_categories_button_partial_path helper method
- Create a _all_selected.html.erb partial file
- Create a _all_not_selected.html.erb partial file
- Define a no_posts_partial_path helper method
- Create a _no_posts.html.erb partial file
- Create a hobby.html.erb template file
- Create a study.html.erb template file
- Create a team.html.erb template file"
```

Specs

Cover helper methods with specs. The `posts_helper_spec.rb` file should look like this:



spec/helpers/posts_helper_spec.rb

Again, specs are pretty simple here. I used the `stub` method to define methods' return values. To define params, I selected the controller and simply defined it like this `controller.params[:param_name]`. And finally, I assigned instance variables by using an assign method.

Commit the changes

```
git add -A
git commit -m "Add specs for PostsHelper methods"
```

Design changes

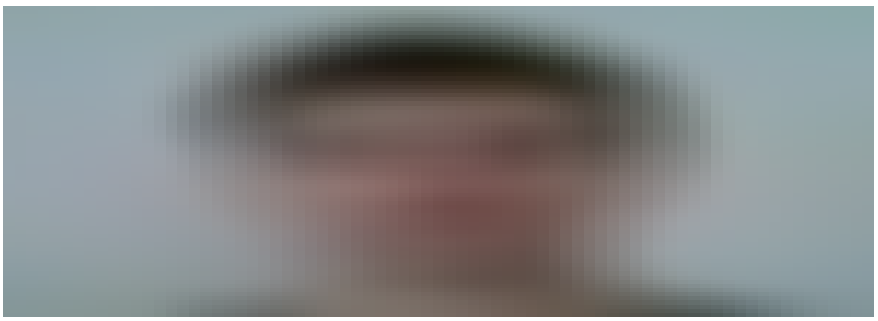
In these branch pages we want to have different posts' design. In the home page we have the cards design. In branch pages let's create a list

design, so a user could see more posts and browse through them more efficiently.

Inside the `posts` directory, create a `post` directory with a `_home_page.html.erb` partial inside.

```
posts/post/_home_page.html.erb
```

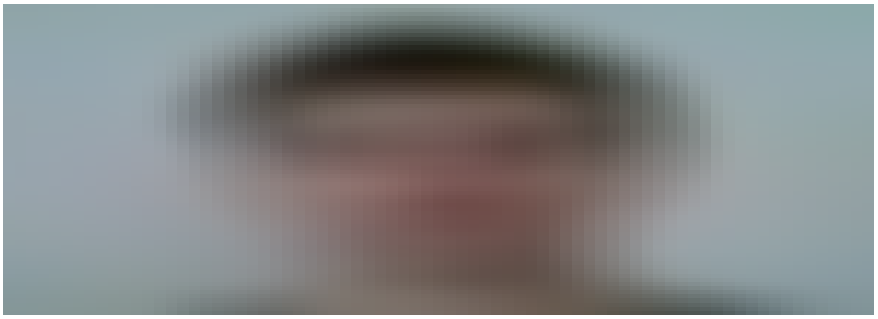
Cut the `_post.html.erb` partial's content and paste it inside the `_home_page.html.erb` partial file. Inside the `_post.html.erb` partial file add the following line of code:



posts/_post.html.erb

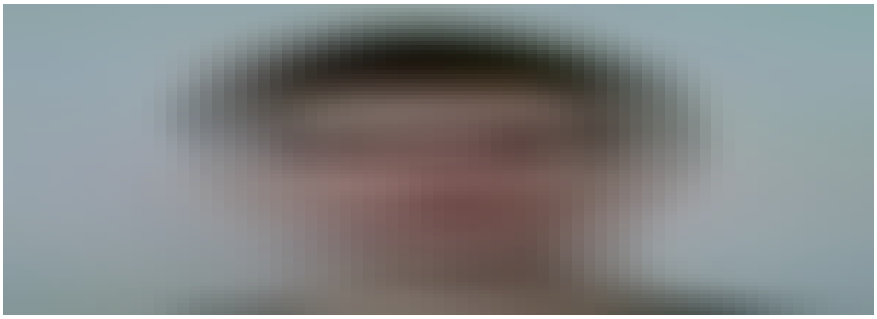
Here we call the `post_format_partial_path` helper method to decide which post design to render, depending on the current path. If a user is on the home page, render the post design for the home page. If a user is on the branch page, render the post design for the branch page. That's why we cut `_post.html.erb` file's content into `_home_page.html.erb` file.

Inside the `post` directory, create a new `_branch_page.html.erb` file and paste this code to define the posts design for the branch page.



posts/post/_branch_page.html.erb

To decide which partial file to render, define the `post_format_partial_path` helper method inside the `posts_helper.rb`



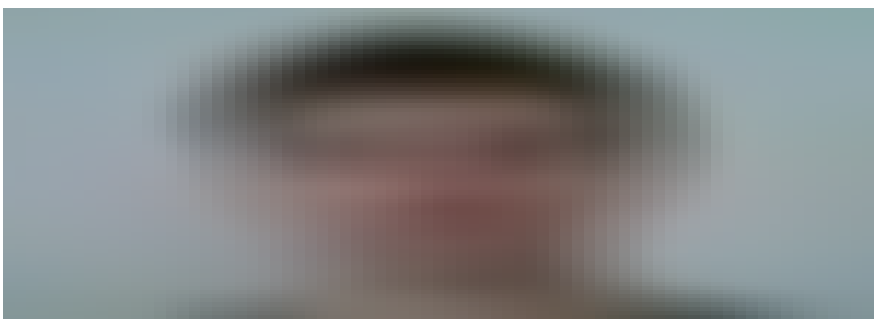
helpers/posts_helper.rb

The `post_format_partial_path` helper method won't be available in the home page, because we render posts inside the home page's template, which belongs to a different controller. To have an access to this method, inside the home page's template, include `PostsHelper` inside the `ApplicationHelper`

```
include PostsHelper
```

Specs

Add specs for the `post_format_partial_path` helper method:



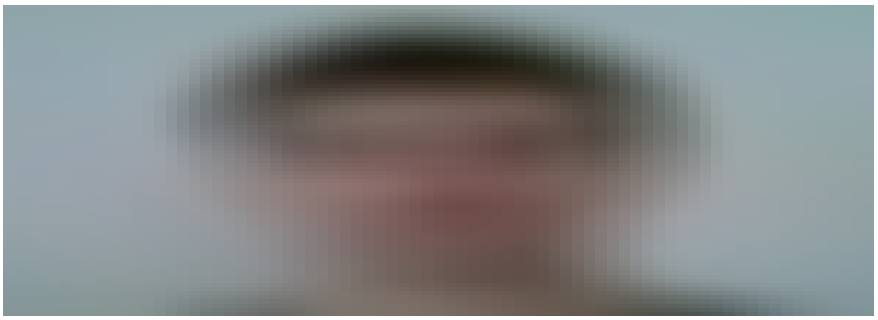
helpers/posts_helper_spec.rb

Commit the changes

```
git add -A
git commit -m "Add specs for the post_format_partial_path
helper method"
```

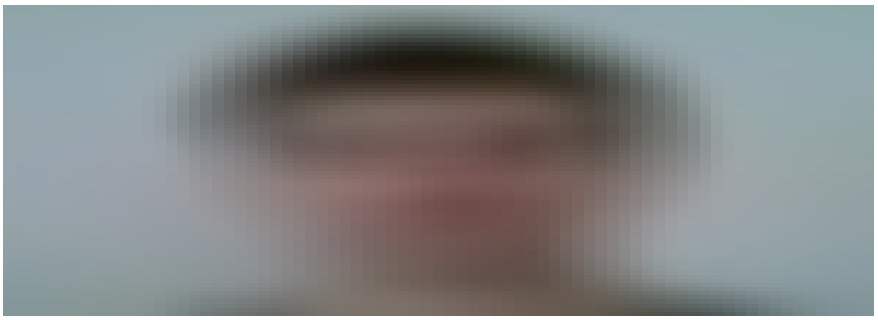
CSS

Describe the posts style in branch pages with CSS. Inside the `posts` directory, create a new `branch_page.scss` style sheet file:



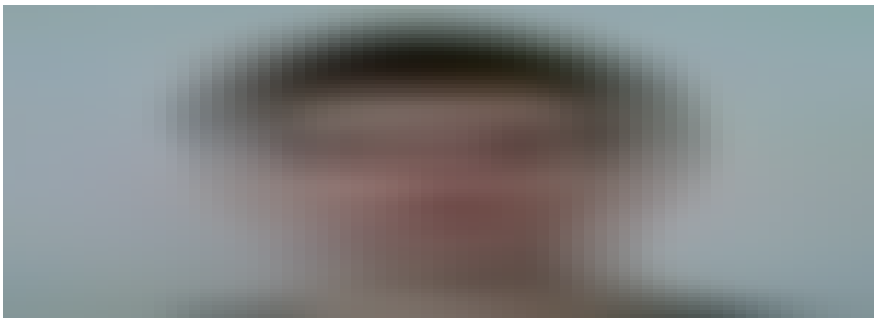
stylesheets/partials/posts/branch_page.scss

Inside the `base/default.scss` add:



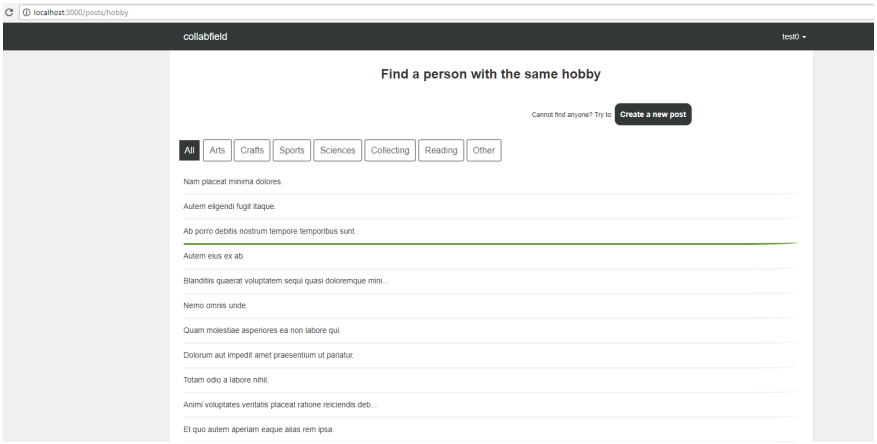
assets/stylesheets/base/default.scss

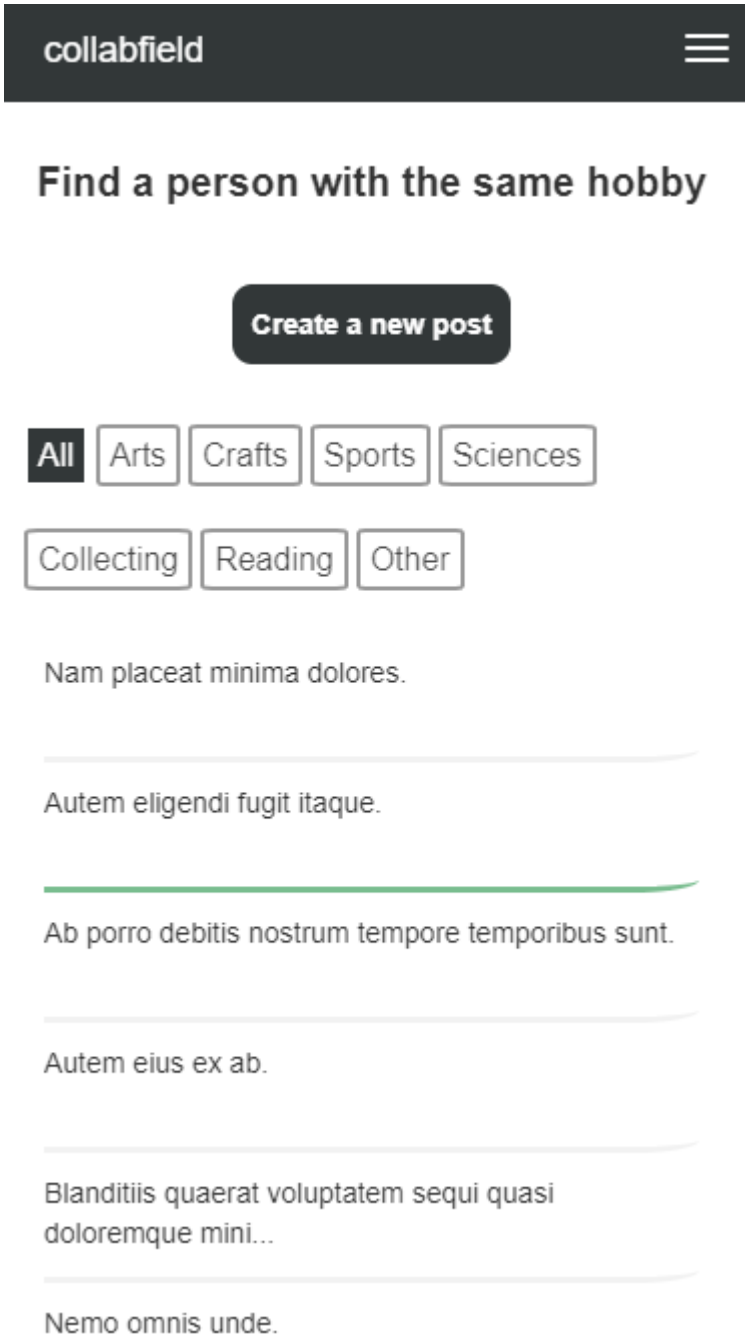
To fix style issues on smaller devices, inside the `responsive/mobile.scss` add:



assets/stylesheets/responsive/mobile.scss

Now the branch pages should look like this:


localhost:3000/posts/hobby
collabfield
test0
Find a person with the same hobby
Cannot find anyone? Try to
Create a new post
All
Arts
Crafts
Sports
Sciences
Collecting
Reading
Other
Nam placeat minima dolores.
Autem eligendi fugit itaque.
Ab porro debitis nostrum tempore temporibus sunt.
Autem eius ex ab.
Blanditiis quaerat voluptatem sequi quasi doloremque mini...
Nemo omnis unde.
Quam molestiae asperiores ea non labore qui.
Dolorum aut impedit amet praesentium ut pariatur.
Totam odio a labore nihil.
Animi voluptates veritatis placeat ratione reiciendis deb...
Et quo autem aperiam eaque alias rem ipsa.

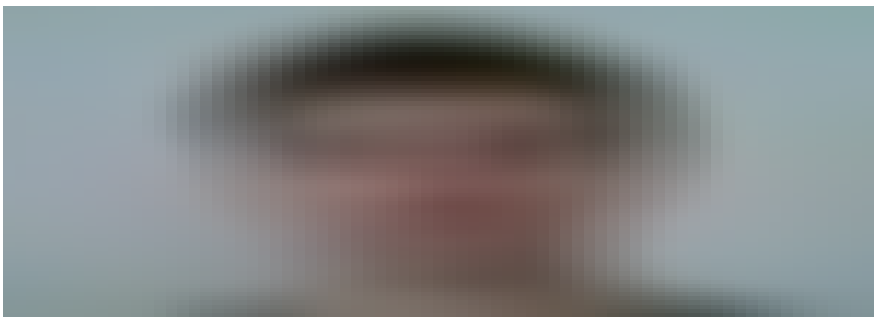

collabfield
Find a person with the same hobby
Create a new post
All
Arts
Crafts
Sports
Sciences
Collecting
Reading
Other
Nam placeat minima dolores.
Autem eligendi fugit itaque.
Ab porro debitis nostrum tempore temporibus sunt.
Autem eius ex ab.
Blanditiis quaerat voluptatem sequi quasi doloremque mini...
Nemo omnis unde.

Commit the changes.

```
git add -A
git commit -m "Describe the posts style in branch pages

- Create a branch_page.scss file and add CSS
- Add CSS to the default.scss file
- Add CSS to the mobile.scss file"
```

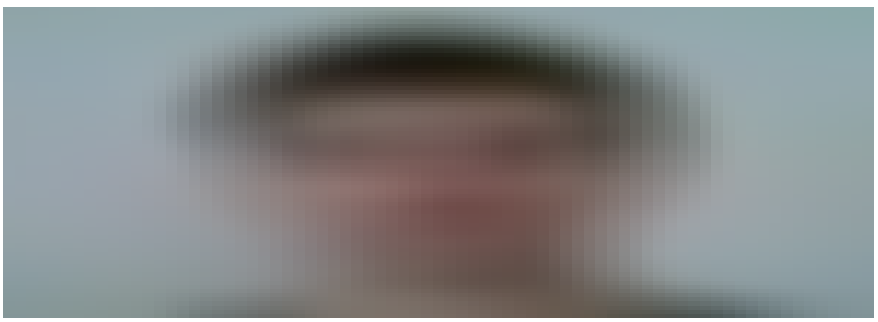
Search bar

We want not only be able to browse through posts, but also search for specific ones. Inside the `_branch.html.erb` partial file, above the `categories` row, add:



posts/_branch.html.erb

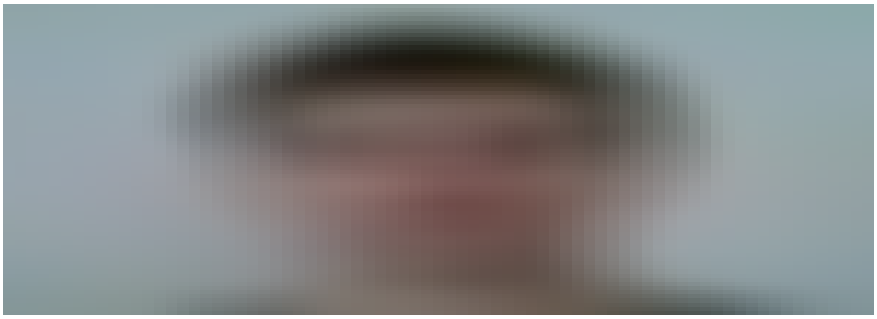
Create a `_search_form.html.erb` partial file inside the `branch` directory and add the following code inside:



posts/branch/_search_form.html.erb

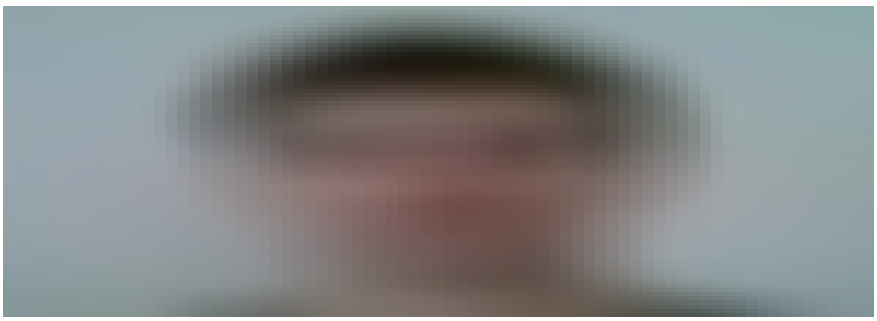
Here with the `send` method we dynamically generate a path to a specific `PostsController`'s action, depending on a current branch. Also we send an extra data field for the category if a specific category is selected. If a user has selected a specific category, only search results from that category will be returned.

Define the `category_field_partial_path` helper method inside the `posts_helper.rb`



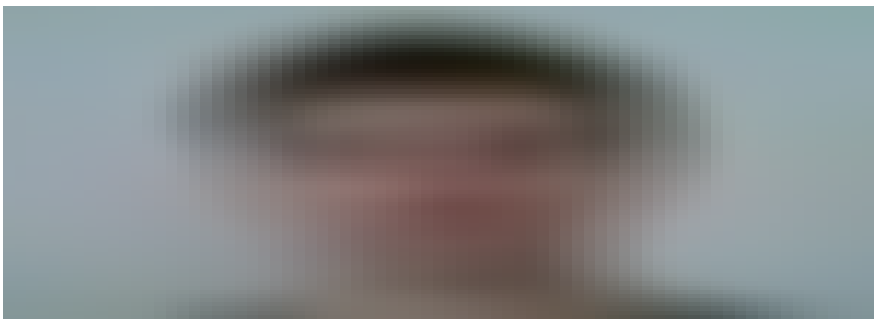
helpers/posts_helper.rb

Create a `_category_field.html.erb` partial file and add the code:



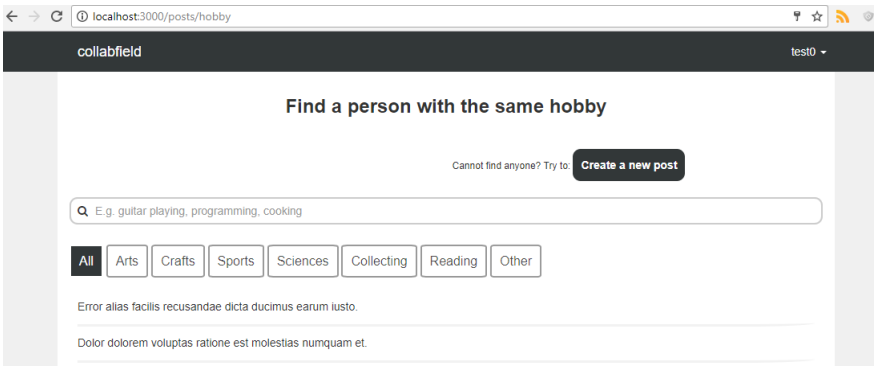
posts/branch/search_form/_category_field.html.erb

To give the search form some style, add CSS to the `branch_page.scss` file:



assets/stylesheets/partials/posts/branch_page.scss

The search form, in branch pages, should look likes this now

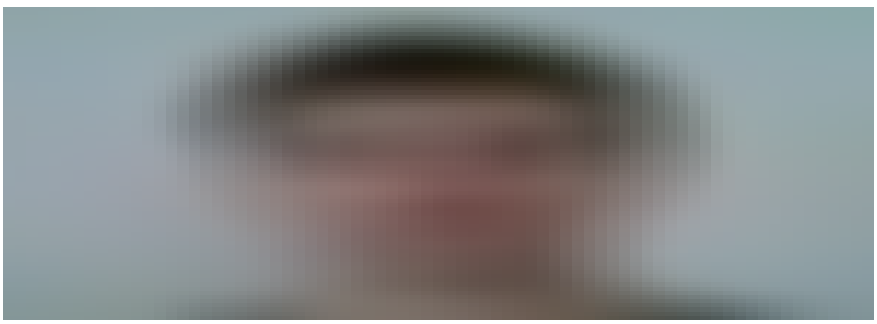


Commit the changes

```
git add -A
git commit -m "Add a search form in branch pages

- Render a search form inside the _branch.html.erb
- Create a _search_form.html.erb partial file
- Define a category_field_partial_path helper method in
PostsHelper
- Create a _category_field.html.erb partial file
- Add CSS for the the search form in branch_page.scss"
```

Currently our form isn't really functional. We could use some gems to achieve search functionality, but our data isn't complicated, so we can create our own simple search engine. We'll use scopes inside the `Post` model to make queries chainable and some conditional logic inside the controller (we will extract it into service object in the next section to make the code cleaner).

Start by defining scopes inside the `Post` model. To warm up, define the `default_scope` inside the `post.rb` file. This orders posts in descending order by the creation date, newest posts are at the top.

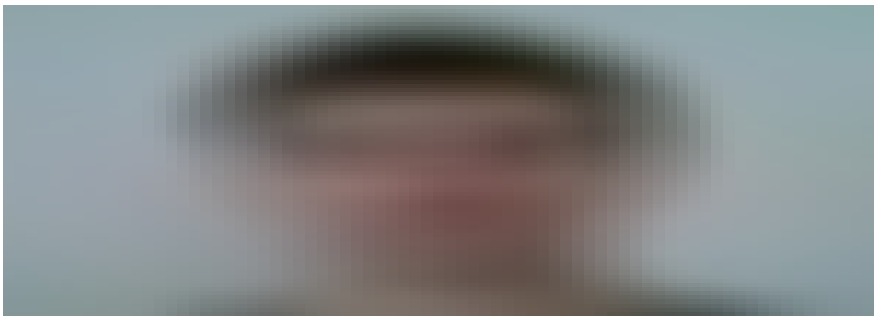


models/post.rb

Commit the change

```
git add -A
git commit -m "Define a default_scope for posts"
```

Make sure that the `default_scope` works correctly by wrapping it with a spec. Inside the `post_spec.rb` file, add:

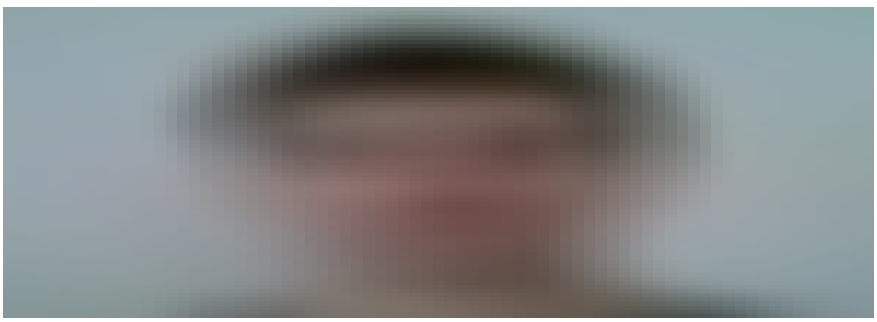


spec/models/post_spec.rb

Commit the change:

```
git add -A
git commit -m "Add a spec for the Post model's
default_scope"
```

Now let's make the search bar functional. Inside the `posts_controller.rb` replace the `get_posts` method's content with:

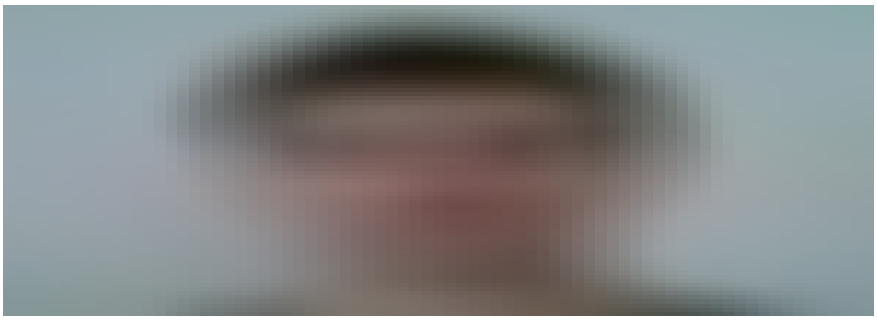


controllers/posts_controller.rb

As I've mentioned a little bit earlier, logic, just like in views, isn't really a good place in controllers. We want to make them clean. So we'll extract the logic out of this method in the upcoming section.

As you see, there is some conditional logic going on. Depending on a user request, data gets queried differently using scopes.

Inside the `Post` model, define those scopes:



models/post.rb

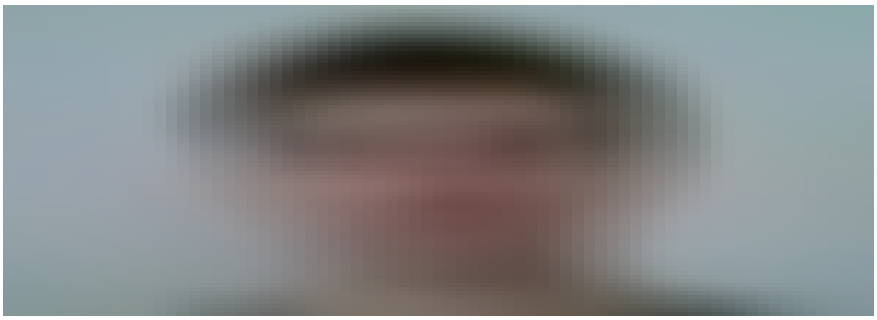
The `joins` method is used to query records from the associated tables. Also the basic SQL syntax is used to find records, based on provided strings.

Now if you restart the server and go back to any of those branch pages, the search bar should work! Also now you can filter posts by clicking on category buttons. And also when you select a particular category, only posts from that category are queried when you use the search form.

Commit the changes

```
git add -A
git commit -m "Make search bar and category filters
in branch pages functional

- Add by_category, by_branch and search scopes in the Post
model
- Modify the get_posts method in PostsController"
```

Cover these scopes with specs. Inside the `post_spec.rb` file's `Scopes` context add:



spec/models/post_spec.rb

Commit the changes

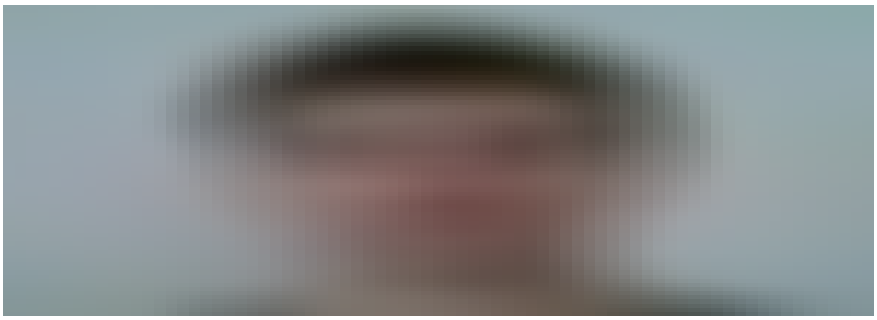
```
git add -A
git commit -m "Add specs for Post model's
by_branch, by_category and search scopes"
```

Infinite scroll

When you go to any of these branch pages, at the bottom of the page you see the pagination

When you click on the next link, it redirects you to another page with older posts. Instead of redirecting to another page with older posts, we can make an infinite scrolling functionality, similar to the Facebook's and Twitter's feed. You just scroll down and without any redirection and page reload, older posts are appended to the bottom of the list. Surprisingly, it is very easy to achieve. All we have to do is write some JavaScript. Whenever a user reaches the bottom of the page, AJAX request is sent to get data from the `next` page and that data gets appended to the bottom of the list.

Start by configuring the AJAX request and its conditions. When a user passes a certain threshold by scrolling down, AJAX request gets fired. Inside the `javascripts/posts` directory, create a new `infinite_scroll.js` file and add the code:

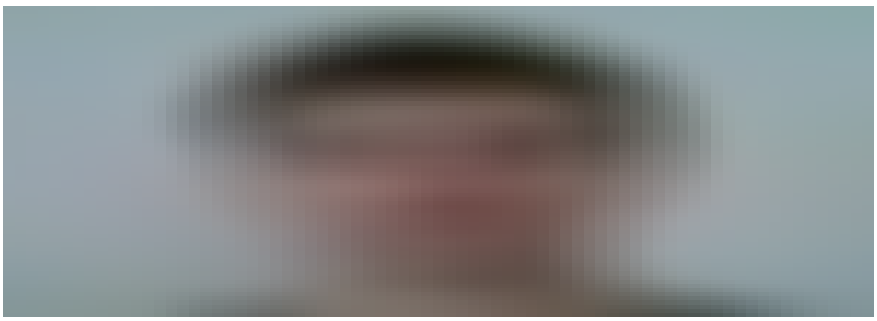


assets/javascripts/posts/infinite_scroll.js

The `isLoading` variable makes sure that only one request is sent at a time. If there is currently a request in progress, other requests won't be initiated.

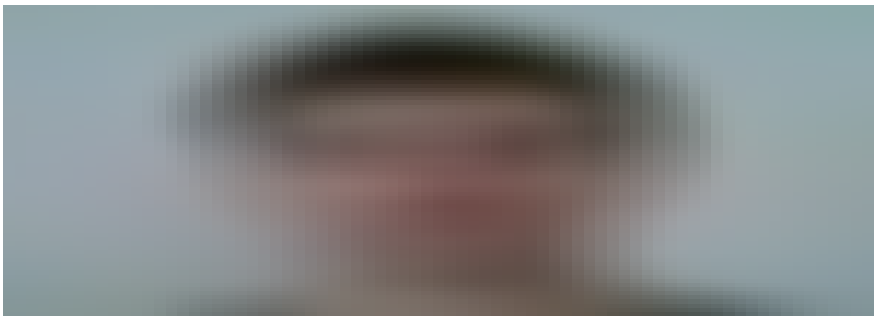
First check if pagination is present, if there are any more posts to render. Next, get a link to the next page, this is where the data will be retrieved from. Then set a threshold when to call an AJAX request, in this case the threshold is `60px` from the bottom of the window. Finally, if all conditions successfully pass, load data from the `next` page using the `getScript()` function.

Because the `getScript()` function loads the JavaScript file, we have to specify which file to render inside the `PostsController`. Inside the `posts_for_branch` method specify `respond_to` formats and which files to render.



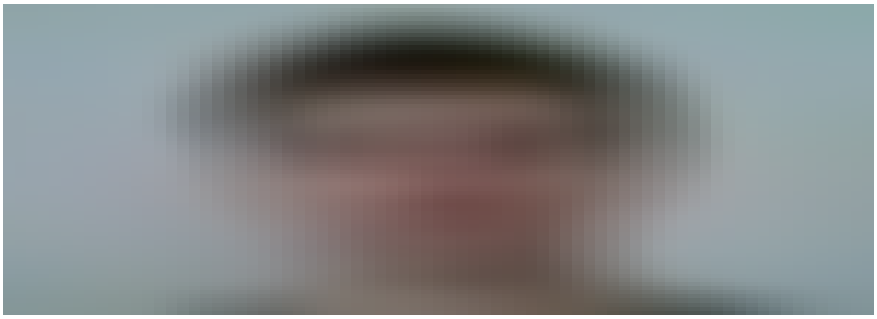
controllers/posts_controller.rb

When the controller tries to respond with the `.js` file, the `posts_pagination_page` template gets rendered. This partial file appends newly retrieved posts to the list. Create this file to append new posts and update the pagination element.



posts/_posts_pagination_page.js.erb

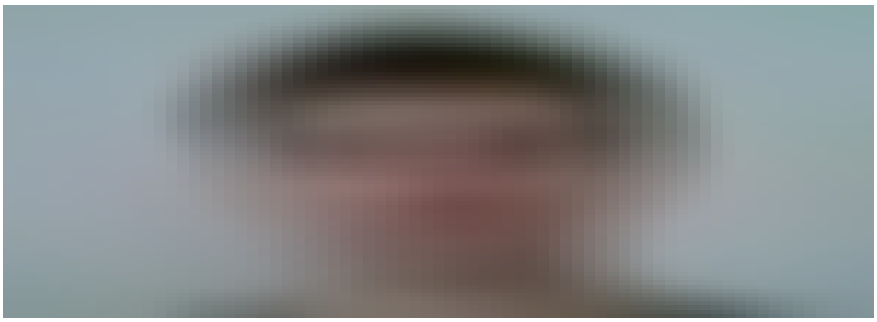
Create an `update_pagination_partial_path` helper method inside the `posts_helper.rb`



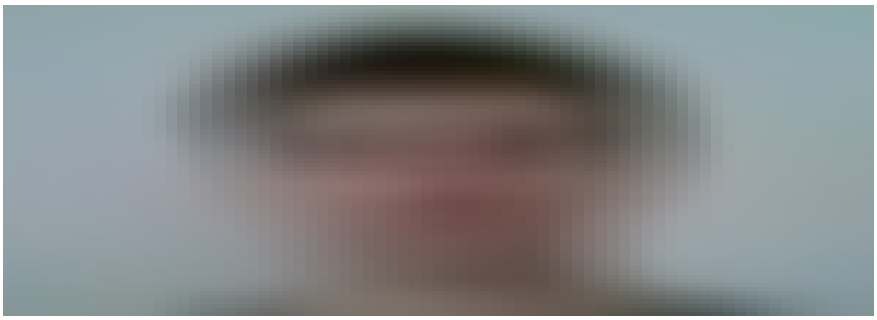
helpers/posts_helper.rb

Here the `next_page` method from the `will_paginate` gem is used, to determine if there are any more posts to load in the future or not.

Create the corresponding partial files:



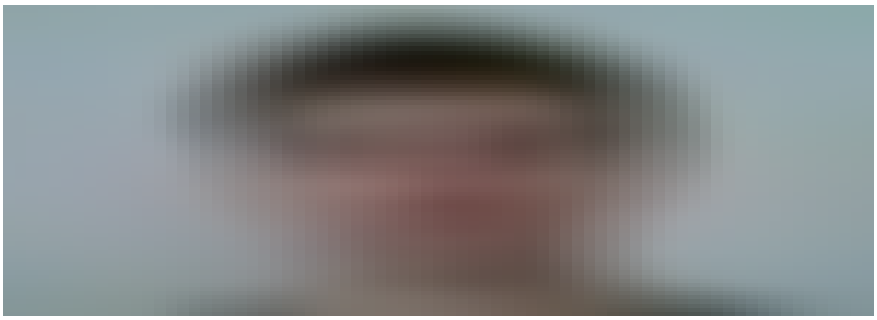
posts/posts_pagination_page/_update_pagination.js.erb



posts/posts_pagination_page/_remove_pagination.js.erb

If you go to any of the branch pages and scroll down, older posts should be automatically appended to the list.

Also we no longer need to see the pagination menu, so hide it with CSS. Inside the `branch_page.scss` file add:



stylesheets/partials/posts/branch_page.scss

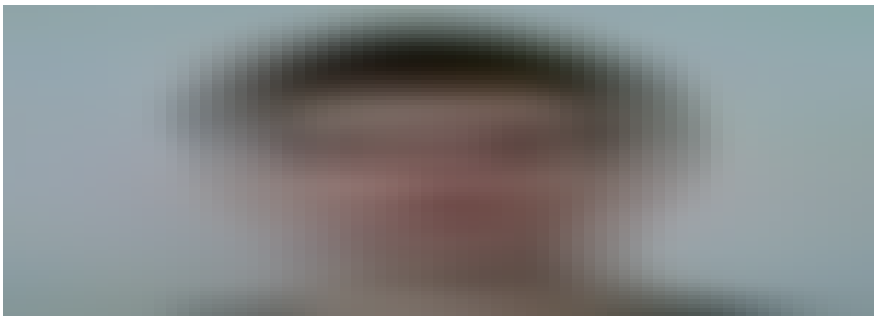
Commit the changes

```
git add -A
git commit -m "Transform posts pagination into infinite
scroll
```

```
- Create an infinite_scroll.js file
- Inside PostController's posts_for_branch method add
respond_to format
- Define an update_pagination_partial_path
- Create _update_pagination.js.erb and
_remove_pagination.js.erb partials
- hide the .infinite-scroll element with CSS"
```

Specs

Cover the `update_pagination_partial_path` helper method with specs:



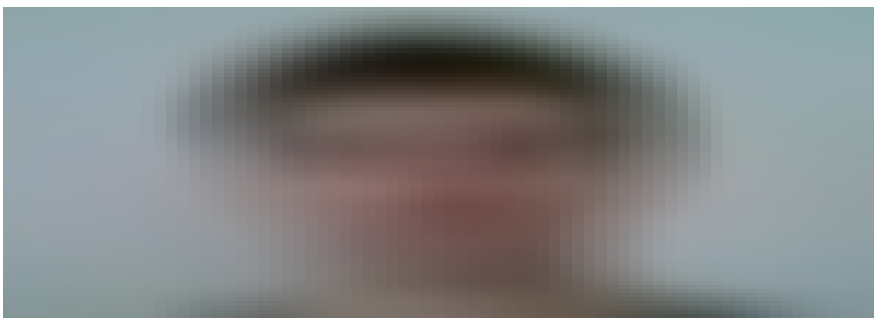
spec/helpers/post_helper_spec.rb

Here I've used a test `double` to simulate the `posts` instance variable and its chained method `next_page`. You can learn more about the RSpec Mocks here.

Commit the changes:

```
git add -A
git commit -m "Add specs for the
update_pagination_partial_path
helper method"
```

We can also write feature specs to make sure that posts are successfully appended, after you scroll down. Create an `infinite_scroll_spec.rb` file:



spec/features/posts/infinite_scroll_spec.rb

In the spec file all branch pages are covered. We make sure that this functionality works on all three pages. The `per_page` is `will_paginate` gem's method. Here the `Post` model is selected and the default number of posts per page is set.

The `check_posts_count` method is defined to reduce the amount of code the file has. Instead of repeating the same code over and over again in different specs, we extracted it into a single method. Once the

page is visited, it is expected to see 15 posts. Then the `execute_script` method is used to run JavaScript, which scrolls the scrollbar to the browser's bottom. Finally, after the scroll, it is expected to see an additional 15 posts. Now in total there should be 30 posts on the page.

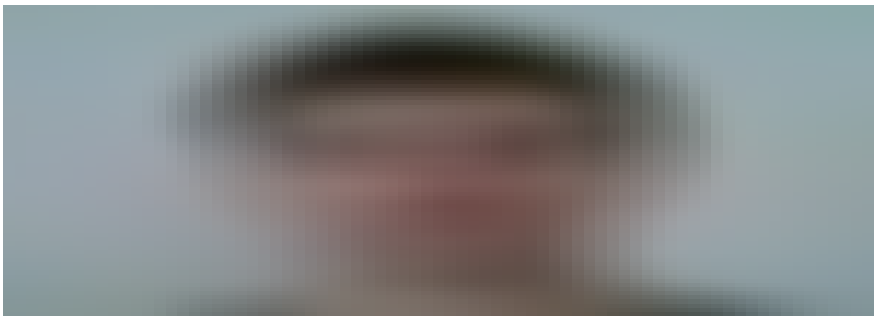
Commit the changes:

```
git add -A
git commit -m "Add feature specs for posts' infinite scroll
functionality"
```

Home page update

Currently on the home page we can only see few random posts. Modify the home page, so we could see a few posts from all branches.

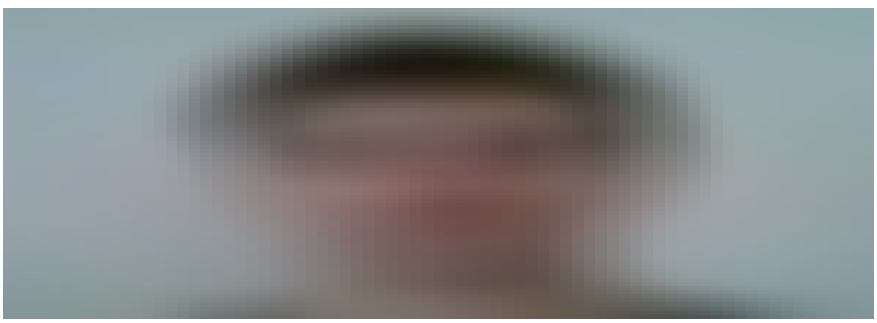
Replace the `_main_content.html.erb` file's content with:



pages/index/_main_content.html.erb

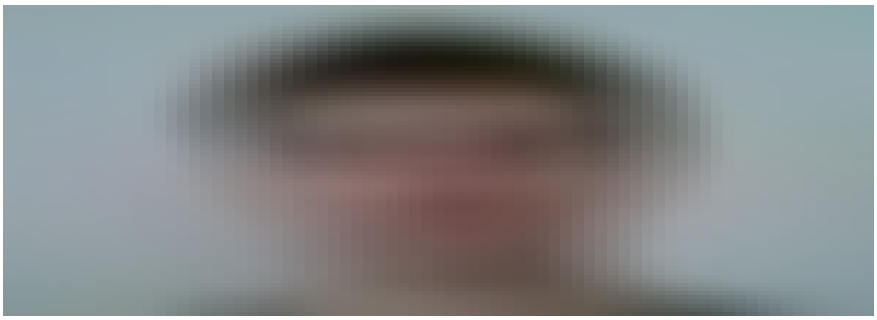
We created sections with posts for every branch.

Define instance variables inside the `PagesController`'s `index` action. The action should look like this:



controllers/pages_controller.rb

We have the `no_posts_partial_path` helper method from before, but we should modify it a little bit and make it more reusable. Currently it works only for branch pages. Add a `posts` parameter to the method, so it should look like this now:



helpers/posts_helper.rb

Here the `posts` parameter was added, instance variable was changed to a simple variable and the partial's path was changed too. So move the `_no_posts.html.erb` partial file from

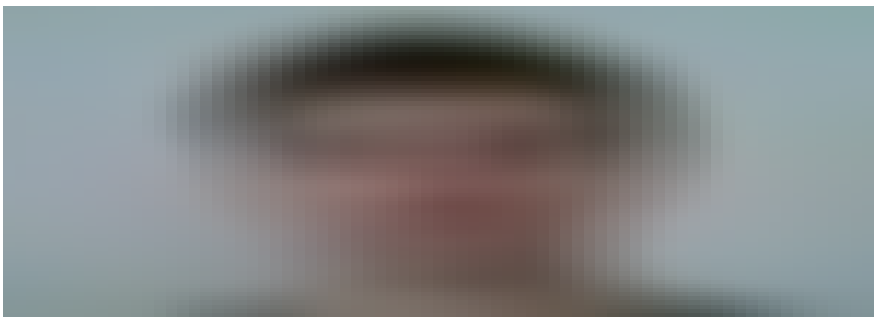
```
posts/branch/_no_posts.html.erb
```

to

```
posts/shared/_no_posts.html.erb
```

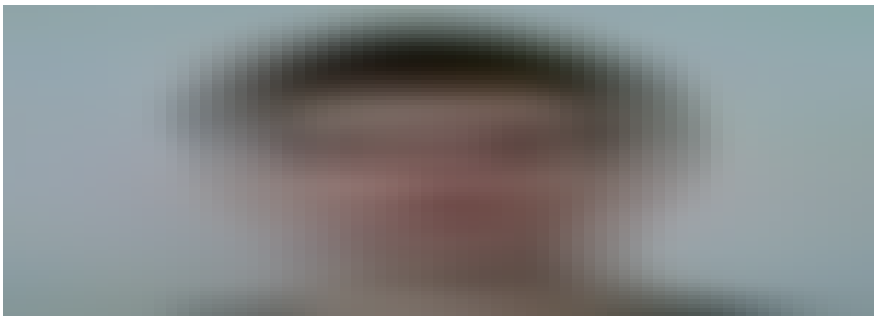
Also inside the `_branch.html.erb` file pass the `@posts` instance variable to the `no_posts_partial_path` method as an argument.

Add some style changes. Inside the `default.scss` file add:



assets/stylesheets/base/default.scss

And inside the `home_page.scss` add:



assets/stylesheets/partials/home_page.scss

The home page should look similar to this right now

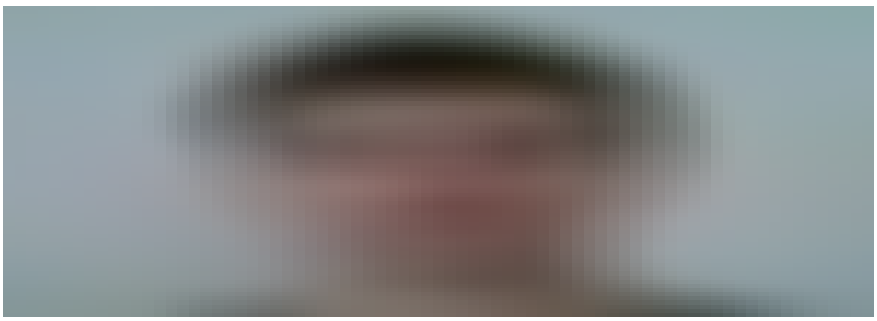
Commit the changes

```
git add -A
git commit -m "Add posts from all branches in the home page

- Modify the _main_content.html.erb file
- Define instance variables inside the PagesController's
index action
- Modify the no_posts_partial_path helper method to be more
reusable
- Add CSS to style the home page"
```

Service objects

As I've mentioned before, if you put logic inside controllers, they become complicated very easily and a huge pain to test. That's why it's a good idea to extract logic from them somewhere else. To do that I use design patterns, service objects (services) to be more specific.

Right now inside the `PostsController`, we have this method:



controllers/posts_controller.rb

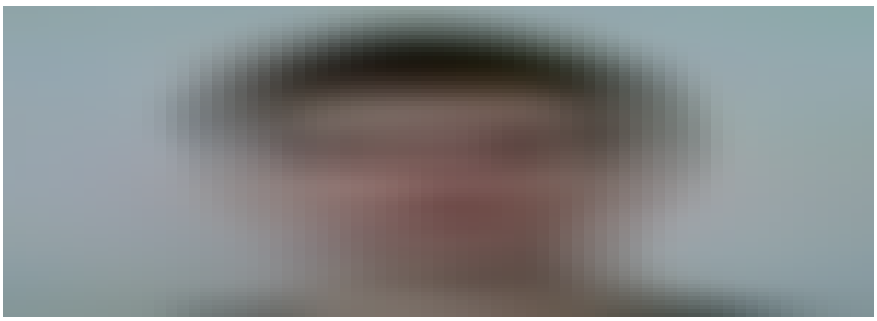
It has a lot of conditional logic which I want to remove by using services. Service objects (services) design pattern is just a basic ruby class. It's very simple, we just pass data which we want to process and call a defined method to get a desired return value.

In ruby we pass data to Class's `initialize` method, in other languages it's known as the `constructor`. And then inside the class, we just create a method which will handle all defined logic. Let's create that and see how it looks in code.

Inside the `app` directory, create a new `services` directory:

```
app/services
```

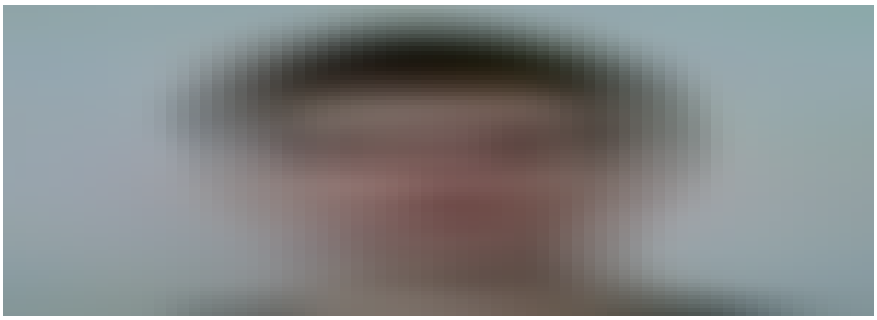
Inside the directory, create a new `posts_for_branch_service.rb` file:



services/posts_for_branch_service.rb

Here, as described above, it is just a plain ruby class with an `initialize` method to accept parameters and a `call` method to handle the logic. We took this logic from the `get_posts` method.

Now simply create a new object of this class and call the `call` method inside the `get_posts` method. The method should look like this right now:



controllers/posts_controller.rb

Commit the changes:

```
git add -A
git commit -m "Create a service object to extract logic
from the get_posts method"
```

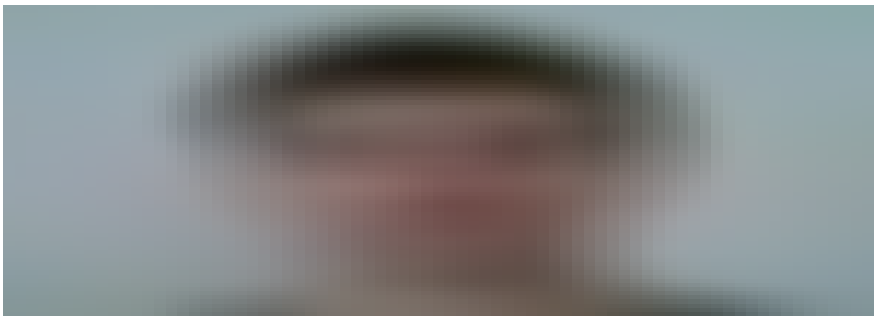
Specs

A fortunate thing about design patterns, like services, is that it's easy to write unit tests for it. We can simply write specs for the `call` method and test each of its conditions.

Inside the `spec` directory create a new `services` directory:

```
spec/services
```

Inside the directory create a new file `posts_for_branch_service_spec.rb`



spec/services/posts_for_branch_service_spec.rb

At the top of the file, the `posts_for_branch_service.rb` file is loaded and then each of the `call` method's conditions are tested.

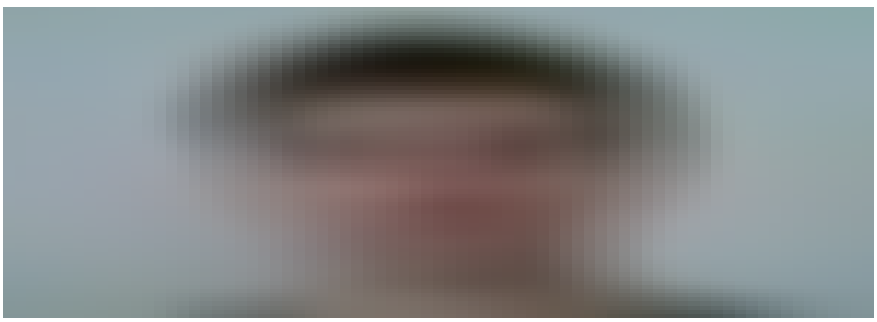
Commit the changes

```
git add -A
git commit -m "Add specs for the PostsForBranchService"
```

Create a new post

Until now posts were created artificially, by using seeds. Let's add a user interface for it, so a user could create posts.

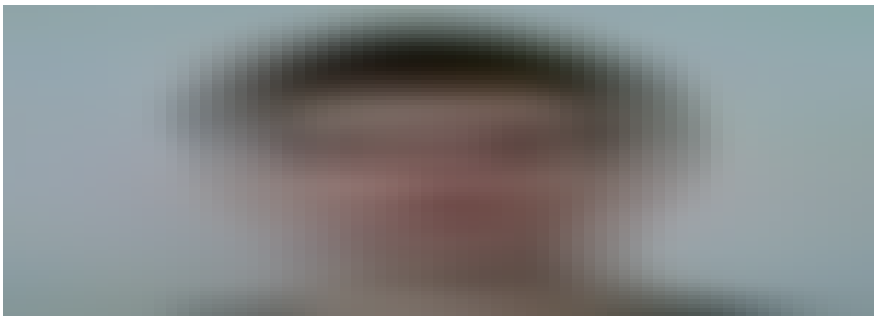
Inside the `posts_controller.rb` file add `new` and `create` actions.



controllers/posts_controller.rb

Inside the `new` action, we define some instance variables for the form to create new posts. Inside the `@categories` instance variable, categories for a specific branch are stored. The `@post` instance variable stores an object of a new post, this is needed for the Rails form.

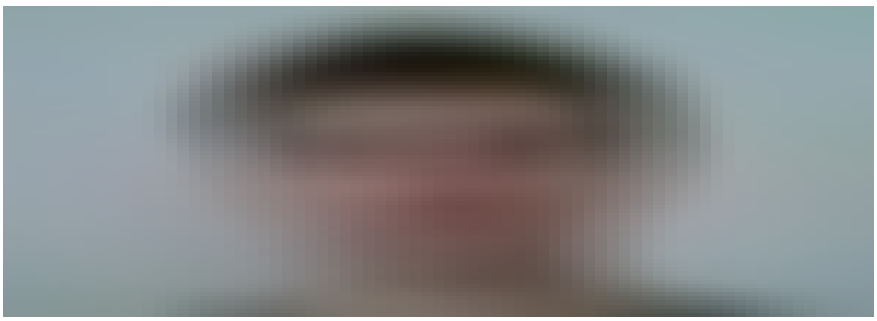
Inside the `create` action's `@post` instance variable, we create a new `Post` object and fill it with data, using the `post_params` method. Define this method within the `private` scope:



controllers/posts_controller.rb

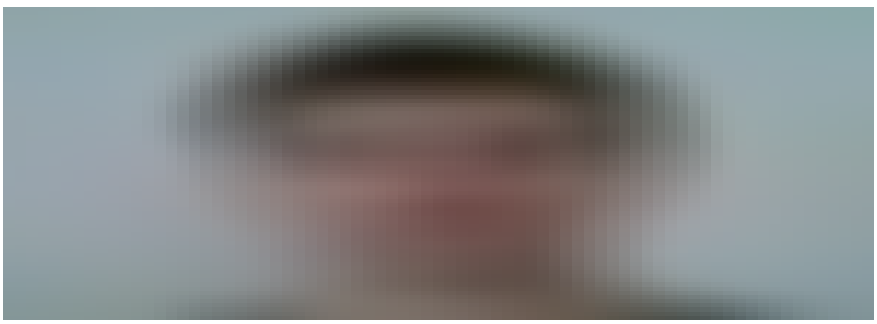
The `permit` method is used to whitelist attributes of the object, so only these specified attributes are allowed to be passed.

Also at the top of the `PostsController`, add the following line:



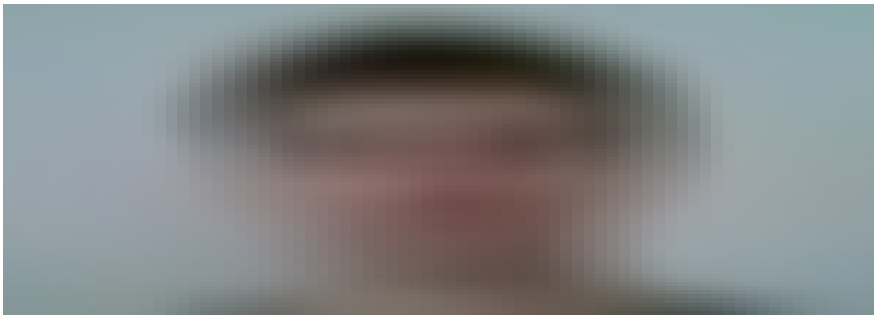
controllers/posts_controller.rb

The `before_action` is one of the Rails filters. We don't want to allow for not signed in users to have an access to a page where they can create new posts. So before calling the `new` action, the `redirect_if_not_signed_in` method is called. We'll need this method across other controllers too, so define it inside the `application_controller.rb` file. Also a method to redirect signed in users would be useful in the future too. So define them both.



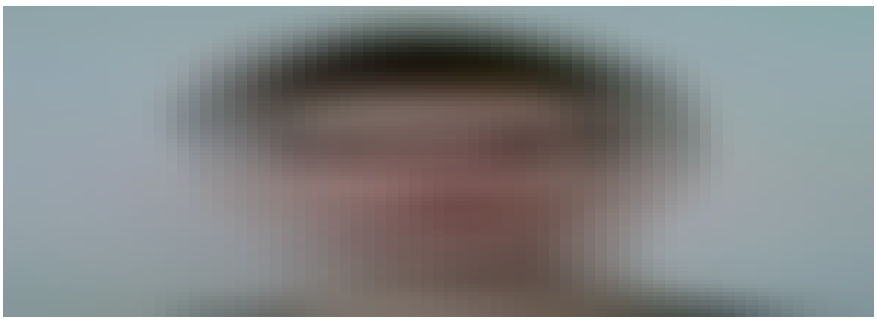
controllers/application_controller.rb

Now the `new` template is required, so a user could create new posts. Inside the `posts` directory, create a `new.html.erb` file:



posts/new.html.erb

Create a `new` directory and a `_post_form.html.erb` partial file inside:



posts/new/_post_form.html.erb

The form is pretty straightforward. Attributes of the fields are defined and the `collection_select` method is used to allow to select one of the available categories.

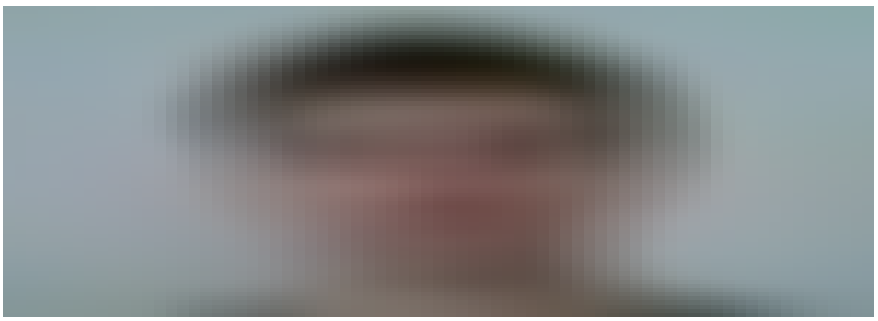
Commit the changes

```
git add -A
git commit -m "Create a UI to create new posts

- Inside the PostsController:
  define new and create actions
  define a post_params method
  define a before_action filter
- Inside the ApplicationController:
  define a redirect_if_not_signed_in method
  define a redirect_if_signed_in method
- Create a new template for posts"
```

We can test if the form works by writing specs. Start by writing request specs, to make sure that we get correct responses after we send particular requests. Inside the `spec` directory create a couple directories.

```
spec/requests/posts
```

And a `new_spec.rb` file inside:



spec/requests/posts/new_spec.rb

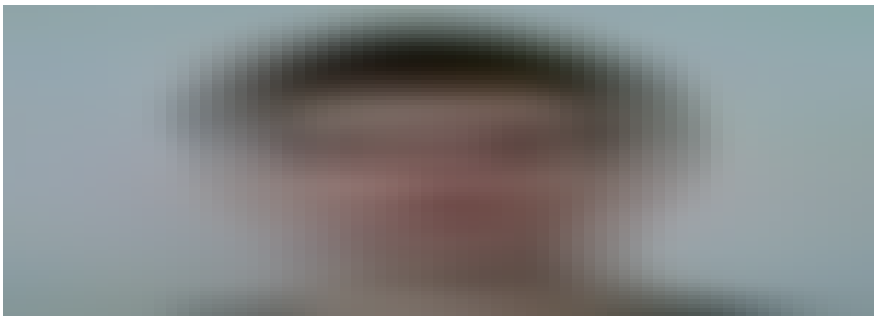
As mentioned in the documentation, request specs provide a thin wrapper around the integration tests. So we test if we get correct responses when we send certain requests. The `include Warden::Test::Helpers` line is required in order to use `login_as` method. The method logs a user in.

Commit the change.

```
git add -A
git commit -m "Add request specs for a new post template"
```

We can even add some more request specs for the pages which we created previously.

Inside the same directory create a `branches_spec.rb` file:



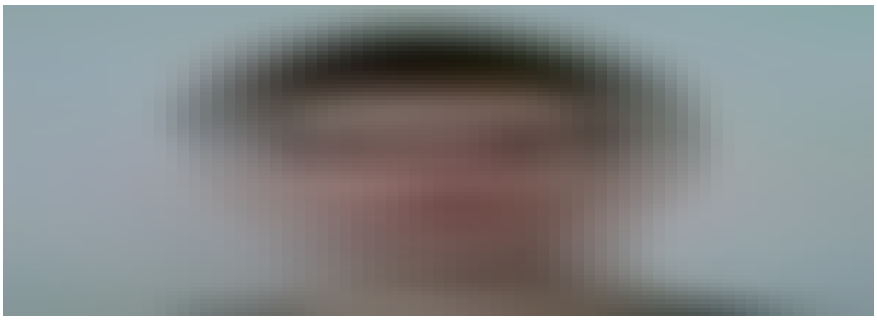
spec/requests/posts/branches_spec.rb

This way we check that all branch pages' templates successfully render. Also the `shared examples` is used to reduce the repetitive code.

Commit the change.

```
git add -A
git commit -m "Add request specs for Posts branch pages'
templates"
```

Also we can make sure that the `show` template renders successfully. Inside the same directory create a `show_spec.rb` file:

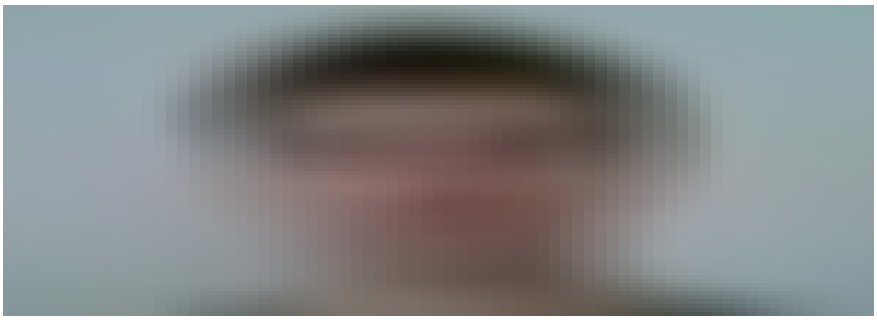


spec/requests/posts/show_spec.rb

Commit the changes.

```
git add -A
git commit -m "Add request specs for the Posts show
template"
```

To make sure that a user is able to create a new post, write feature specs to test the form. Inside the `features/posts` directory create a new file `create_new_post_spec.rb`



spec/features/posts/create_new_post_spec.rb

Commit the changes.

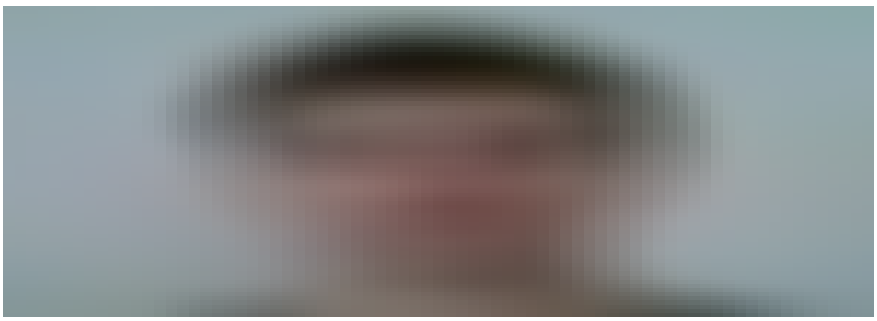
```
git add -A
git commit -m "Create a create_new_post_spec.rb file with
feature specs"
```

Apply some design to the `new` template.

Within the following directory:

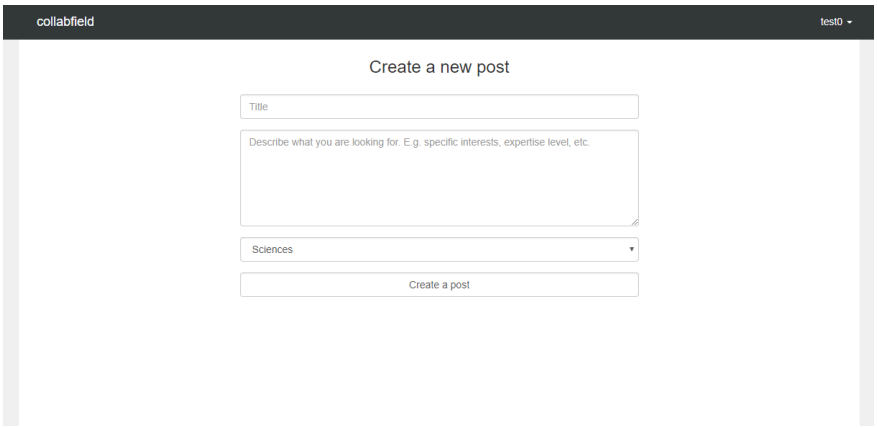
```
assets/stylesheets/partials/posts
```

Create a `new.scss` file:



assets/stylesheets/partials/posts/new.scss

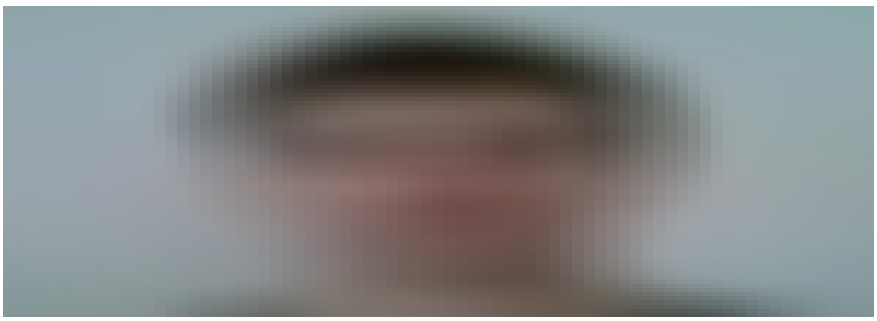
If you go to the template in a browser now, you should see a basic form



Commit the changes

```
git add -A
git commit -m "Add CSS to the Posts new.html.erb template"
```

Finally, we want to make sure that all fields are filled correctly. Inside the `Post` model we're going to define some validations. Add the following code to the `Post` model:



models/post.rb

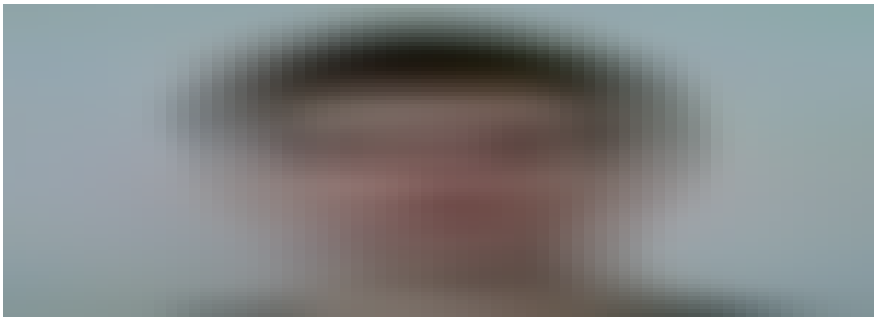
Commit the changes.

```
git add -A
git commit -m "Add validations to the Post model"
```

Cover these validations with specs. Go to the `Post` model's spec file:

```
spec/models/post_spec.rb
```

Then add:



spec/models/post_spec.rb

Commit the changes.

```
git add -A
git commit -m "Add specs for the Post model's validations"
```

Merge the `specific_branches` branch with the `master`

```
git checkout -b master
git merge specific_branches
git branch -D specific_branches
```

Instant Messaging

Users are able to publish posts and read other users' posts, but they have no ability to communicate with each other. We could create a simple mail box system, which would be much easier and faster to develop. But that is a very old way to communicate with someone. Real time communication is much more exciting to develop and comfortable to use.

Fortunately, Rails has Action Cables which makes real time features' implementation relatively easy. The core concept behind the Action Cables is that it uses a WebSockets Protocol instead of HTTP. And the core concept of WebSockets is that it establishes a client-server connection and keeps it open. This means that no page reloads are required to send and receive additional data.

Private conversation

The goal of this section is to create a working feature which would allow to have a private conversation between two users.

Switch to a new branch

```
git checkout -B private_conversation
```

Namespacing models

Start by defining necessary models. We'll need two different models for now, one for private conversations and another for private messages. We could name them `PrivateConversation` and `PrivateMessage`, but you can quickly encounter a little problem. While everything would work fine, imagine how the `models` directory would start to look like after we create more and more models with similar name prefixes. The directory would become hardly manageable in no time.

To avoid chaotic structure inside directories, we can use a namespacing technique.

Let's see how it would look like. An ordinary model for private conversation would be called `PrivateConversation` and its file would be called `private_conversation.rb`, and stored inside the `models` directory

```
models/private_conversation.rb
```

Meanwhile, the namespaced version would be called `Private::Conversation`. The file would be called `conversation.rb` and located inside the `private` directory

```
models/private/conversation.rb
```

Can you see how it might be useful? All files with the `private` prefix would be stored inside the `private` directory, instead of accumulating inside the main `models` directory and making it hardly to read.

As usually, Rails makes the development process enjoyable. We're able to create namespaced models by specifying a directory which we want to put a model in.

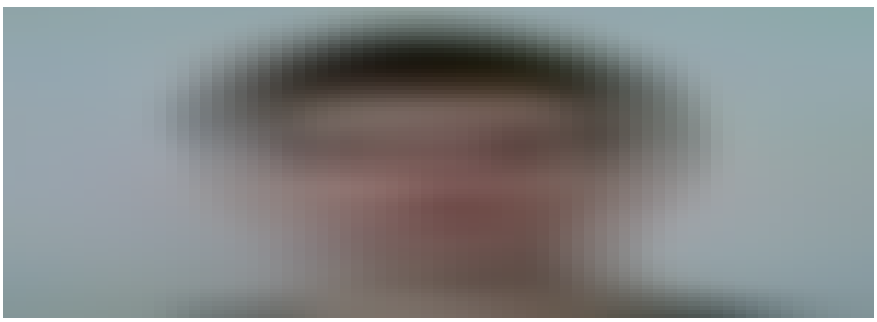
To create the namespaced `Private::Conversation` model run the following command:

```
rails g model private/conversation
```

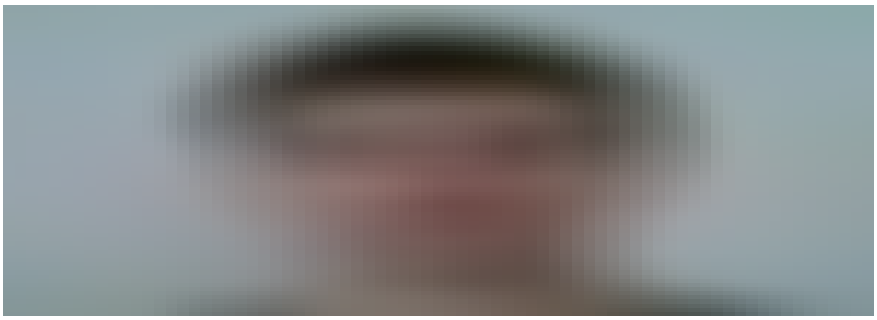
Also generate the `Private::Message` model:

```
rails g model private/message
```

If you look at the `models` directory, you will see a `private.rb` file. This is required to add prefix to database tables' names, so models could be recognized. Personally I don't like keeping those files inside the `models` directory, I prefer to specify a table's name inside a model itself. To specify a table's name inside a model, you have to use `self.table_name =` and provide a table's name as a string. If you choose to specify names of database tables this way, like I do, then the models should look like this:



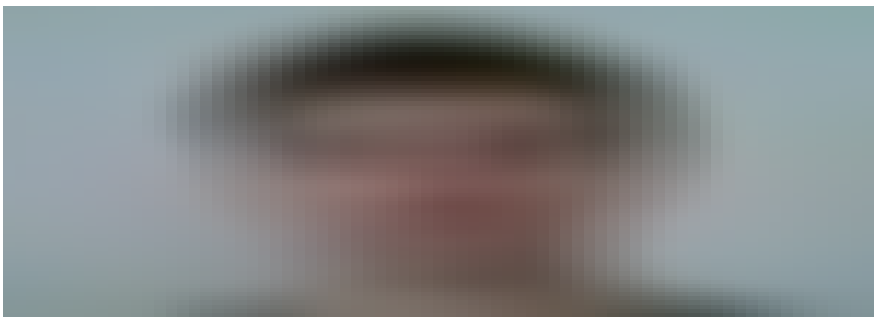
models/private/conversation.rb



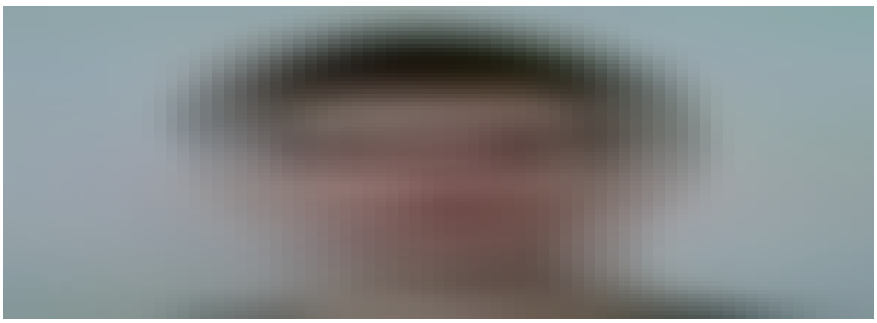
models/private/message.rb

The `private.rb` file, inside the `models` directory, is no longer needed, you can delete it.

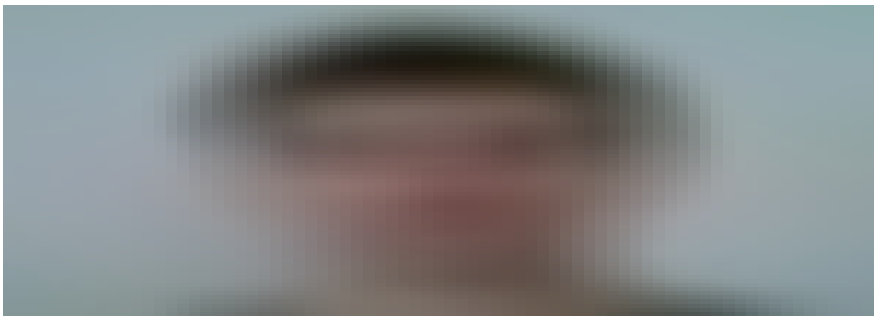
A user will be able to have many private conversations and conversations will have many messages. Define these associations inside the models:



models/private/conversation.rb



models/private/message.rb



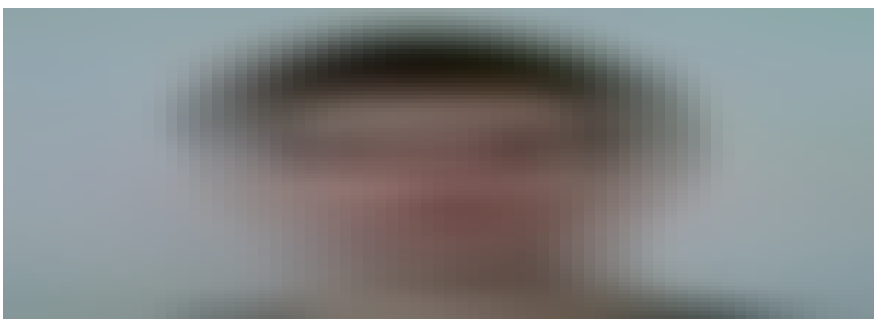
models/user.rb

Here the `class_name` method is used to define a name of an associated model. This allows to use custom names for our associations and make sure that namespaced models get recognized. Another use case of the `class_name` method would be to create a relation to itself, this is useful when you want to differentiate same model's data by creating some kind of hierarchies or similar structures.

The `foreign_key` is used to specify a name of association's column in a database table. A data column in a table is only created on the `belongs_to` association's side, but to make the column recognizable, we've to define the `foreign_key` with same values on both models.

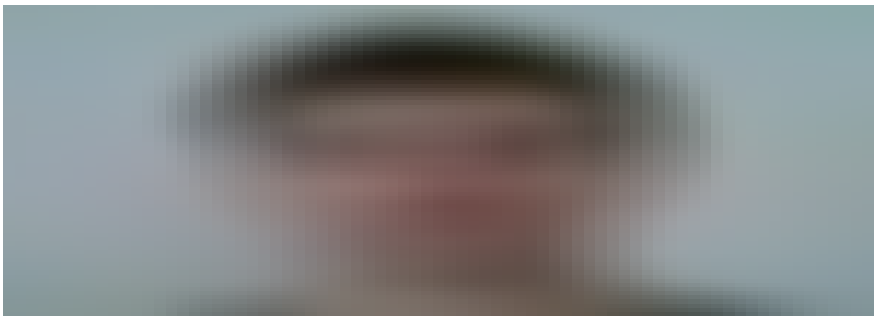
Private conversations are going to be between two users, here these two users are `sender` and `recipient`. We could've named them like `user1` and `user2`. But it's handy to know who initiated a conversation, so the `sender` here is a creator of a conversation.

Define data tables inside the migration files:



db/migrate/CREATION_DATE_create_private_conversations.rb

The `private_conversations` table is going to store users' ids, this is needed for `belongs_to` and `has_many` associations to work and of course to create a conversation between two users.



db/migrate/CREATION_DATE_create_private_messages.rb

Inside the `body` data column, a message's content is going to be stored. Instead of adding indexes and id columns to make associations between two models work, here we used the `references` method, which simplified the implementation.

run migration files to create tables inside the development database

```
rails db:migrate
```

Commit the changes

```
git add -A
git commit -m "Create Private::Conversation and
Private::Message models

- Define associations between User, Private::Conversation
  and Private::Message models
- Define private_conversations and private_messages tables"
```

A non-real time private conversation window

We have a place to store data for private conversations, but that's pretty much it. Where should we start from now? As mentioned in previous sections, personally I like to create a basic visual side of a feature and then write some logic to make it functional. I like this approach because when I have a visual element, which I want to make functional, it's more obvious what I want to achieve. Once you have a user interface, it's easier to start breaking down a problem into smaller steps, because you know what should happen after a certain event. It's harder to program something that doesn't exist yet.

To start building the user interface for private conversations, create a `Private::Conversations` controller. Once I namespace something in the app, I like to stay consistent and namespace all its other related parts too. This allows to understand and navigate through the source code more intuitively.

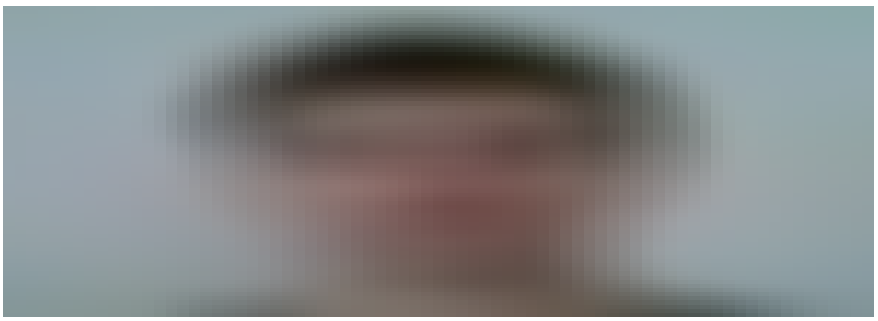
```
rails g controller private/conversations
```

Rails generator is pretty sweet. It created a namespaced model and namespaced views, everything is ready for development.

Create a new conversation

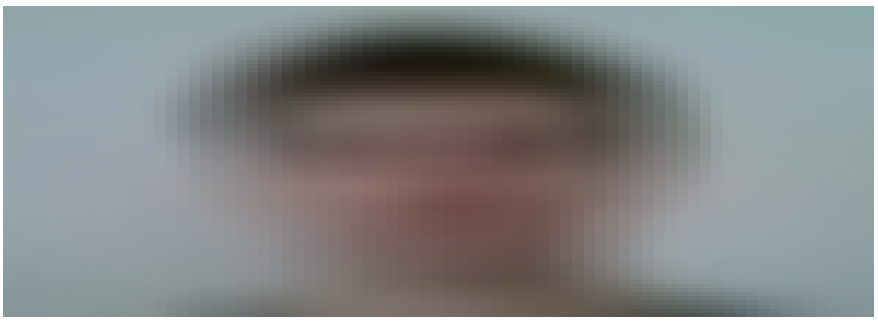
We need a way to initiate a new conversation. In a case of our app, it makes sense that you want to contact a person which has similar interests to yours. A convenient place for this functionality is inside a single post's page.

Inside the `posts/show.html.erb` template, create a form to initiate a new conversation. Below the `<p><%= @post.content %></p>` line add:



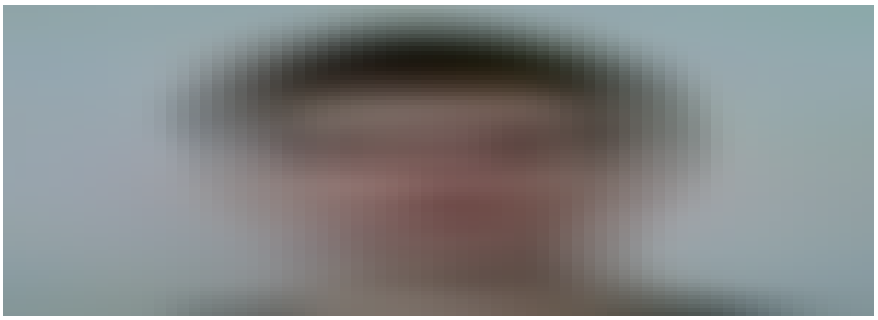
posts/show.html.erb

Define the helper method inside the `posts_helper.rb`



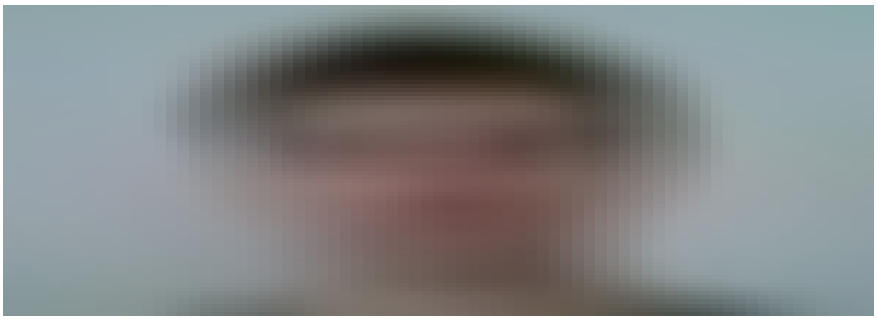
helpers/posts_helper.rb

Add specs for the helper method:

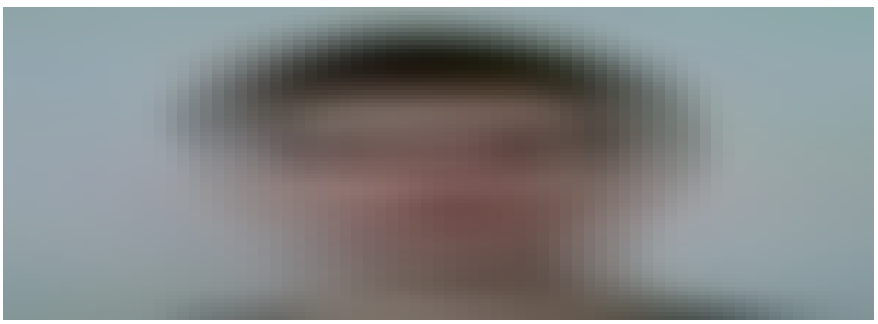


spec/helpers/posts_helper_spec.rb

Create a `show` directory and the corresponding partial files:

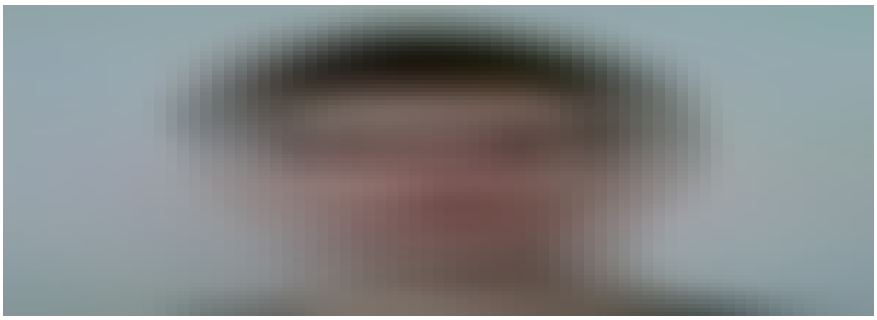


posts/show/_contact_user.html.erb



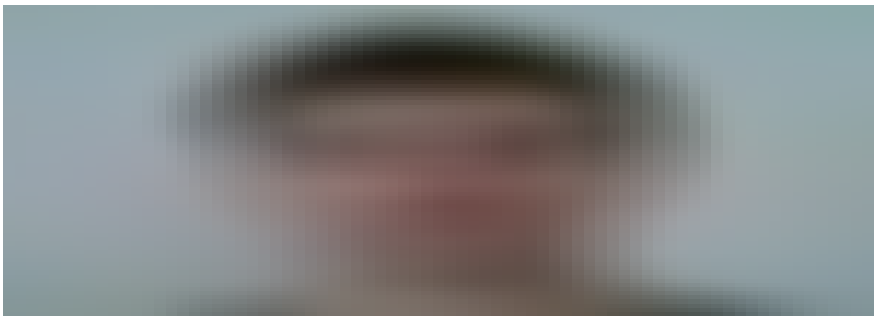
posts/show/_login_required.html.erb

Define the `leave_message_partial_path` helper method inside the `posts_helper.rb`



helpers/posts_helper.rb

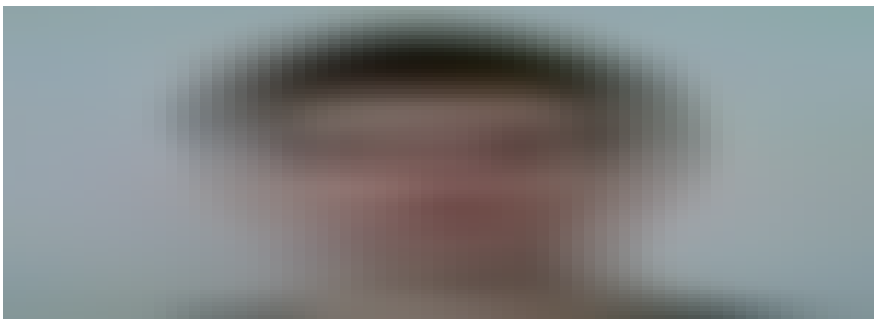
Add specs for the method



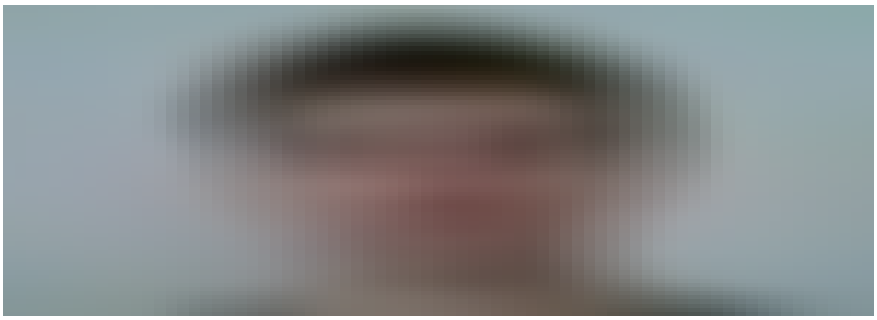
spec/helpers/posts_helper_spec.rb

We'll define the `@message_has_been_sent` instance variable inside the `PostsController` in just a moment, it will determine if an initial message to a user was already sent, or not.

Create partial files, corresponding to the `leave_message_partial_path` helper method, inside a new `contact_user` directory

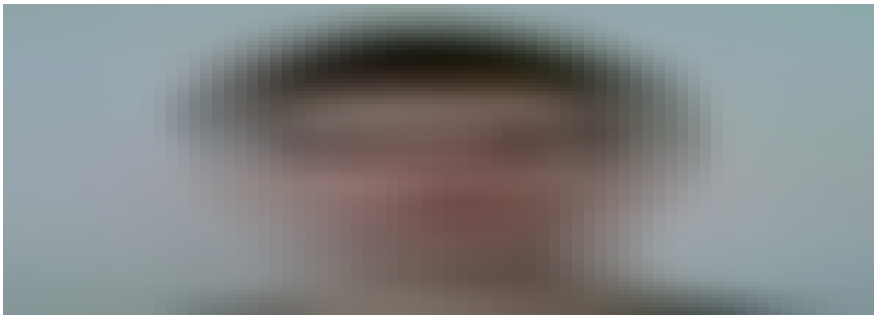


posts/show/contact_user/_already_in_touch.html.erb



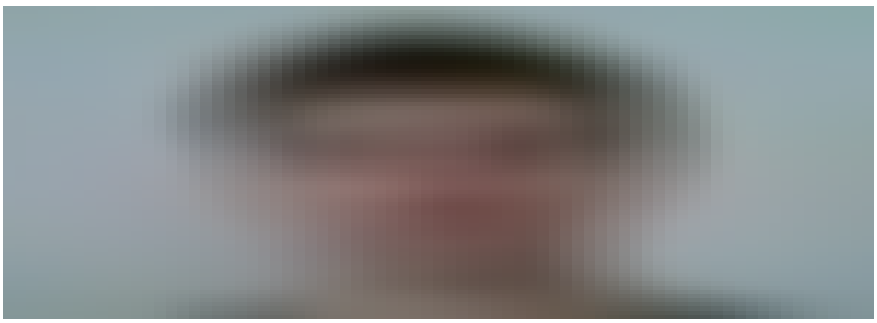
posts/show/contact_user/_message_form.html.erb

Now configure the `PostsController`'s `show` action. Inside the action add



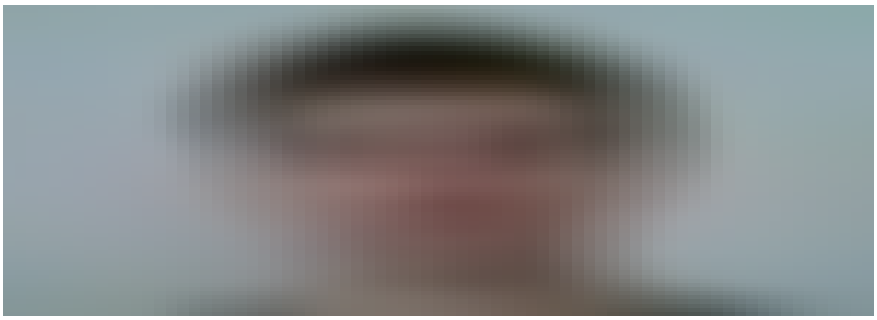
controllers/posts_controller.rb

Within the controller's `private` scope, define the `conversation_exist?` method



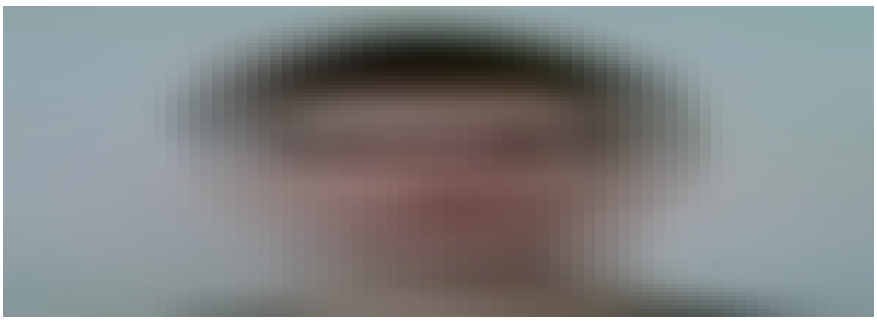
controllers/posts_controller.rb

The `between_users` method queries private conversations between two users. Define it as a scope inside the `Private::Conversation` model



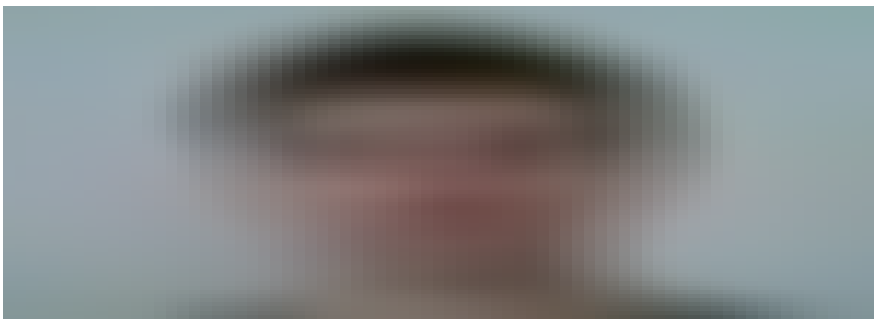
models/private/conversation.rb

We have to test if the scope works. Before writing specs, define a `private_conversation` factory, because we'll need sample data inside the test database.



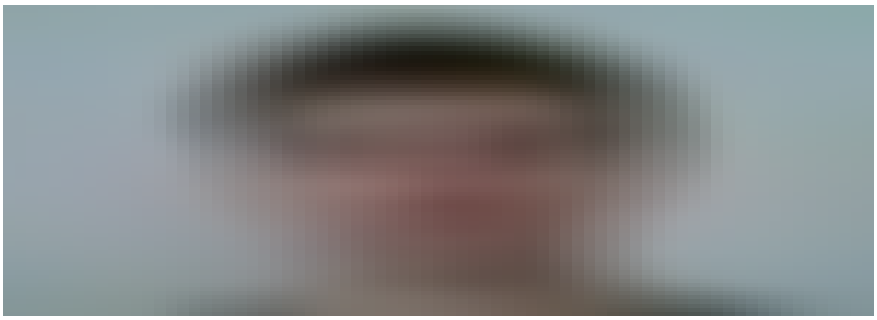
spec/factories/private_conversations.rb

We see a nested factory here, this allows to create a factory with its parent's configuration and then modify it. Also because we'll create messages with the `private_conversation_with_messages` factory, we need to define the `private_message` factory too



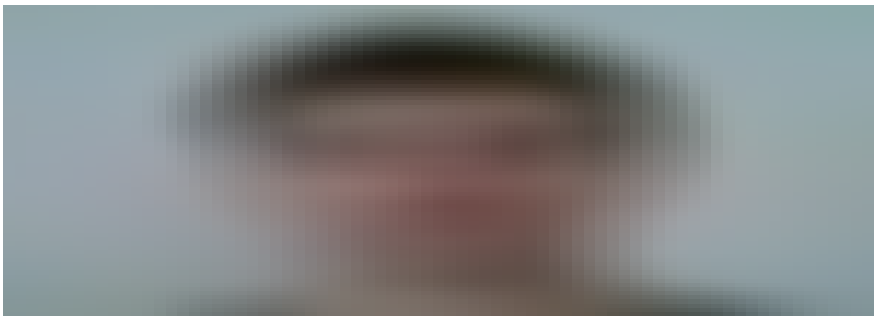
spec/factories/private_messages.rb

Now we have everything ready to test the `between_users` scope with specs.



spec/models/private/conversation_spec.rb

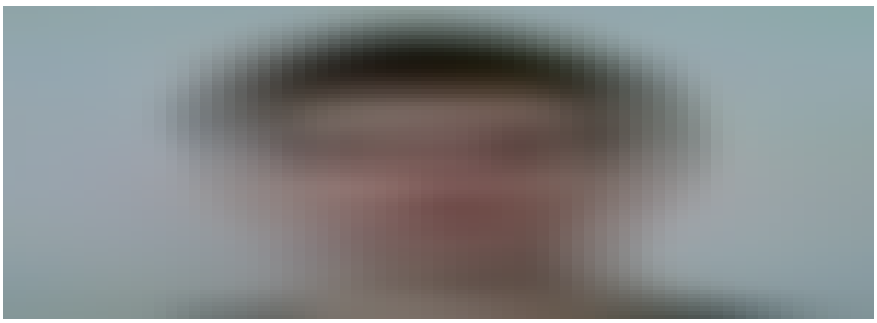
Define the `create` action for the `Private::Conversations` controller



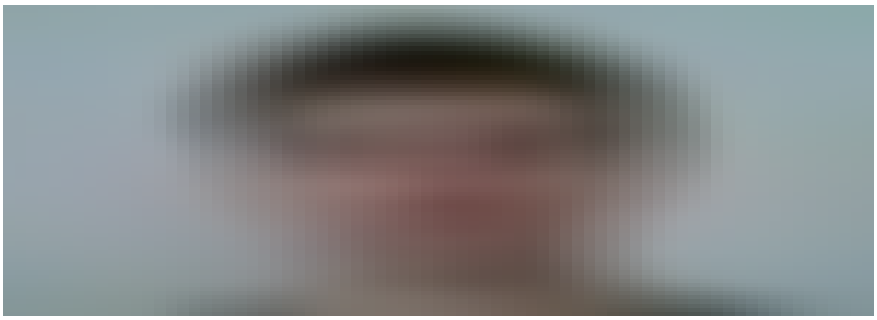
controllers/private/conversation_controller.rb

Here we create a conversation between a post's author and a current user. If everything goes well, the app will create a message, written by a current user, and give a feedback by rendering a corresponding JavaScript partial.

Create these partials

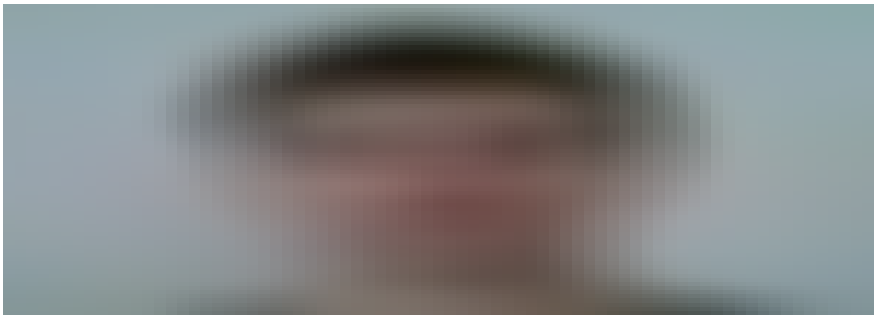


posts/show/contact_user/message_form/_success



posts/show/contact_user/message_form/_fail

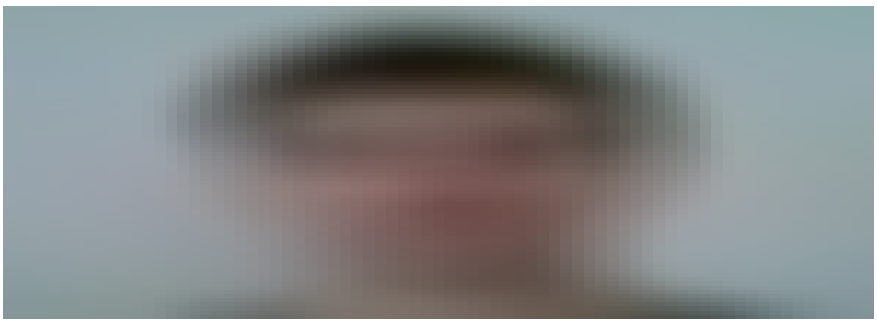
Create routes for the `Private::Conversations` and `Private::Messages` controllers



routes.rb

For now we'll only need few actions, this is where the `only` method is handy. The `namespace` method allows to easily create routes for namespaced controllers.

Test the overall `.contact-user` form's performance with feature specs



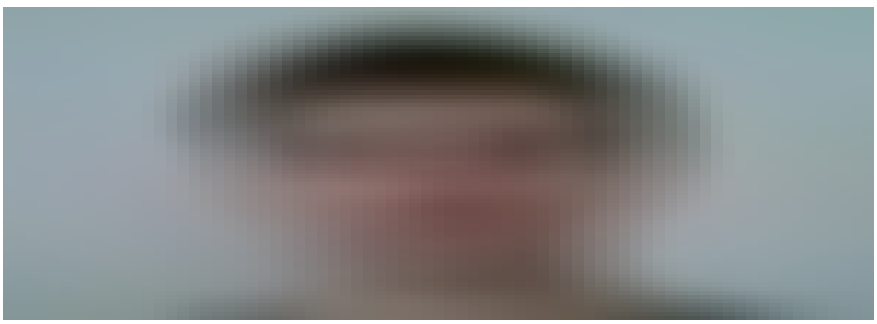
spec/features/posts/contact_user_spec.rb

Commit the changes

```
git add -A
git commit -m "Inside a post add a form to contact a user
```

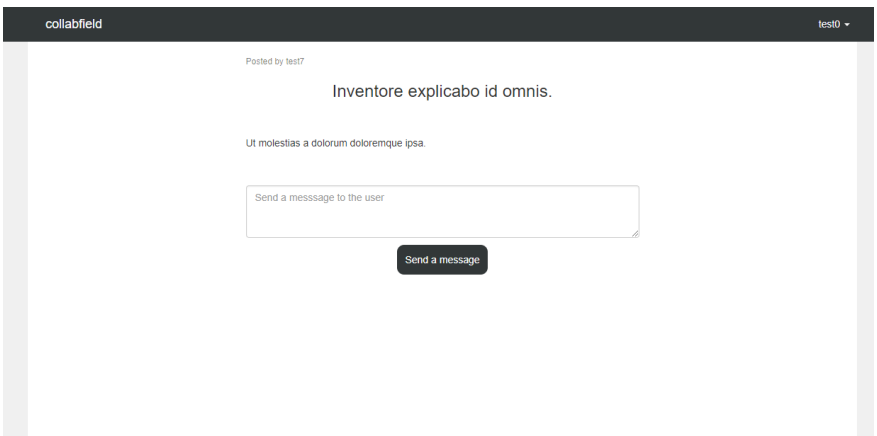
```
- Define a contact_user_partial_path helper method in
PostsHelper.
  Add specs for the method
- Create _contact_user.html.erb and _login_required.html.erb
partials
- Define a leave_message_partial_path helper method in
PostsHelper.
  Add specs for the method
- Create _already_in_touch.html.erb and
_message_form.html.erb
  partial files
- Define a @message_has_been_sent in PostsController's show
action
- Define a between_users scope inside the
Private::Conversation model
  Add specs for the scope
- Define private_conversation and private_message factories
- Define routes for Private::Conversations and
Private::Messages
- Define a create action inside the Private::Conversations
- Create _success.js and _fail.js partials
- Add feature specs to test the overall .contact-user form"
```

Change the form's style a little bit by adding CSS to the `branch_page.scss` file

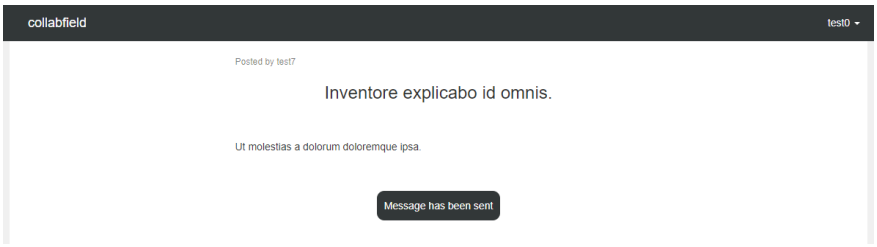


stylesheets/partials/posts/branch_page.scss

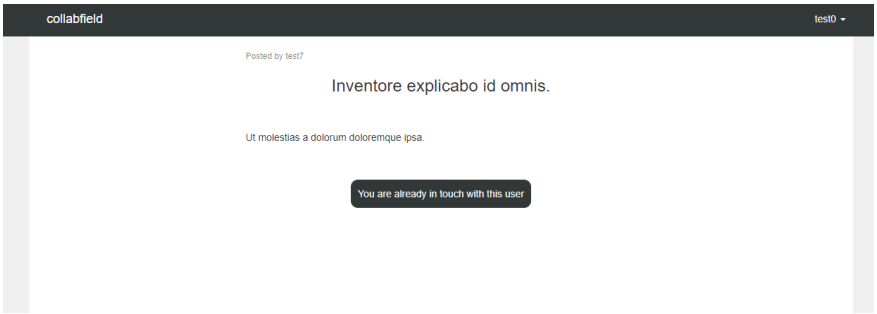
When you visit a single post, the form should look something like this



When you send a message to a post's author, the form disappears



That's how it looks like when you are already in touch with a user



Commit the changes

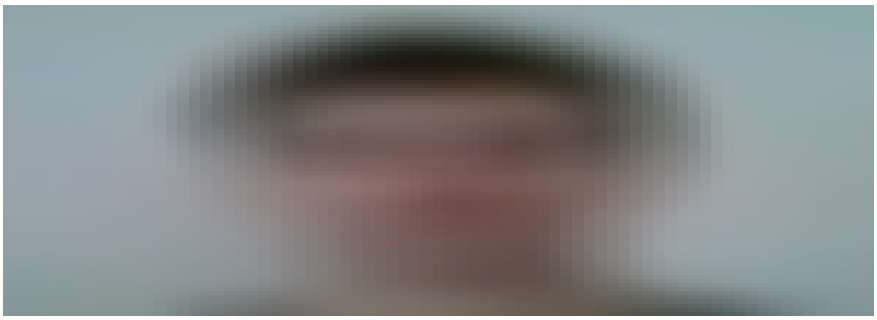
```
git add -A
git commit -m "Add CSS to style the .contact-user form"
```

Render a conversation window

We sent a message and created a new conversation. That is the only our power right now, we cannot do anything else. What a useless power thus far. We need a conversation window to read and write messages.

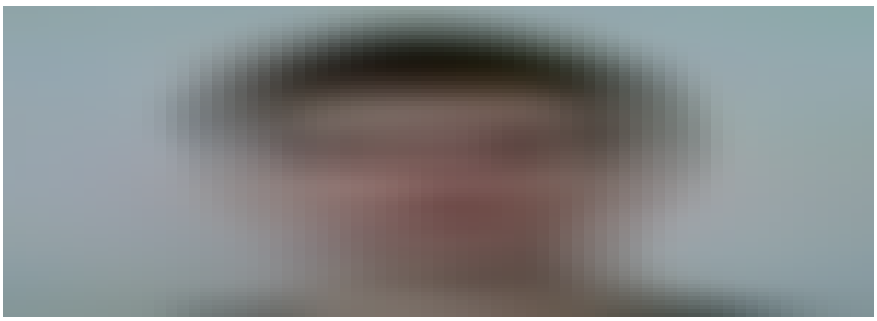
Store opened conversations' ids inside the session. This allows to keep conversations opened in the app until a user closes them or destroys the session.

Inside the `Private::ConversationsController`'s `create` action call a `add_to_conversations unless already_added?` method if a conversation is successfully saved. Then define the method within the `private` scope



controllers/private/conversations_controller.rb

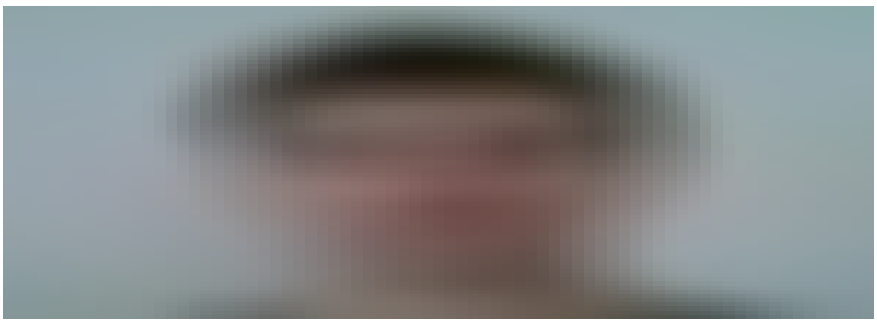
This will store the conversation's id inside the session. And the `already_added?` private method is going to make sure that the conversation's id isn't added inside the session yet.



controllers/private/conversations_controller.rb

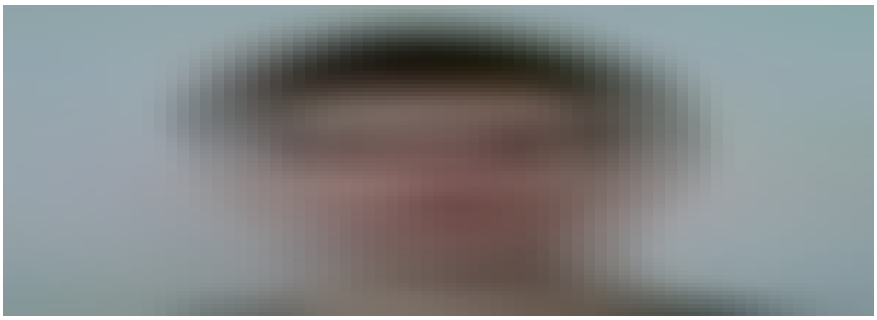
And lastly, we'll need an access to the conversation inside the views, so convert the `conversation` variable into an instance variable.

Now we can start building a template for the conversation window. Create a partial file for the window



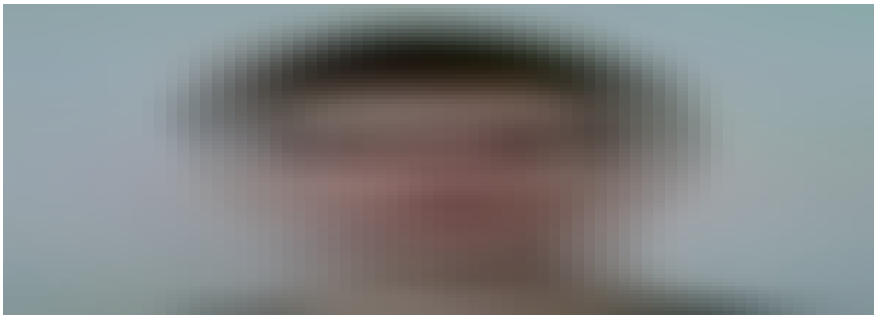
private/conversations/_conversation.html.erb

Here we get the conversation's recipient with the `private_conv_recipient` method. Define the helper method inside the `Private::ConversationsHelper`



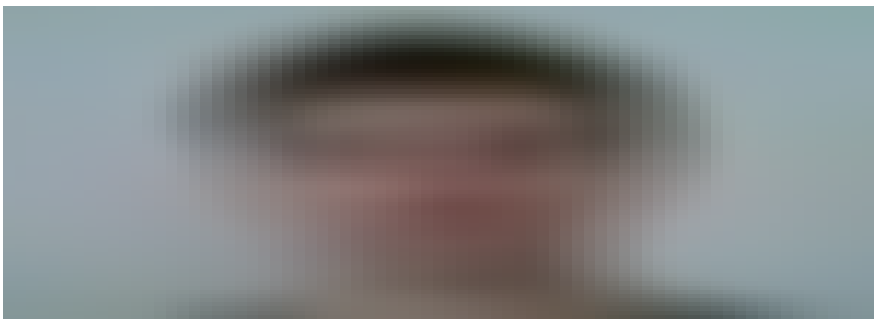
helpers/private/conversations_helper.rb

The `opposed_user` method is used. Go to the `Private::Conversation` model and define the method



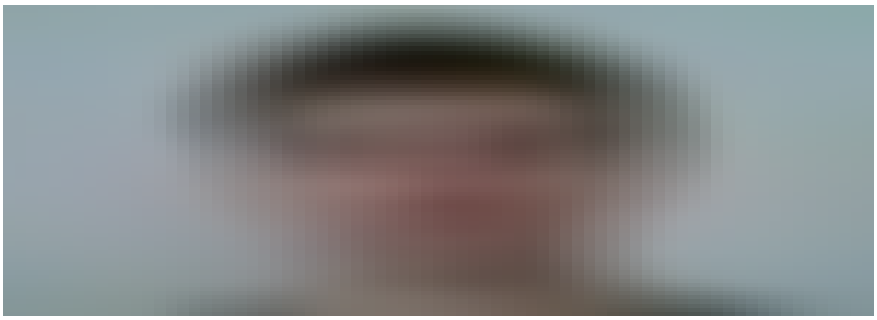
models/private/conversation.rb

This will return an opposed user of a private conversation. Make sure that the method works correctly by covering it with specs

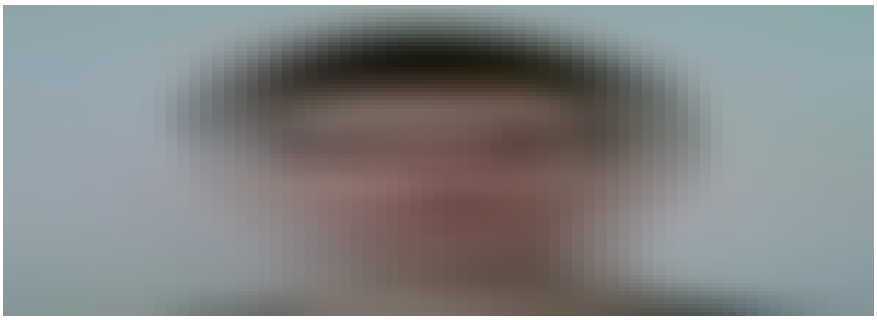


spec/models/private/conversation_spec.rb

Next, create missing partial files for the `_conversation.html.erb` file

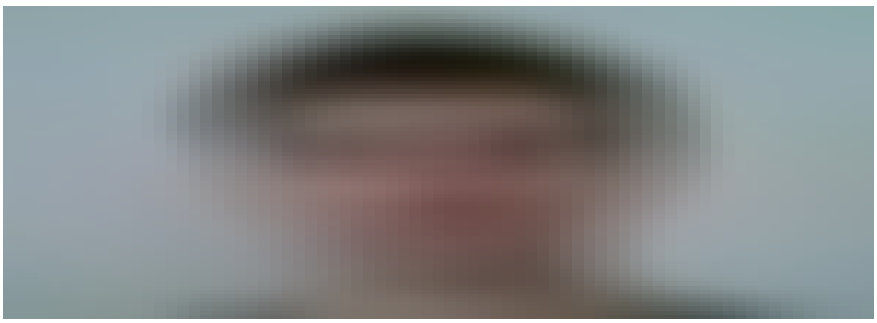


private/conversations/conversation/_heading.html.erb



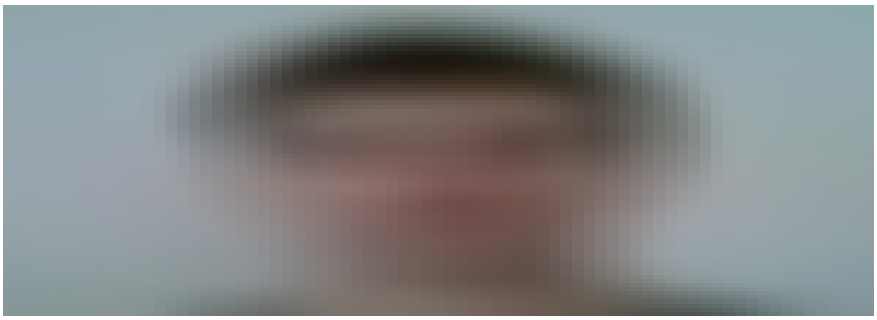
private/conversations/conversation/_messages_list.html.erb

Inside the `Private::ConversationsHelper`, define the `load_private_messages` helper method



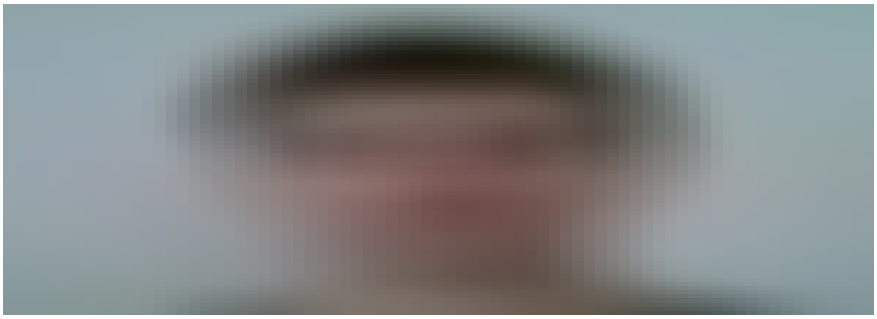
helpers/private/conversations_helper.rb

This will add a link to load previous messages. Create a corresponding partial file inside a new `messages_list` directory



private/conversations/conversation/messages_list/_link_to_previous_messages.html.erb

Don't forget to make sure that everything is fine with the method and write specs for it

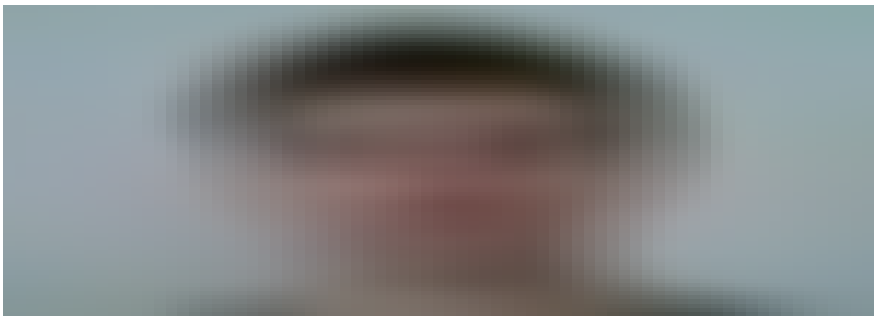


spec/helpers/private/conversations_helper_spec.rb

Because conversations' windows are going to be rendered throughout the whole app, it means we'll need an access to `Private::ConversationsHelper` helper methods. To have an access to all these methods across the whole app, inside the `ApplicationHelper` add

```
include Private::ConversationsHelper
```

Then create the last missing partial file for the conversation's new message form



private/conversations/conversation/_new_message_form.html.erb

We'll make this form functional a little bit later.

Now let's create a feature that after a user sends a message through an individual post, the conversation window gets rendered on the app.

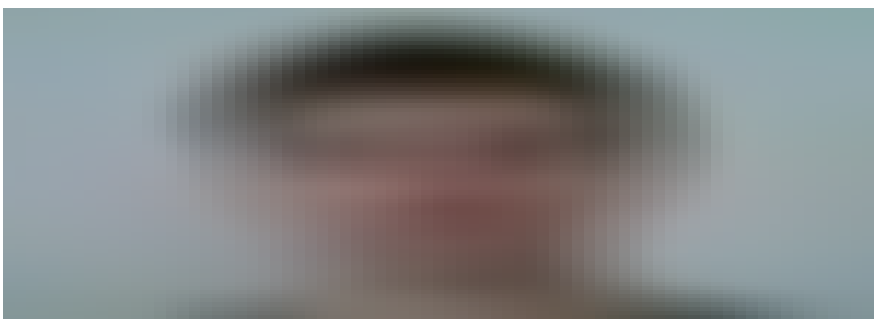
Inside the `_success.js.erb` file

```
posts/show/contact_user/message_form/_success.js.erb
```

add

```
<%= render 'private/conversations/open' %>
```

This partial file's purpose is to add a conversation window to the app. Define the partial file

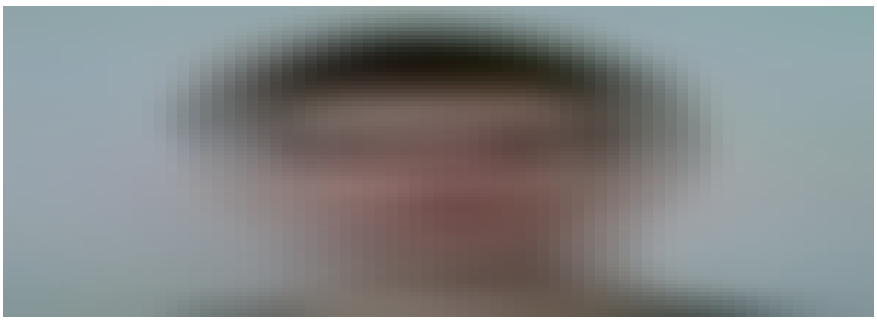


private/conversations/_open.js.erb

This callback partial file is going to be reused in multiple scenarios. To avoid rendering the same window multiple times, before rendering a window we check if it already exists on the app. Then we expand the

window and auto focus the message form. At the bottom of the file, the `positionChatWindows()` function is called to make sure that all conversations' windows are well positioned. If we didn't position them, they would just be rendered at the same spot, which of course would be unusable.

Now in the `assets` directory create a file which will take care of the conversations' windows visibility and positioning



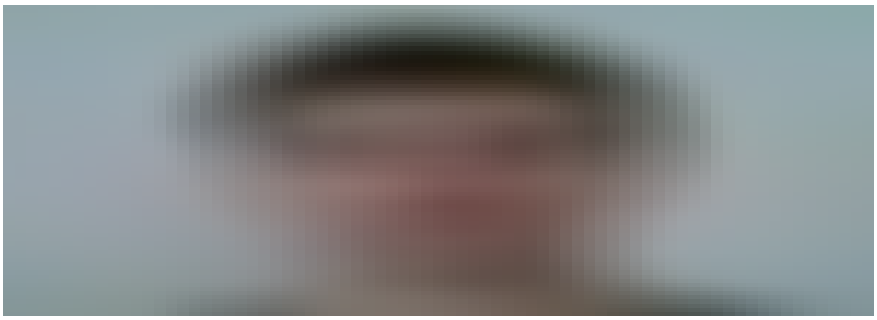
assets/javascripts/conversations/position_and_visibility.js

Instead of creating our own functions for setting and getting cookies or similar way to manage data between JavaScript, we can use the gon gem. An original usage of this gem is to send data from the server side to JavaScript. But I also find it useful for keeping track of JavaScript variables across the app. Install and set up the gem by reading the instructions.

We keep track of the viewport's width with an event listener. When a conversation gets close to the viewport's left side, the conversation gets hidden. Once there is enough of free space for a hidden conversation window, the app displays it again.

On a page visit we call the positioning and visibility functions to make sure that all conversations' windows are in right positions.

We're using the bootstrap's panel component to easily expand and collapse conversations' windows. By default they are going to be collapsed and not interactive at all. To make them toggleable, inside the `javascripts` directory create a new file `toggle_window.js`

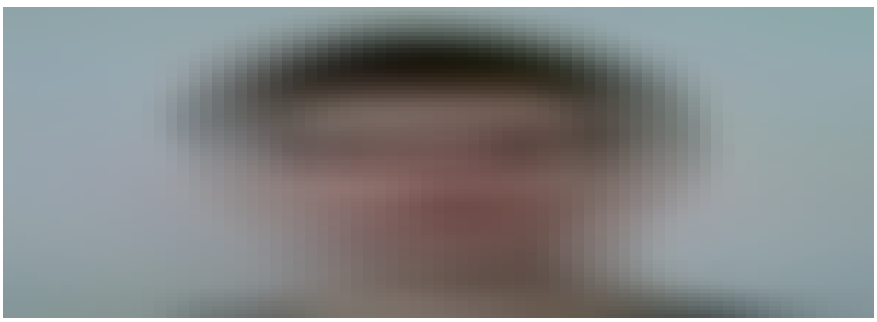


javascripts/conversations/toggle_window.js

Create a new `conversation_window.scss` file

```
assets/stylesheets/partials/conversation_window.scss
```

And add CSS to style conversations' windows



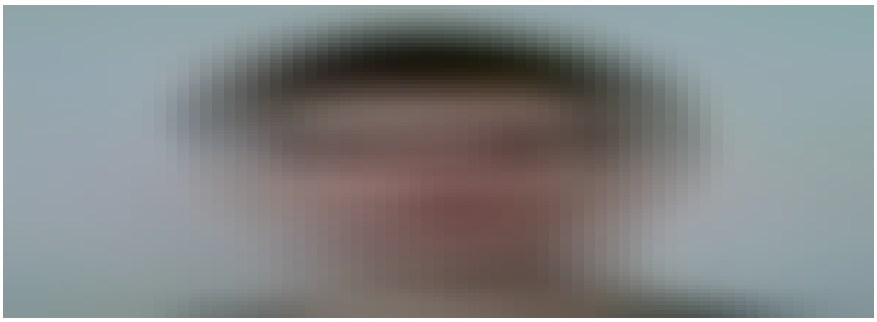
assets/stylesheets/partials/conversation_window.scss

You might noticed that there are some classes that haven't been defined yet in any HTML file. That's because the future files, we'll create in the `views` directory, are going to have shared CSS with already existent HTML elements. Instead of jumping back and forth to CSS files multiple times after we add any minor HTML element, I have included some classes, defined in future HTML elements, right now. Remember, you can always go to style sheets and analyze how a particular styling works.

Previously we've saved an id of a newly created conversation inside the session. It's time to take an advantage of it and keep the conversation window opened until a user closes it or destroys the session. Inside the `ApplicationController` define a filter

```
before_action :opened_conversations_windows
```

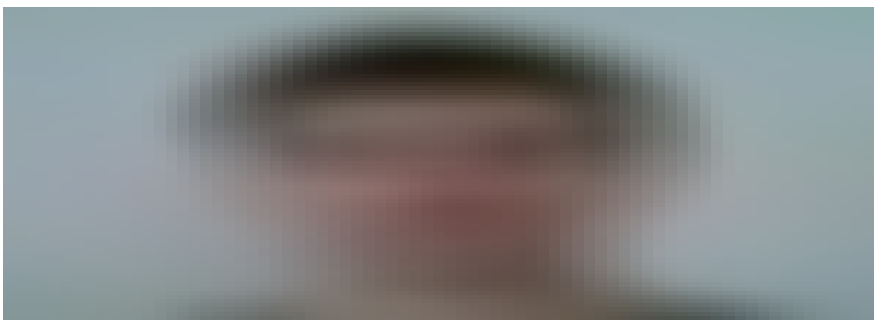
and then define the `opened_conversations_windows` method



controllers/application_controller.rb

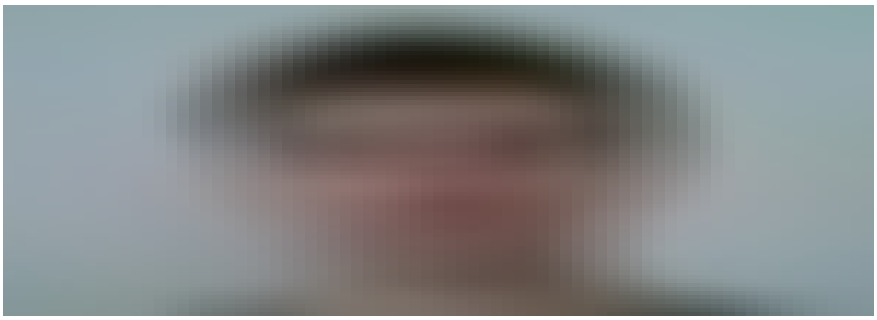
The `includes` method is used to include the data from associated database tables. In the near future we'll load messages from a conversation. If we didn't use the `includes` method, we wouldn't have loaded messages records of a conversation with this query. This would lead to a N + 1 query problem. If we didn't load messages with the query, an additional query would be fired for every message. This would significantly impact performance of the app. Now instead of 100 queries for 100 messages, we have an only one initial query for any number of messages.

Inside the `application.html.erb` file, just below the `yield` method, add



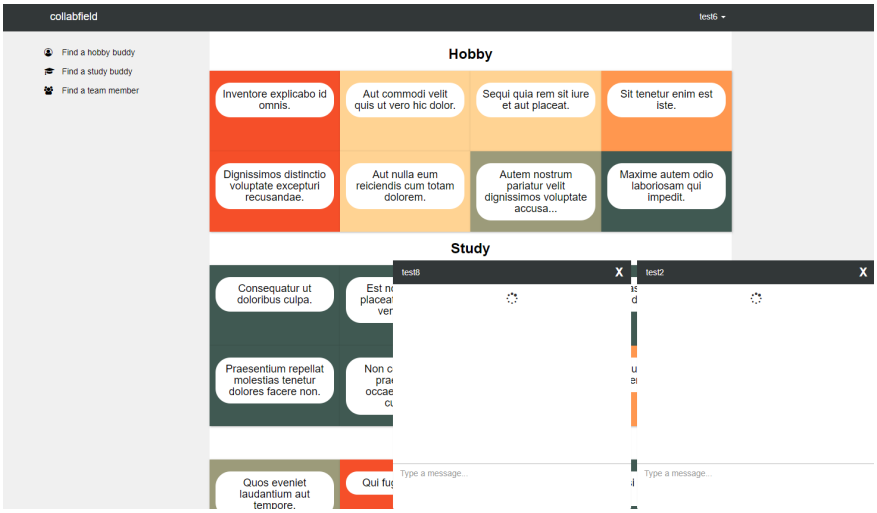
layouts/application.html.erb

Create a new `application` directory and inside create the `_private_conversations_windows.html.erb` partial file



layouts/application/_private_conversations_windows.html.erb

Now when we browse through the app, we see opened conversations all the time, no matter what page we are in.



Commit the changes

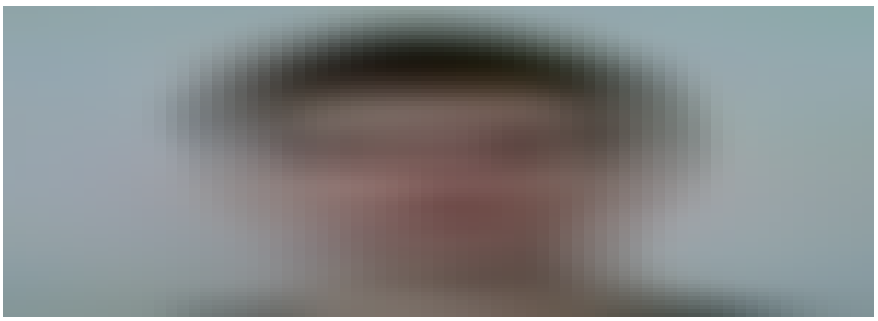
```
git add -A
git commit -m "Render a private conversation window on the
app
```

```
- Add opened conversations to the session
- Create a _conversation.html.erb file inside
private/conversations
- Define a private_conv_recipient helper method in the
  private/conversations_helper.rb
- Define an opposed_user method in Private::Conversation
model
  and add specs for it
- Create _heading.html.erb and _messages_list.html.erb files
  inside the private/conversations/conversation
- Define a load_private_messages in
private/conversations_helper.rb
  and add specs for it
- Create a _new_message_form.html.erb inside the
  private/conversations/conversation
- Create a _open.js.erbinside private/conversations
- Create a  position_and_visibility.js inside the
```

```
    assets/javascripts/conversations
  - Create a  conversation_window.scss inside the
    assets/stylesheets/partials
  - Define an opened_conversations_windows helper method in
    ApplicationController
  - Create a _private_conversations_windows.html.erb inside
  the
    layouts/application
```

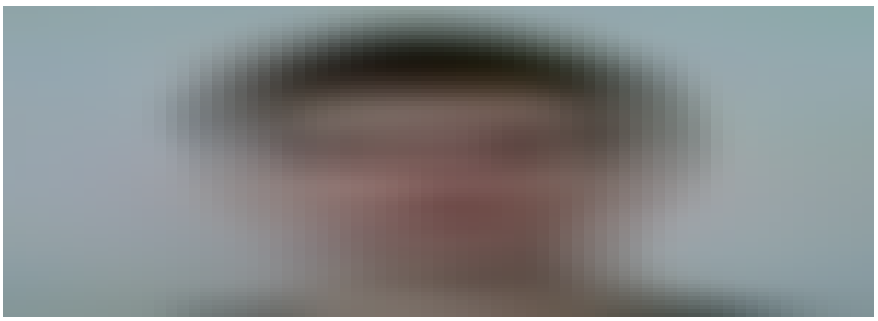
Close a conversation

The conversation's close button isn't functional yet. But we have everything ready to make it so. Inside the `Private::ConversationsController`, define a `close` action



controllers/private/conversations_controller.rb

When the close button is clicked, this action will be called. The action deletes conversation's id from the session and then responds with a js partial file, identical to the action's name. Create the partial file



private/conversations/close.js.erb

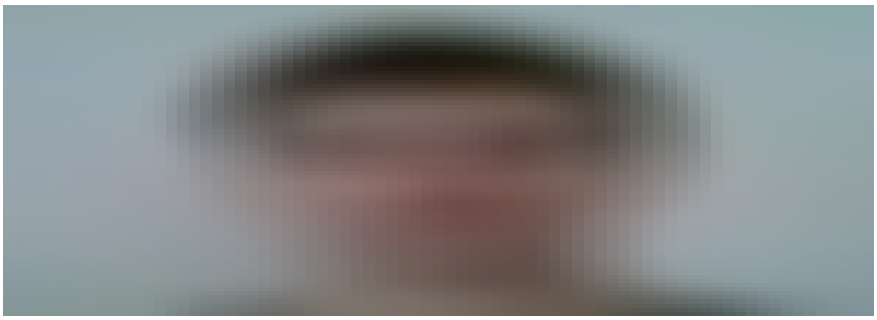
It removes the conversation's window from the DOM and re-positions the rest of conversations' windows.

Commit the changes

```
git add -A
git commit -m "Make the close conversation button functional

- Define a close action inside the
Private::ConversationsController
- Create a close.js.erb inside the private/conversations"
```

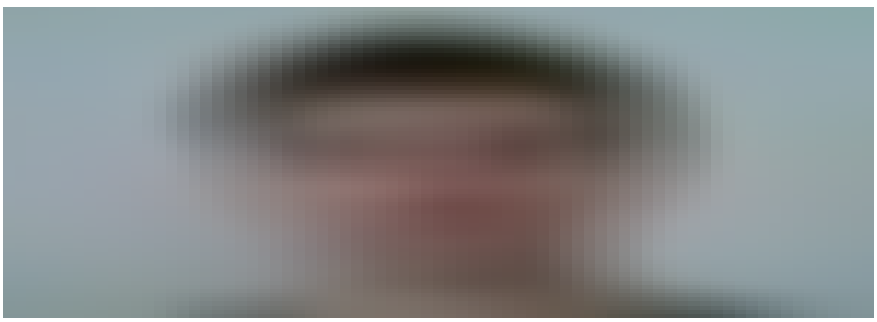
Render messages

Currently in the messages list we see a loading icon without any messages. That's because we haven't created any templates for messages. Inside the `views/private` directory, create a `messages` directory. Inside the directory, create a new file



private/messages/_message.html.erb

The `private_message_date_check` helper method checks if this message is written at the same day as a previous message. If not, it renders an extra line with a new date. Define the helper method inside the `Private::MessagesHelper`

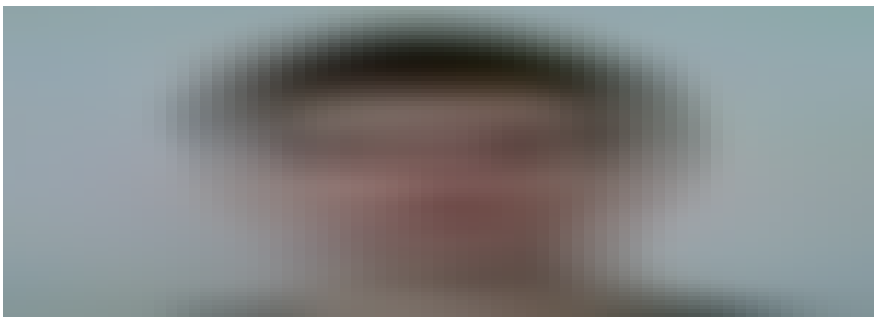


helpers/private/messages_helper.rb

Inside the `ApplicationHelper`, include the `Private::MessagesHelper`, so we could have an access to it across the app

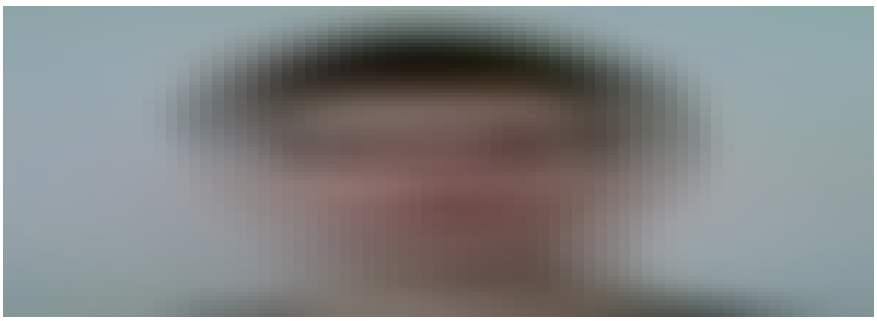
```
include Private::MessagesHelper
```

Write specs for the method. Create a new `messages_helper_spec.rb` file



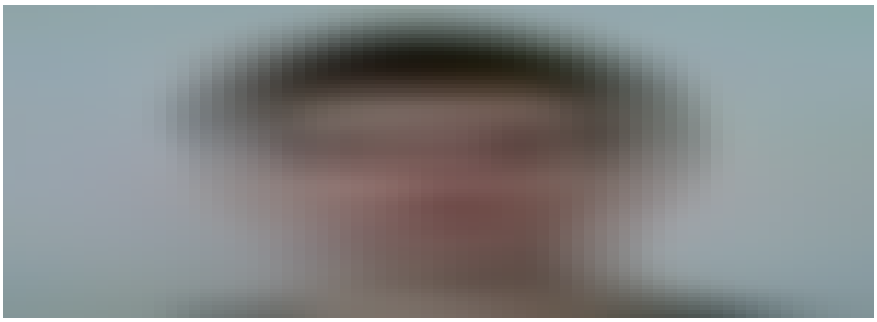
spec/helpers/private/messages_helper_spec.rb

Inside a new `message` directory, create a `_new_date.html.erb` file



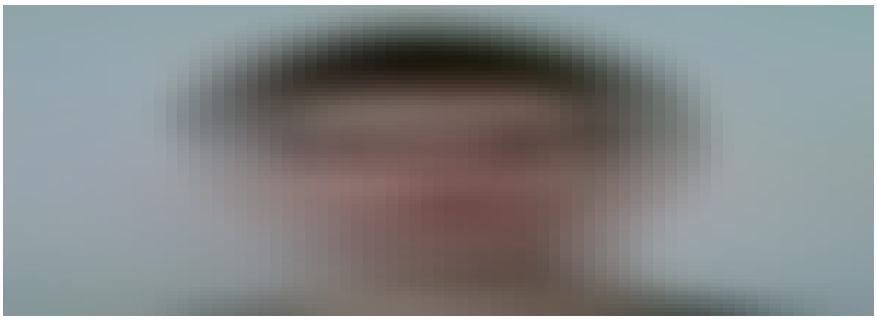
private/messages/message/_new_date.html.erb

Then inside the `_message.html.erb` file, we have `sent_or_received` and `seen_or_unseen` helper methods. They return different classes in different cases. Define them inside the `Private::MessagesHelper`



helpers/private/messages_helper.rb

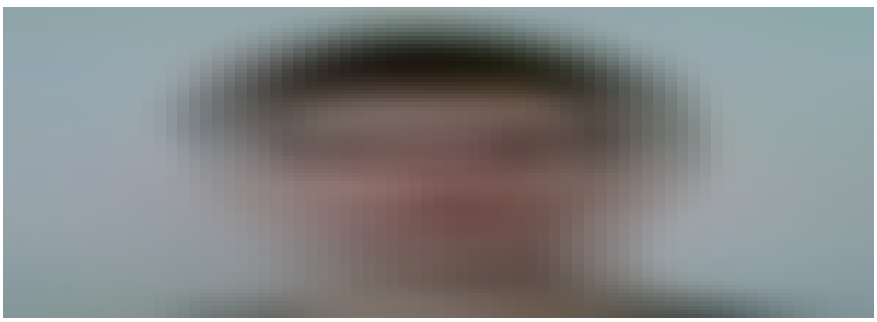
Write specs for them:



spec/helpers/private/messages_helper_spec.rb

Now we need a component to load messages into the messages list. Also this component is going to add previous messages at the top of the list, when a user scrolls up, until there is no messages left in a conversation. We are going to have an infinite scroll mechanism for messages, similar to the one we have in posts' pages.

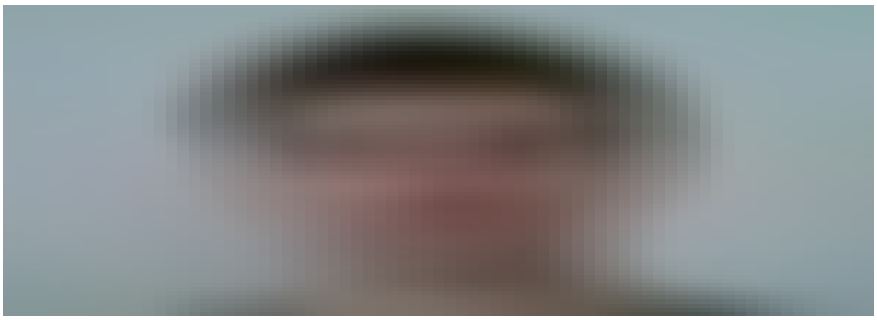
Inside the `views/private/messages` directory create a `_load_more_messages.js.erb` file:



private/messages/_load_more_messages.js.erb

The `@id_type` instance variable determines a type of the conversation. In the future we will be able to create not only private conversations, but group too. This leads to common helper methods and partial files between both types.

Inside the `helpers` directory, create a `shared` directory. Create a `messages_helper.rb` file and define a helper method



helpers/shared/messages_helper.rb

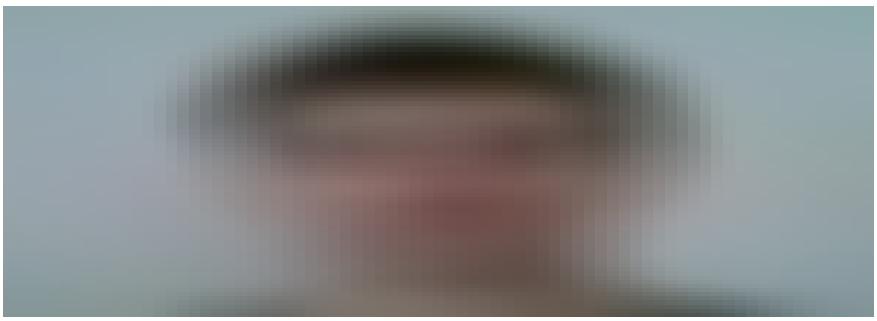
So far the method is pretty dumb. It just returns a partial's path. We'll give some intelligence to it later, when we'll build extra features to our messaging system. Right now we won't have an access to helper methods, defined in this file, in any other file. We have to include them inside other helper files. Inside the `Private::MessagesHelper`, include methods from the `Shared::MessagesHelper`

```
require 'shared/messages_helper'
include Shared::MessagesHelper
```

Inside the `shared` directory, create few new directories:

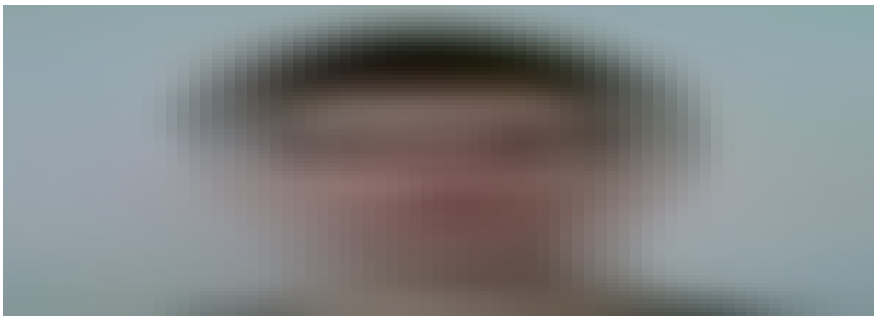
```
shared/load_more_messages/window
```

Then create an `_append_messages.js.erb` file:



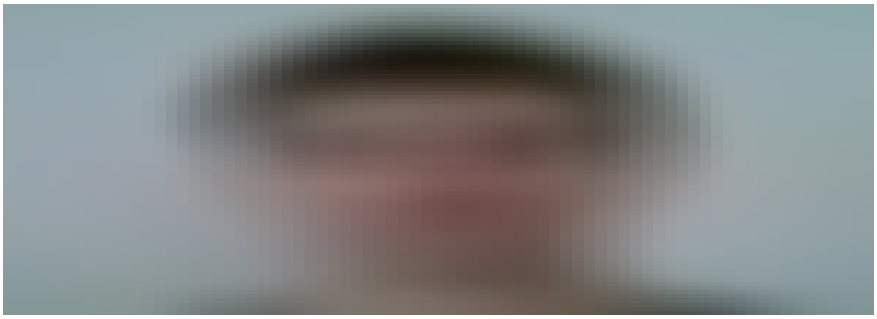
shared/load_more_messages/window/_append_messages.js.erb

This code takes care that previous messages get appended to the top of the messages list. Then define another, again, not that fascinating, helper method inside the `Private::MessagesHelper`



helpers/private/messages_helper.rb

Create the corresponding directories inside the `private/messages` directory and create a `_add_link_to_messages.js.erb` file

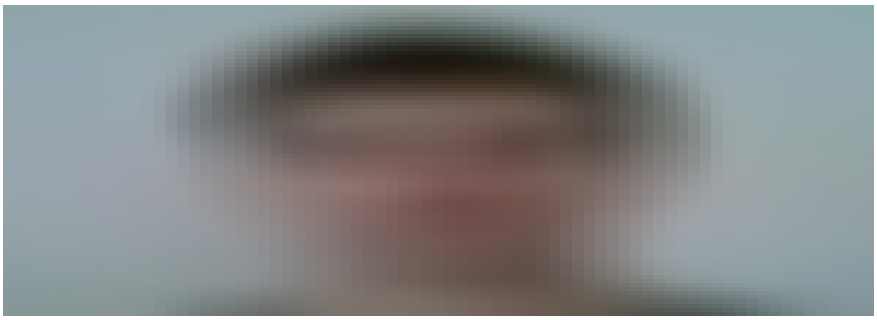


private/messages/load_more_messages/window/_add_link_to_messages.js.erb

This file is going to update the link which loads previous messages. After previous messages are appended, the link is replaced with an updated link to load older previous messages.

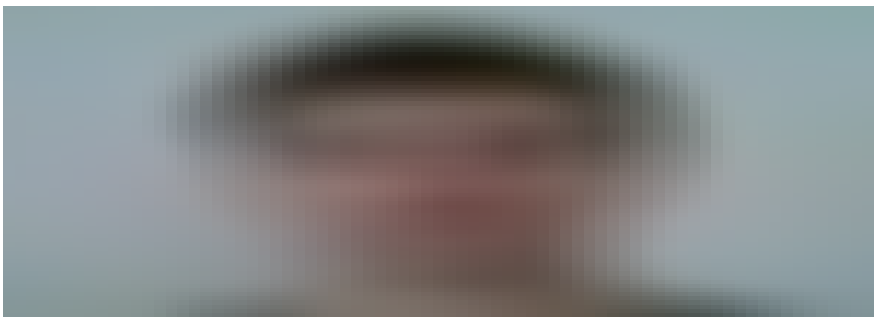
Now we have all this system, how previous messages get appended to the top of the messages list. But, if we tried to go to the app and opened a conversation window, we wouldn't see any rendered messages. Why? Because nothing triggers the link to load previous messages. When we open a conversation window for the first time, we want to see the most recent messages. We can program the conversation window in a way that once it gets expanded, the load more messages link gets triggered, to load the most recent messages. It initiates the first cycle of appending previous messages and replacing the load more messages link with an updated one.

Inside the `toggle_window.js` file update the `toggle` function to do exactly what is described above



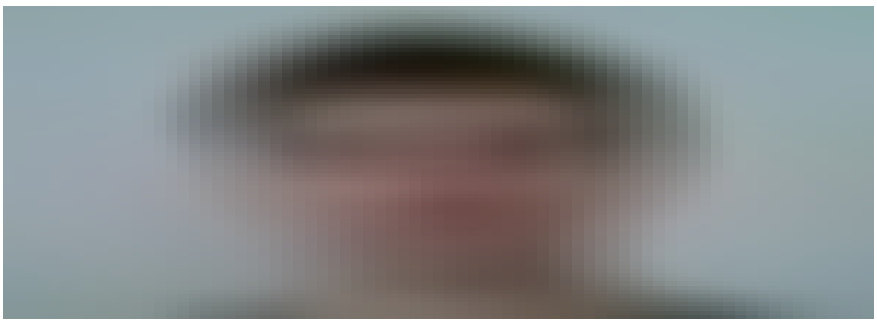
javascripts/conversations/toggle_window.js

Create an event handler, so whenever a user scrolls up and reaches almost the top of the messages list, the load more messages link will be triggered.



assets/javascripts/conversations/messages_infinite_scroll.js

When the load more messages link is going to be clicked, a `Private::MessagesController`'s `index` action gets called. That's the path, we defined to the load previous messages link. Create the controller and its `index` action



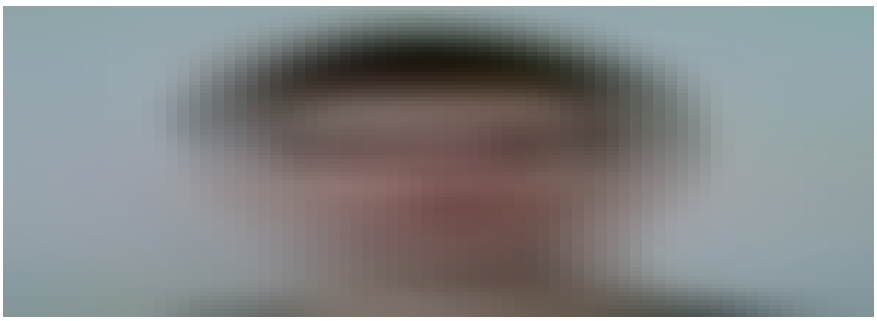
controllers/private/messages_controller.rb

Here we include methods from the `Messages` module. The module is stored inside the `concerns` directory. ActiveSupport::Concern is one of the places, where you can store modules which you can later use in classes. In our case we include extra methods to our controller from the module. The `get_messages` method comes from the `Messages`

module. The reason why it is stored inside the module is that we'll use this exact same method in another controller a little bit later. To avoid code duplication, we make the method more reusable.

I've seen some people complaining about the `ActiveSupport::Concern` and suggest not to use it at all. I challenge those people to fight me in the octagon. I'm kidding :D. This is an independent application and we can create our app however we like it. If you don't like `concerns`, there are bunch of other ways to create reusable methods.

Create the module



controllers/concerns/messages.rb

Here we require the `active_support/concern` and then extend our module with `ActiveSupport::Concern`, so Rails knows that it is a concern.

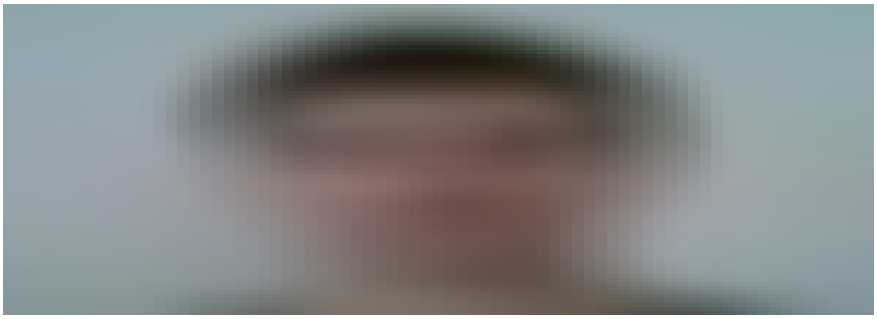
With the `constantize` method we dynamically create a constant name by inputting a string value. We call models dynamically. The same method is going to be used for `Private::Conversation` and `Group::Conversation` models.

After the `get_messages` method sets all necessary instance variables, the `index` action responds with the `_load_more_messages.js.erb` partial file.

Finally, after messages get appended to the top of the messages list, we want to remove the loading icon from the conversation window. At the bottom of the `_load_more_messages.js.erb` file add

```
<%= render remove_link_to_messages %>
```

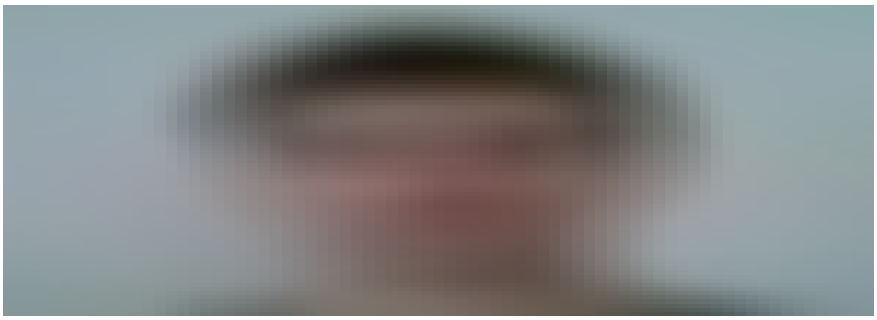
Now define the `remove_link_to_messages` helper method inside the `Shared::MessagesHelper`



helpers/shared/messages_helper.rb

Try to write specs for the method on your own.

Create the `_remove_more_messages_link.js.erb` partial file



shared/load_more_messages/window/_remove_more_messages_link.js.erb

Now in a case, where are no previous messages left, the link to previous messages and the loading icon will be removed.

If you try to contact a user now, a conversation window will be rendered with a message, you sent, inside. We're able to render messages via AJAX requests.

Commit the changes.

```
git add -A
git commit -m "Render messages with AJAX


- Create a _message.html.erb inside private/messages
- Define a private_message_date_check helper method in
  Private::MessagesHelper and write specs for it
- Create a _new_date.html.erb inside
private/messages/message
- Define sent_or_received and seen_or_unseen helper methods
in
  Private::MessagesHelper and write specs for them
- Create a _load_more_messages.js.erb inside
private/messages
- Define an  append_previous_messages_partial_path helper
method in
  Shared::MessagesHelper
- Create a _append_messages.js.erb inside
  shared/load_more_messages/window
- Define a  replace_link_to_private_messages_partial_path in
  Private::MessagesHelper
- Create a _add_link_to_messages.js.erb inside
  private/messages/load_more_messages/window
- Create a toggle_window.js inside javascripts/conversations
- Create a messages_infinite_scroll.js inside
  assets/javascripts/conversations
- Define an index action inside the
Private::MessagesController
- Create a messages.rb inside controllers/concerns
- Define a remove_link_to_messages inside helpers/shared
- Create a _remove_more_messages_link.js.erb inside
  shared/load_more_messages/window"
```

Real time functionality with Action Cable

Conversations' windows look pretty neat already. And they also have some sweet functionality. But, they are lacking the most important

feature—ability to send and receive messages in real time.

As briefly discussed previously, Action Cable will allow us to achieve the desired real time feature for conversations. You should skim through the documentation to be aware how it all works.

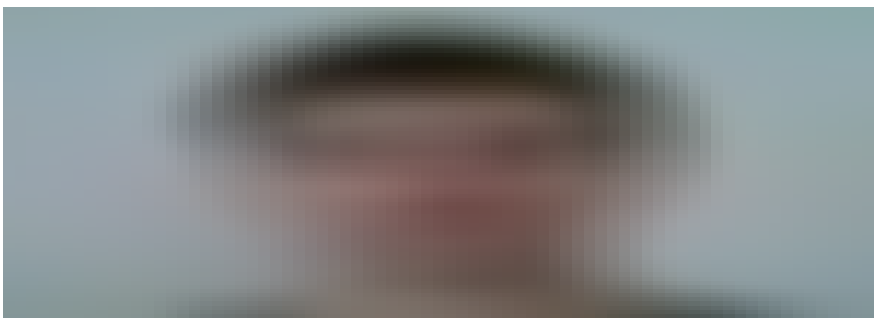
The first thing which we should do is create a WebSocket connection and subscribe to a specific channel. Luckily, WebSocket connections are already covered by default Rails configuration. Inside the `app/channels/application_cable` directory you see `channel.rb` and `connection.rb` files. The `Connection` class takes care of the authentication and the `Channel` class is a parent class to store shared logic between all channels.

Connection is set by default. Now we need a private conversation channel to subscribe to. Generate a namespaced channel

```
rails g channel private/conversation
```

Inside the generated `Private::ConversationChannel`, we see `subscribed` and `unsubscribed` methods. With the `subscribed` method a user creates a connection to the channel. With the `unsubscribed` method a user, obviously, destroys the connection.

Update those methods:

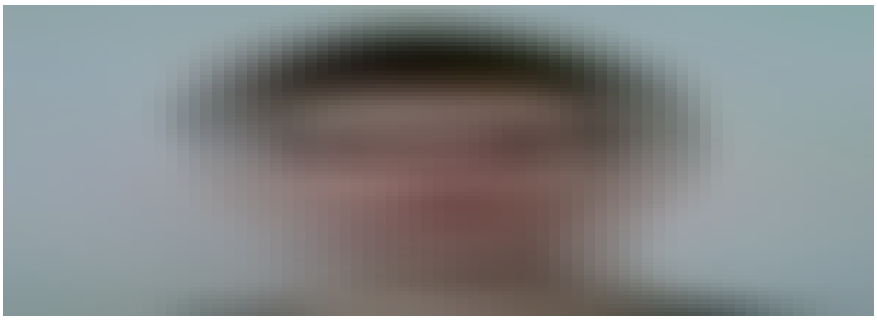


channels/private/conversation_channel.rb

Here we want that a user would have its own unique channel. From the channel a user will receive and send data. Because users' ids are unique, we make the channel unique by adding a user's id.

This is a server side connection. Now we need to create a connection on the client side too.

To create an instance of the connection on the client side, we have to write some JavaScript. Actually, Rails has already created it with the channel generator. Navigate to `assets/javascripts/channels/private` and by default Rails generates `CoffeeScript` files. I'm going to use JavaScript here. So rename the file to `conversation.js` and replace its content with:



assets/javascripts/channels/private/conversation.js

Restart the server, go to the app, login and check the server log.

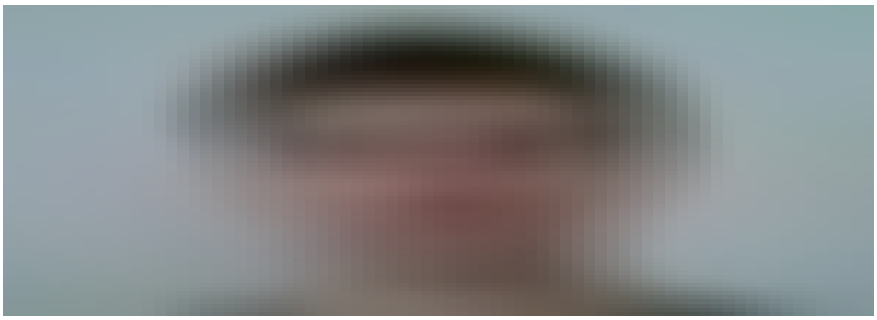
```
Private::ConversationChannel is transmitting the subscription confirmation
Private::ConversationChannel is streaming from private_conversations_1
```

We got the connection. The core of the real time communication is set. We've a constantly open client-server connection. It means that we can send and receive data from the server without restarting the connection or refreshing a browser, man! A really powerful thing when you think about it. From now we'll build the messaging system around this connection.

Commit the changes.

```
git add -A
git commit -m "Create a unique private conversation channel
and subscribe to it"
```

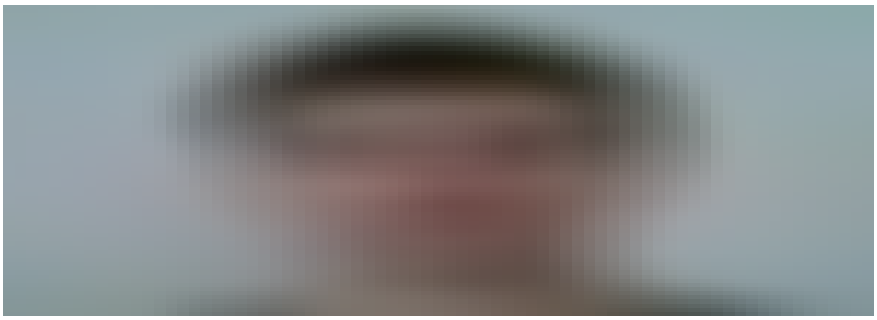
Let's make the conversation window's new message form functional. At the bottom of the `assets/javascripts/channels/private/conversations.js` file add this function:



assets/javascripts/channels/private/conversation.js

The function is going to get values from the new message form and pass them to a `send_message` function. The `send_message` function is going to call a `send_message` method on the server side, which will take care of creating a new message.

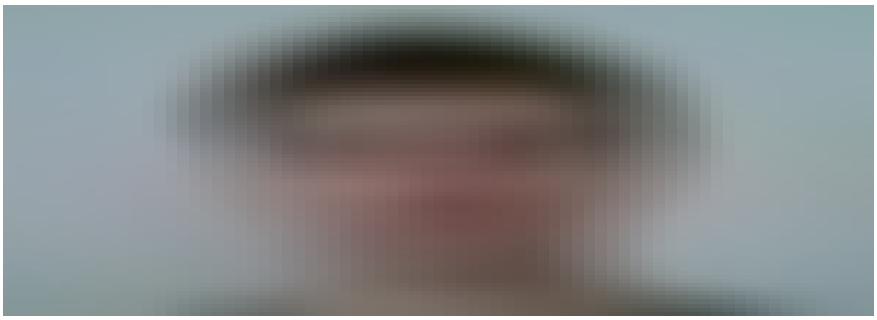
Also take a note, the event handler is on a submit button, but on the conversation window we don't have any visible submit buttons. It's a design choice. We have to program the conversation window in a way that submit button is triggered when the enter key is clicked on a keyboard. This function is going to be used in the future by other features, so create a `conversation.js` file inside the `assets/javascripts/conversations` directory



assets/javascripts/conversations/conversation.js

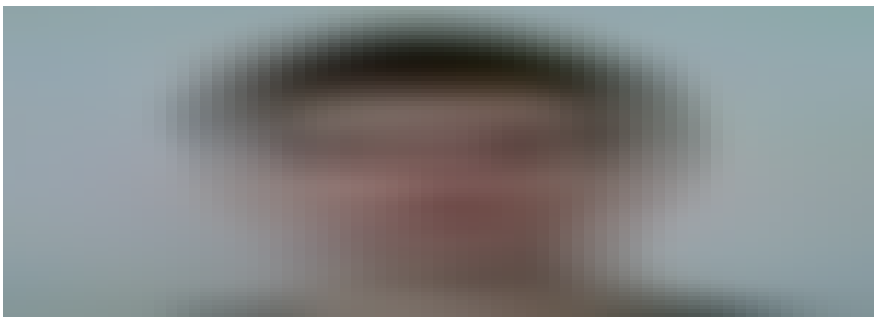
In the file we describe some general behavior for conversations' windows. The first behavior is to keep the scrollbar away from the top, so previous messages aren't loaded when it is not needed. The second function makes sure that the submit button is triggered on the enter key click and then cleans input's value back to an empty string.

Start by creating the `send_message` function inside the `private_conversation` object. Add it below the `received` callback function



assets/javascripts/channels/private/conversation.js

This calls the `send_message` method on the server side and passes the message value. The server side method should be defined inside the `Private::ConversationChannel`. Define the method:



channels/private/conversation_channel.rb

This will take care of a new message's creation. The `data` parameter, which we get from the passed argument, is a nested hash. So to reduce this nested complexity into a single hash, the `each_with_object` method is used.

If you try to send a new message inside a conversation's window, a new message record will actually be created. It won't show up on the conversation window instantly yet, only when you refresh the website. It would show up, but we haven't set anything to broadcast newly created messages to a private conversation's channel. We'll implement it in just a moment. But before we continue and commit changes, quickly recap how the current messaging system works.

1. A user fills the new message form and submits the message
2. The event handler inside the `javascripts/channels/private/conversations.js` gets a conversation window's data, a conversation id and a message value, and triggers the channel instances on the client-side `send_message` function.

3. The `send_message` function on the client side calls the `send_message` method on the server side and passes data to it

4. The `send_message` method on the client side processes provided data and creates a new `Private::Message` record

Commit the changes.

```
git add -A
git commit -m "Make a private conversation window's new
message form functional

- Add an event handler inside the
  javascripts/channels/private/conversation.js to trigger
the submit button
- Define a common behavior among conversation windows inside
the
  assets/javascripts/conversations/conversation.js
- Define a send_message function on both, client and server,
sides"
```

Broadcast a new message

After a new message is created, we want to broadcast it to a corresponding channel somehow. Well, Active Record Callbacks arms us with plenty of useful callback methods for models. There is a `after_create_commit` callback method, which runs whenever a new model's record gets created. Inside the `Private::Message` model's file add

```
1  ...
2  after_create_commit do
3    Private::MessageBroadcastJob.perform_later(self, pre
4  end
5
6  def previous_message
7    previous_message_index = self.conversation.messages.
```

models/private/message.rb

As you see, after a record's creation, the `Private::MessageBroadcastJob.perform_later` gets called. And what's that? It's a background job, handling back-end operations. It allows to run certain operations whenever we want to. It could be immediately

after a particular event, or be scheduled to run some time later after an event. If you aren't familiar with background jobs, checkout Active Job Basics.

Add specs for the `previous_message` method. If you are going to try run specs now, comment out the `after_create_commit` method. We haven't defined the `Private::MessageBroadcastJob`, so currently specs would raise an undefined constant error.

```
...
context 'Methods' do
  it 'gets a previous message' do
    conversation = create(:private_conversation)
    message1 = create(:private_message, conversation_i
    message2 = create(:private_message, conversation_i
    expect(message2.previous_message).to eq message1
```

spec/models/private/message_spec.rb

Now we can create a background job which will broadcast a newly created message to a private conversation's channel.

```
rails g job private/message_broadcast
```

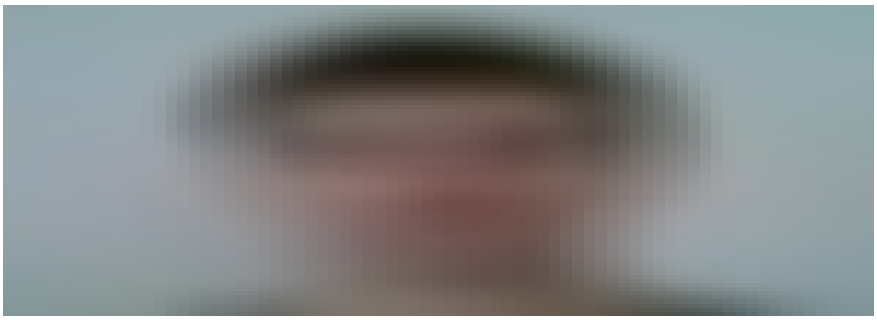
Inside the file we see a `perform` method. By default, when you call a job, this method is called. Now inside the job, process the given data and broadcast it to channel's subscribers.

```
class Private::MessageBroadcastJob < ApplicationJob
  queue_as :default

  def perform(message, previous_message)
    sender = message.user
    recipient = message.conversation.opposed_user(send

    broadcast_to_sender(sender, recipient, message, pr
    broadcast_to_recipient(sender, recipient, message,
  end

  private

  def broadcast_to_sender(sender, recipient, message,
    ActionCable.server.broadcast(
      "private_conversations_#{sender.id}",
      message: render_message(message, previous_messag
      conversation_id: message.conversation_id,
      recipient_info: recipient
    )
  end

  def broadcast_to_recipient(sender, recipient, messag
    ActionCable.server.broadcast(
      "private_conversations_#{recipient.id}",
      recipient: true,
      sender info: sender.
```

jobs/private/message_broadcast_job.rb

Here we render a message and send it to both channel's subscribers. Also we pass some additional key-value pairs to properly display the message. If we tried to send a new message, users would receive data, but the message wouldn't be appended to the messages list. No visible changes would be made.

When data is broadcasted to a channel, the `received` callback function on the client side gets called. This is where we have an opportunity to append data to the DOM. Inside the `received` function add the following code:



assets/javascripts/channels/private/conversation.js

Here we see that the sender and the recipient get treated a little bit differently.

```
// change style of conv window when there are unseen
messages
// add an additional class to the conversation's window or
something
```

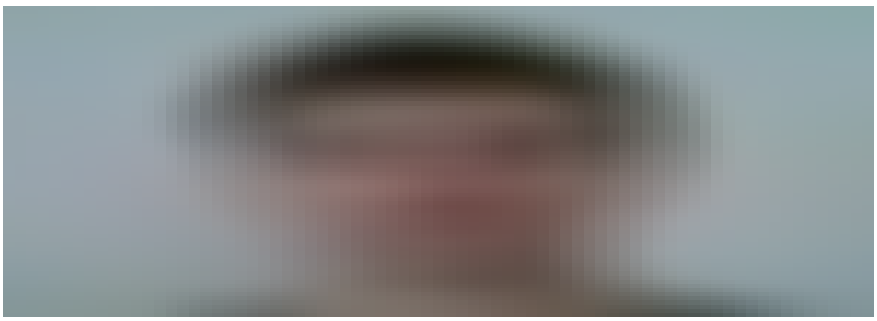
I've created this intentionally, so whenever a conversation has unseen messages, you could style its window however you like it. You can change a window's color, make it blink, or whatever you want to.

Also there are `findConv`, `ConvRendered`, `ConvMessagesVisibility` functions used. We'll use these functions for both type of chats, private and group.

Create a `shared` directory:

```
assets/javascripts/channels/shared
```

Create a `conversation.js` file inside this directory.



assets/javascripts/channels/shared/conversation.js

A messenger is mentioned in the code quite a lot and we don't have the messenger yet. The messenger is going to be a separate way to open conversations. To prevent a lot of small changes in the future, I've included cases with the messenger right now.

That's it, the real time functionality should work. Both users, the sender and the recipient, should receive and get displayed new messages on the DOM. When we send a new message, we should see it instantly appended to the messages list. But there's one little problem now. We only have a one way to render a conversation window. It gets rendered only when a conversation is created. We'll add additional ways to render conversations' windows in just a moment. But before that, let's recap how data reaches channel's subscribers.

1. After a new `Private::Message` record is created, the `after_create_commit` method gets triggered, which calls the background job
2. `Private::MessageBroadcastJob` processes given data and broadcasts it to channel's subscribers
3. On the client side the `received` callback function is called, which appends data to the DOM

Commit the changes.

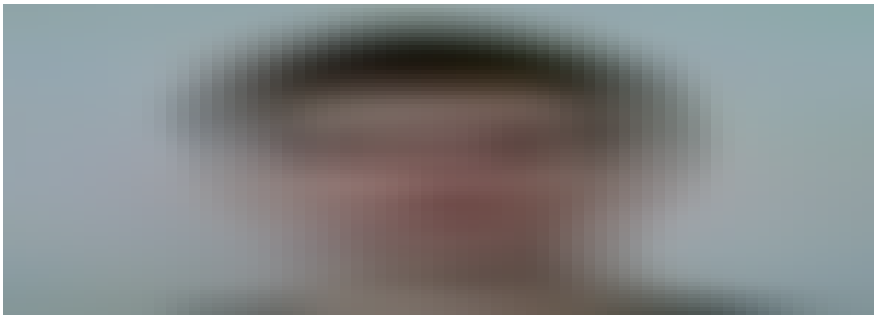
```
git add -A
git commit -m "Broadcast a new message
```

```
- Inside the Private::Message define an after_create_comit
callback method.
- Create a Private::MessageBroadcastJob
- Define a received function inside the
  assets/javascripts/channels/private/conversation.js
- Create a conversation.js inside the
  assets/javascripts/channels/shared"
```

Navigation bar update

On the navigation bar we're going to render a list of user's conversations. When a list of conversations is opened, we want to see conversations ordered by the latest messages. Conversations with the most recent messages are going to be at the top of the list. This list should be accessible throughout the whole application. So inside the `ApplicationController`, store ordered user's conversations inside an

instance variable. The way I suggest doing that is define an `all_ordered_conversations` method inside the controller

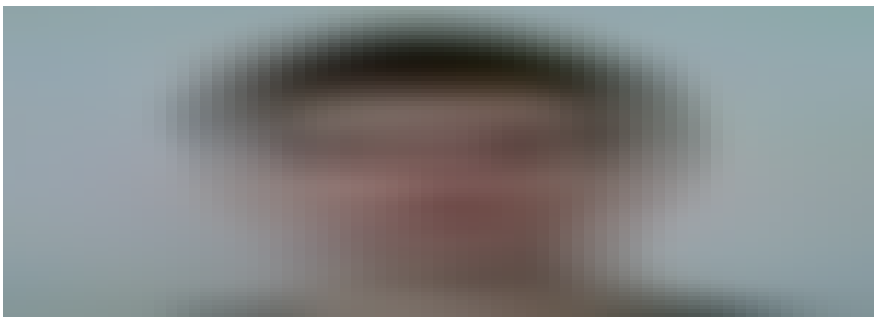


controllers/application_controller.rb

Add a `before_action` filter, so the `@all_conversations` instance variable is available everywhere.

```
before_action :all_ordered_conversations
```

And then create an `OrderConversationsService` to take care of conversations' querying and ordering.

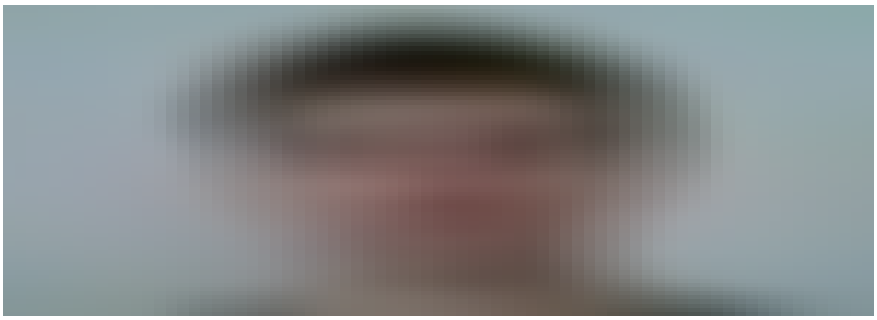


services/order_conversations_service.rb

Currently this service only deals with private conversations, that's the only type of conversations we've developed so far. In the future we'll mash private and group conversations together, and sort them by their latest messages. The `sort` method is used to sort an array of conversations. Again, if we didn't use the `includes` method, we would experience a N + 1 query problem. Because when we sort conversations, we check the latest messages' creation dates of every conversation and compare them. That's why with the query we have included messages' records.

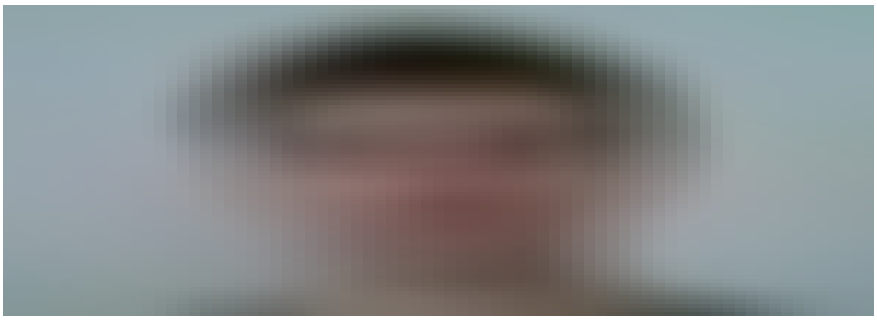
The `<=>` operator evaluates which `created_at` value is higher. If we used
`a <=> b`, it would sort a given array in ascending order. When you evaluate values in the opposite way, `b <=> a`, it sorts an array in descending order.

We haven't defined the `all_by_user` scope inside the `Private::Conversation` model yet. Open the model and define the scope:

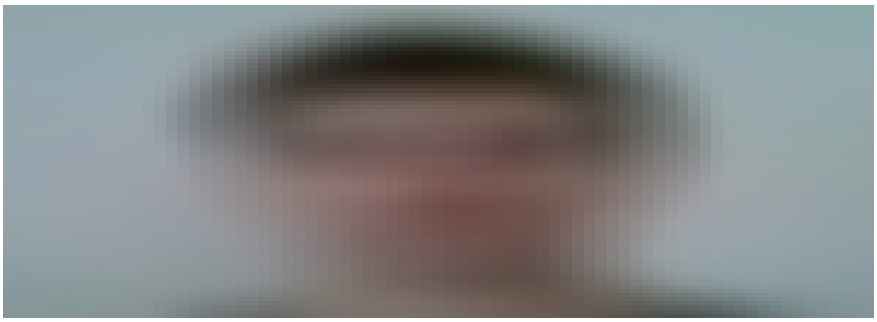


models/private/conversation.rb

Write specs for the service and the scope:



spec/models/private/conversation_spec.rb



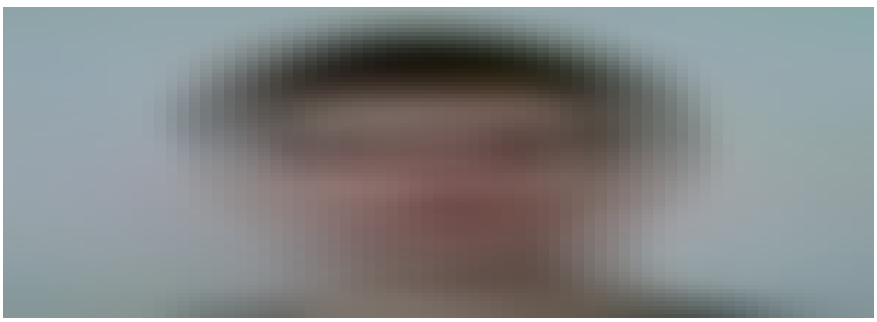
spec/services/order_conversations_service_spec.rb

Commit the changes.

```
git add -A
git commit -m "
- Create an OrderConversationsService and add specs for it
- Define an all_by_user scope inside the
Private::Conversation
  model and add specs for it"
```

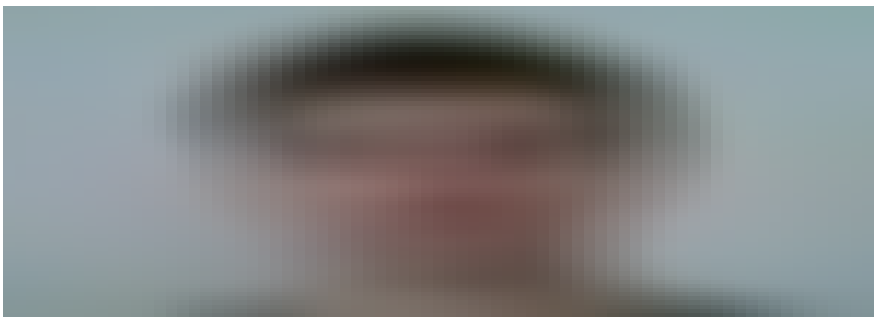
Now inside views, we have an access to an array of ordered conversations. Let's render a list of their links. Whenever a user clicks on any of them, a conversation window gets rendered on the app. If you recall, our navigation bar has two major components. Inside one component, elements are displayed constantly. Within another component, elements collapse on smaller devices. So inside the navigation's header, where components are visible all the time, we're going to create a drop down menu of conversations. As usually, to prevent having a large view file, split it into multiple smaller ones.

Open the navigation's `_header.html.erb` file and replace its content with the following:



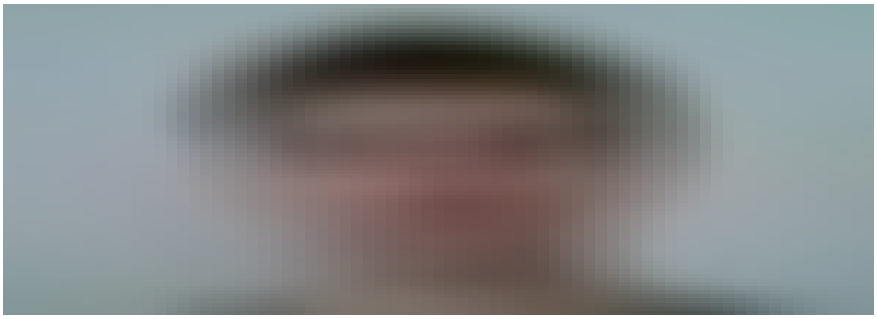
layouts/navigation/_header.html.erb

Now create a `header` directory with a `_toggle_button.html.erb` file inside



layouts/navigation/header/_toggle_button.html.erb

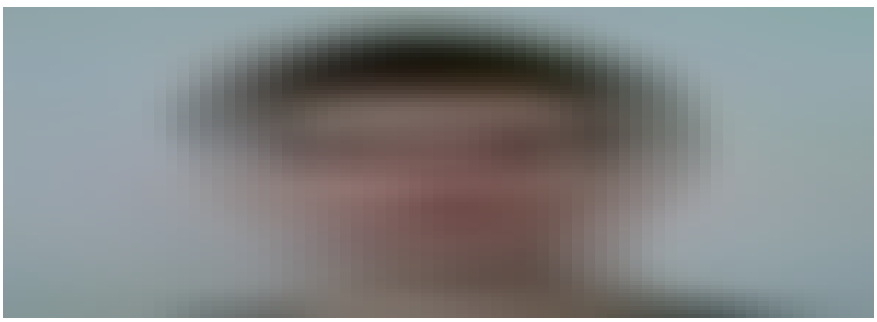
This is a toggle button which was formerly located inside the `_header.html.erb` file. Create another file inside the `header` directory



layouts/navigation/header/_home_button.html.erb

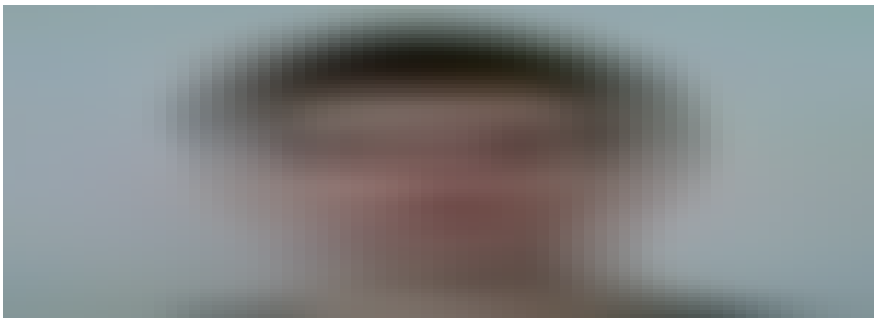
And this is the home button from the `_header.html.erb` . Also there is an extra link here. On smaller devices we're going to display an icon, instead of the name of the application.

Look back at the `_header.html.erb` file. There is a helper method `nav_header_content_partials` , which returns an array of partials' paths. The reason why we don't just render partials one by one is because the array is going to differ in different cases. Inside the `NavigationHelper` define the method



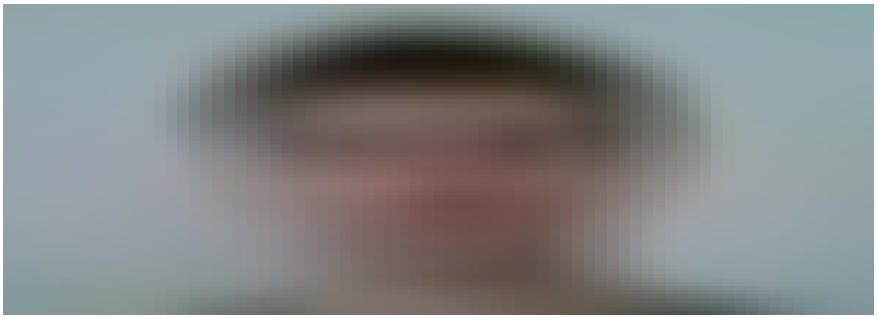
helpers/navigation_helper.rb

Write specs for the methods inside the `navigation_helper_spec.rb`



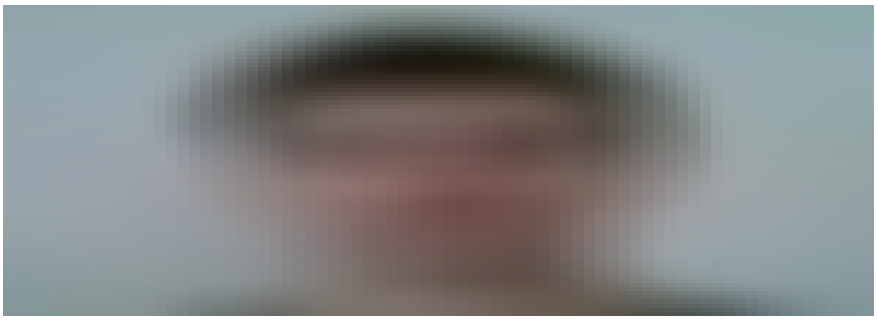
spec/helpers/navigation_helper_spec.rb

Now create necessary files to display drop down menus on the navigation bar. Start by creating a `_dropdowns.html.erb` file



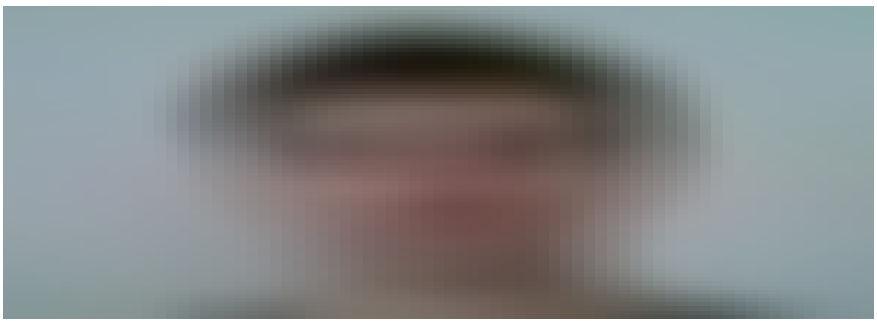
layouts/navigation/header/_dropdowns.html.erb

Create a `dropdowns` directory with a `_conversations.html.erb` file inside



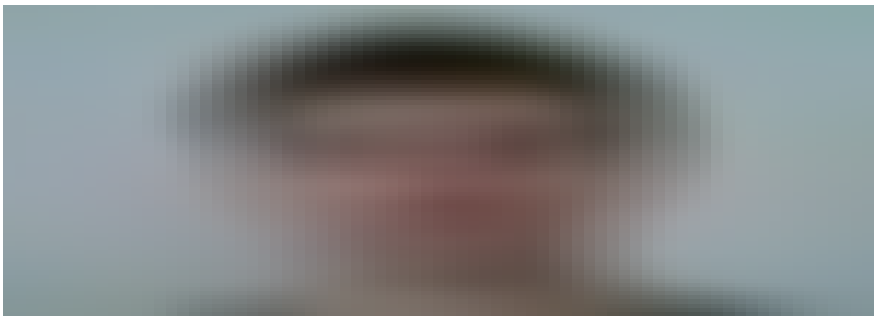
layouts/navigation/header/dropdowns/_conversation.html.erb

This where we use the `@all_conversations` instance variable, defined inside the controller before, and render links to open conversations. Links for different type of conversations are going to differ. We'll need to create two different versions of links for private and group conversations. First define the `conversation_header_partial_path` helper method inside the `NavigationHelper`



helpers/navigation_helper.rb

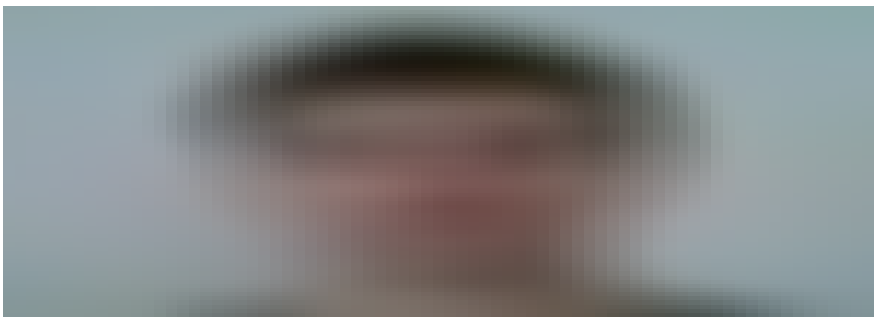
Write specs for it:



spec/helpers/navigation_helper.rb

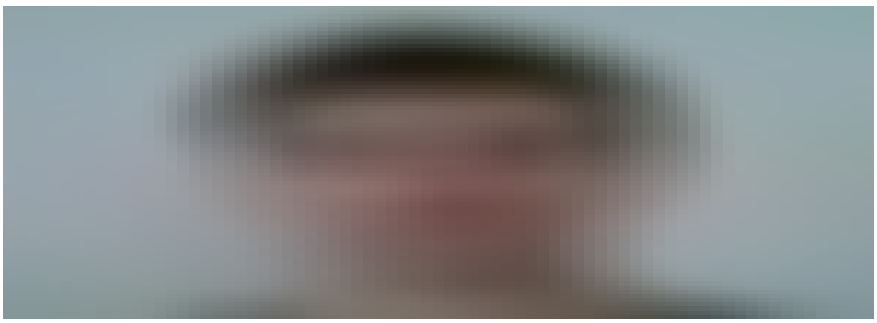
Of course we haven't done anything with group conversations yet. So you have to comment out the group conversation's part in specs for a while to avoid failure.

Create a file for private conversations' links:



layouts/navigation/header/dropdowns/conversations/_private.html.erb

Define the `private_conv_seen_status` helper method inside a new `Shared::ConversationsHelper`

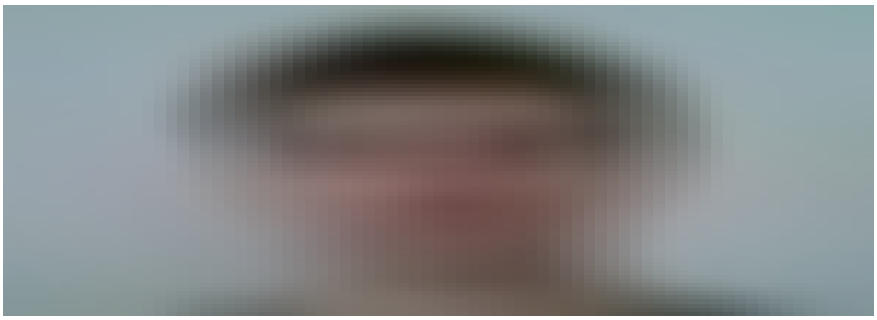


helpers/shared/conversations_helper.rb

Add this this module to the `Private::ConversationsHelper`

```
  include Shared::ConversationsHelper
```

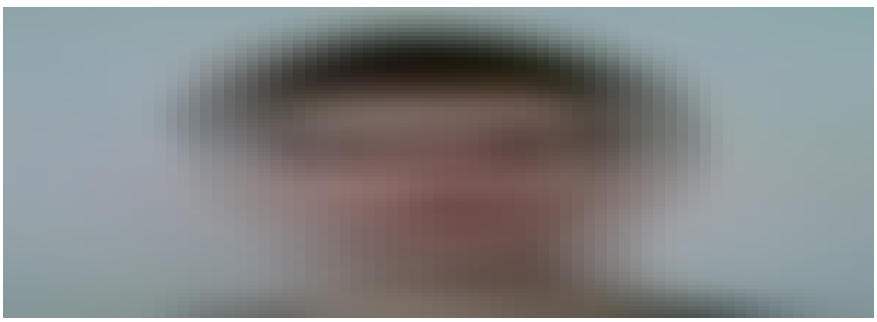
Inside specs create a `shared` directory with a `conversations_helper_spec.rb` file to test the `private_conv_seen_status` helper method.



spec/helpers/shared/conversations_helper_spec.rb

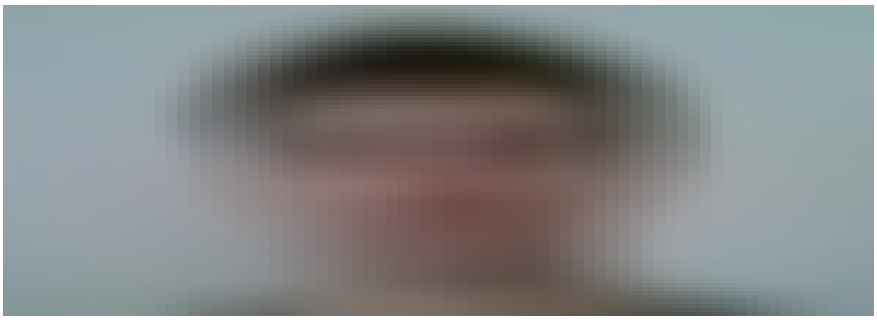
When a link to a conversation is clicked, the `Private::Conversation` controller's `open` action gets called. Define a route to this action. Inside the `routes.rb` file, add a `post :open` member inside the namespaced `privateconversations` resources, just below the `post :close` .

Of course don't forget to define the action itself inside the controller:



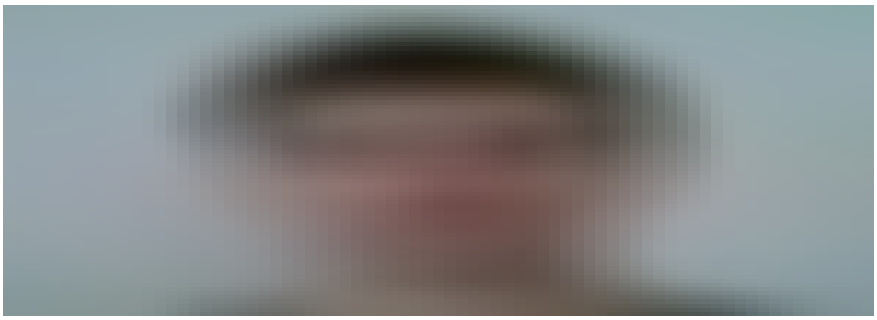
conversations_controller.rb

Now a conversation window should open when you click on its link. The navigation bar right now is messy, we have to take care of its design. To style the drop down menus, add CSS to the `navigation.scss` file.



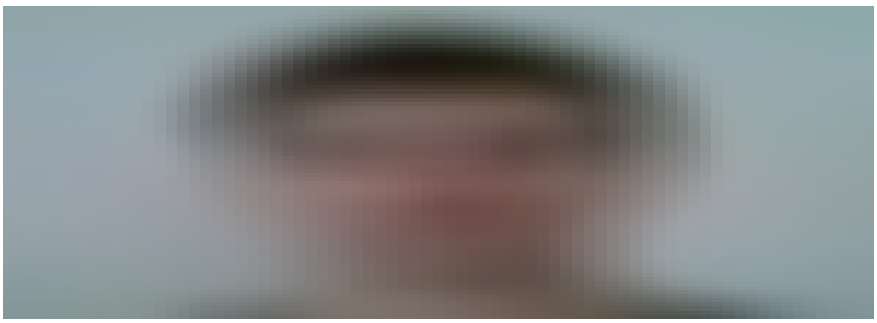
assets/stylesheets/partials/layout/navigation.scss

Update the `max-width: 767px` media query inside the `mobile.scss`



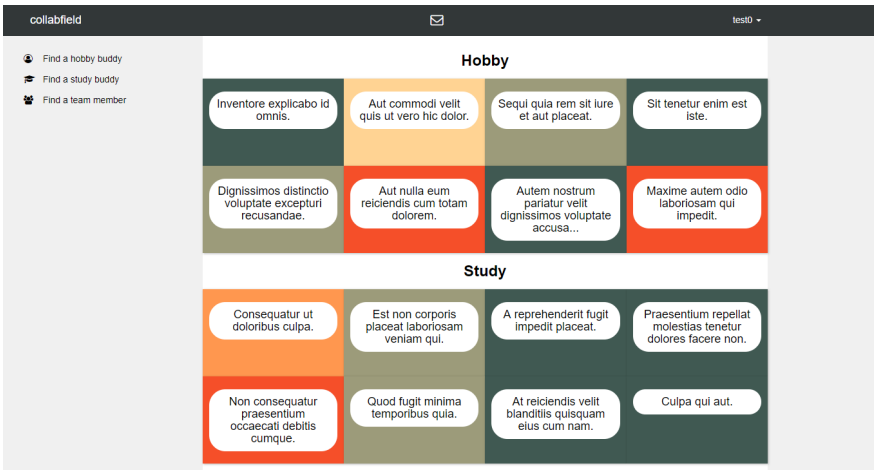
assets/stylesheets/responsive/mobile.scss

Update the `min-width: 767px` media query in `desktop.scss`

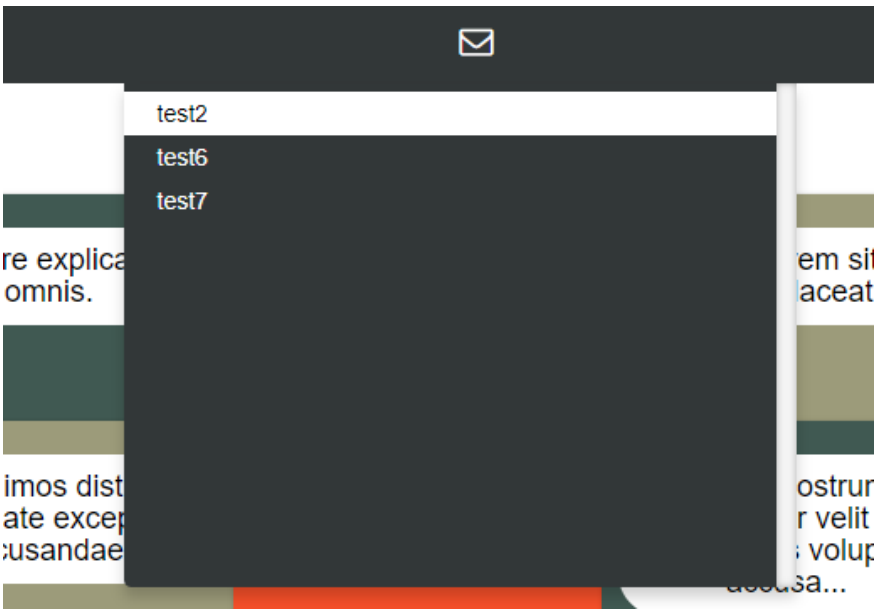


assets/stylesheets/responsive/desktop.scss

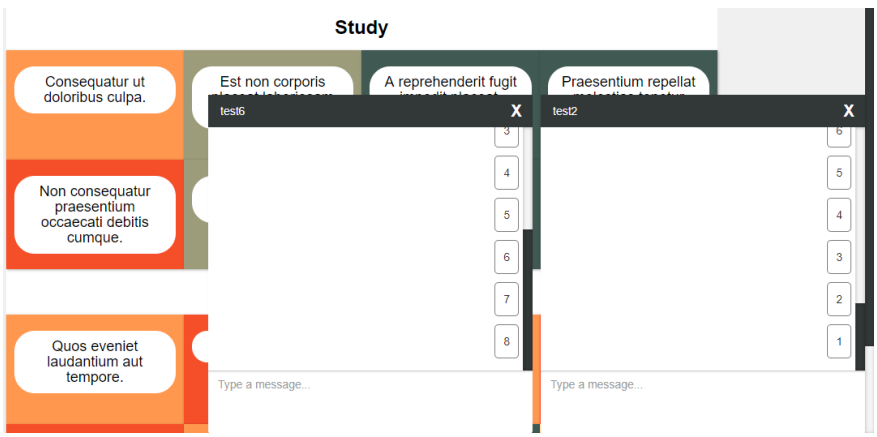
The app looks like this now



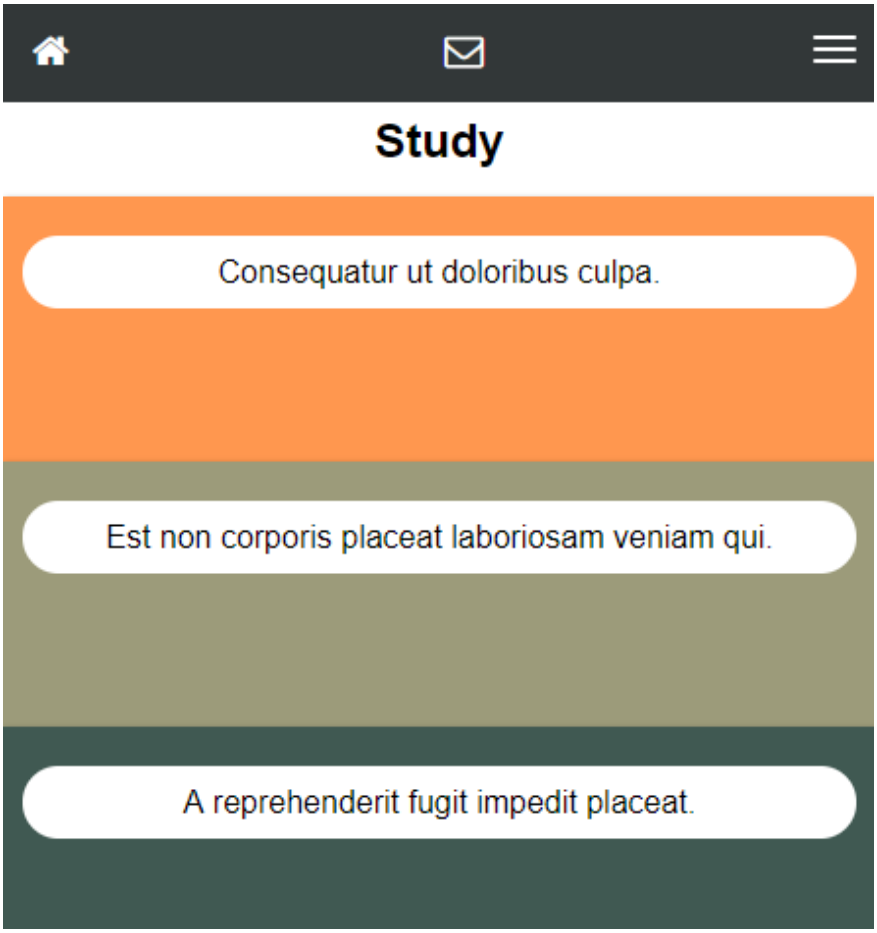
Then you can expand the conversations list



By clicking on any of the menu links, a conversation window should appear on the app



If you try to contract the browser's size, conversations should be hidden one by one



Also notice that instead of the collabfield logo we have the home page icon now. And the conversations list is still available on smaller screens. Well, if conversations' windows are hidden on smaller devices, how users are going to communicate on mobile devices? We'll create a messenger which will be opened instead of a conversation window.

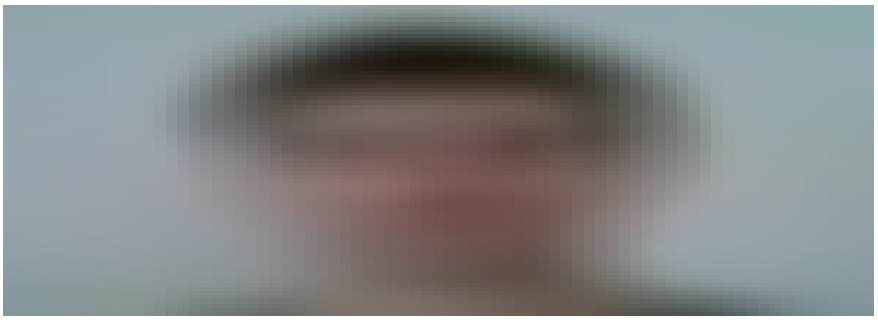
Commit the changes

```
git add -A
git commit -m "Render a drop down menu of conversations
links

- split layouts/navigation/_header.html.erb file's content
into partials
- Create a _toggle_button.erb.html inside
layouts/navigation/header
- Create a _home_button.html.erb inside
layouts/navigation/header
- Define a nav_header_content_partials inside
NavigationHelper
```

```
  and write specs for it
- Create a _dropdowns.html.erb inside
layouts/navigation/header
- Create a _conversation.html.erb inside
  layouts/navigation/header/dropdowns
- Define a conversation_header_partial_path inside
NavigationHelper
  and write specs for it
- Create a  _private.html.erb inside
  layouts/navigation/header/dropdowns/conversations
- Define a private_conv_seen_status inside
  Shared::ConversationsHelper and write specs for it
- Define an open action inside the Private::Conversations
controller
- add CSS to style drop down menus on the navigation bar.
  Inside navigation.scss, mobile.scss and desktop.scss"
```

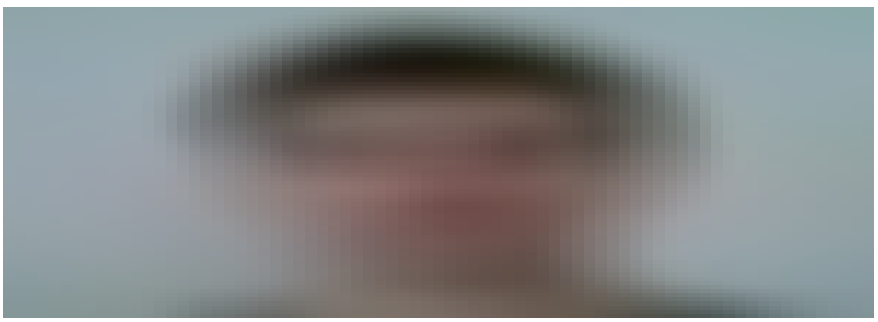
It's a good time to make sure that all features of the real time messaging works correctly.

Because we're adding elements to the DOM dynamically, sometimes elements are added too late and Capybara thinks that an element doesn't exist, because the wait time by default is 2 seconds only. To avoid these failures, inside the `rails_helper.rb` , change wait time somewhere between 5 to 10 seconds.



spec/rails_helper.rb

Inside the `spec/features/private/conversations folder` create a `window_spec.rb` file.



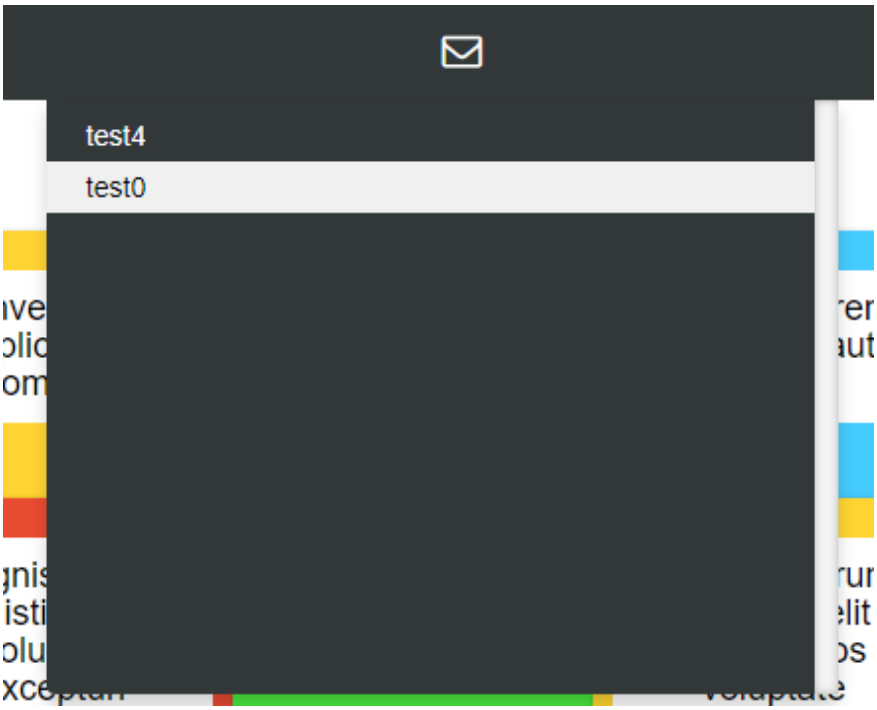
spec/features/private/conversations/window_spec.rb

Here I haven't defined specs, to test if a recipient user receives messages in real time. Try to figure out how to write such tests on your own.

Commit the changes

```
git add -A
git commit -m "Add specs to test the conversation window's
functionality"
```

If you logged in an account which has received messages, you would notice a conversation, marked as an unseen



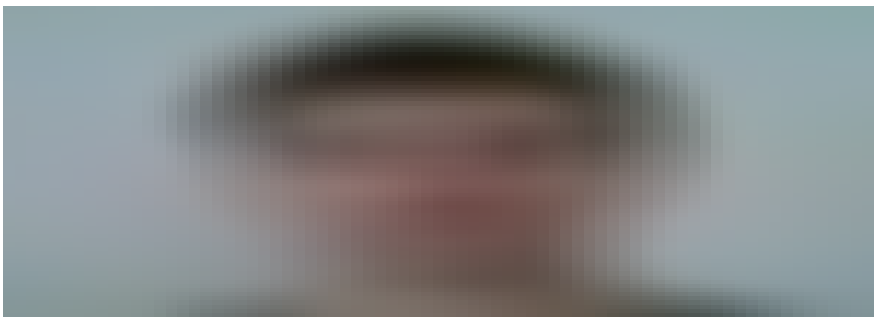
At this moment there is no way to mark conversations as seen. By default a new message has an unseen value. Program the app in a way that when a conversation window is opened or clicked, its messages get marked as seen. Also note that currently we only see highlighted unseen conversations when the drop down menu is expanded. In the future we will create a notifications feature, so users will know that they got new messages without expanding anything.

Let's tackle the first problem. When a conversation is already rendered on the app, but it is collapsed, and a user clicks on the drop down menu's link to open that conversation, nothing happens. That collapsed conversation stays collapsed. We've to add some JavaScript, so in the

case of the drop down menu's link click, the conversation should expand and focus its new message area.

Open the file below and add the code from the Gist to achieve that.

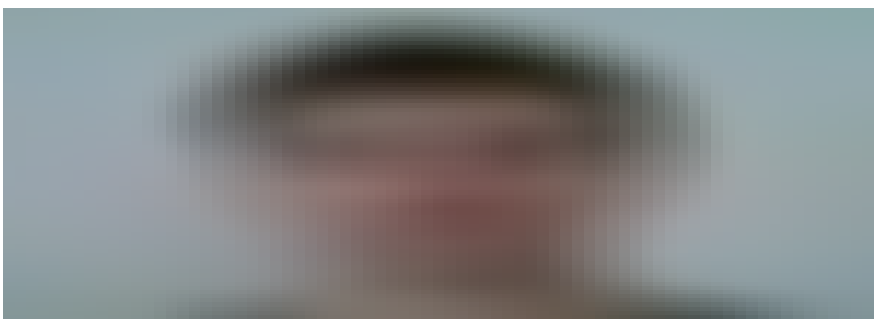
```
assets/javascripts/conversations/toggle_window.js
```



assets/javascripts/conversations/toggle_window.js

When you click on a link, to open a conversation window, no matter if a conversation is already present on the app, or not, it will be expanded.

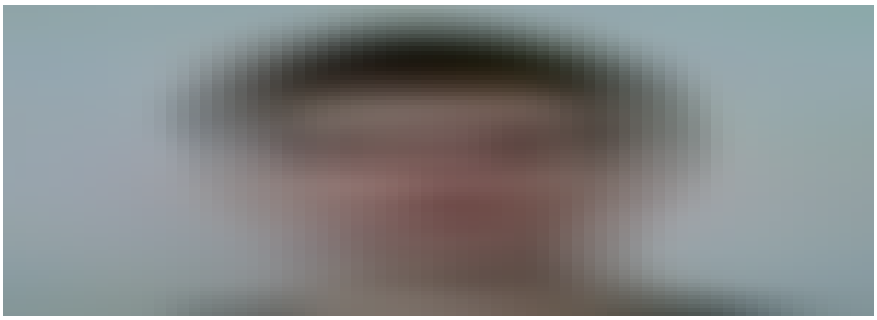
Now we need an event handler. After a conversation window which has unseen messages is clicked, the private conversation client's side should fire a callback function. First, define an event handler inside the private conversation client's side, at the bottom of the file



assets/javascripts/channels/private/conversation.js

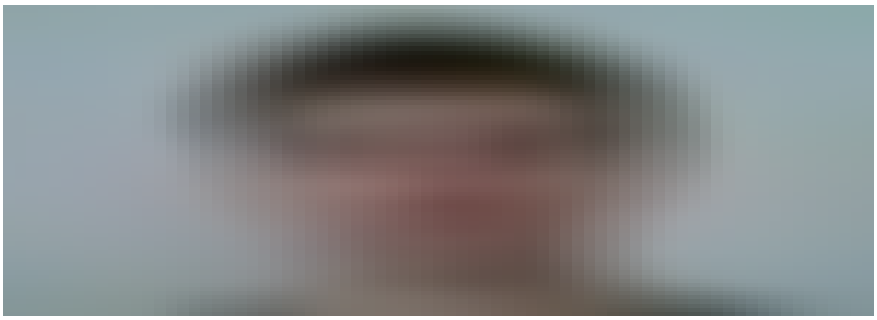
A case of messenger's existence is already included in this snippet of code.

Then, define the callback function inside the `private_conversation` instance, just below the `send_message` function



assets/javascripts/channels/private/conversation.js

Finally, define this method on the server side



channels/private/conversation_channel.rb

After a user clicks on a link to open a conversation window or clicks directly on a conversation window, its unseen messages will be marked as seen.

Commit the changes

```
git add -A
git commit -m "Add ability to mark unseen messages as seen
```

```
- Add an event handler to expand conversation windows inside
the
  assets/javascripts/conversations/toggle_window.js
- Add an event handler to mark unseen messages as seen
inside the
  assets/javascripts/channels/private/conversation.js
- Define a set_as_seen method for
Private::ConversationChannel"
```

Make sure that everything works as we expect by writing the specs.

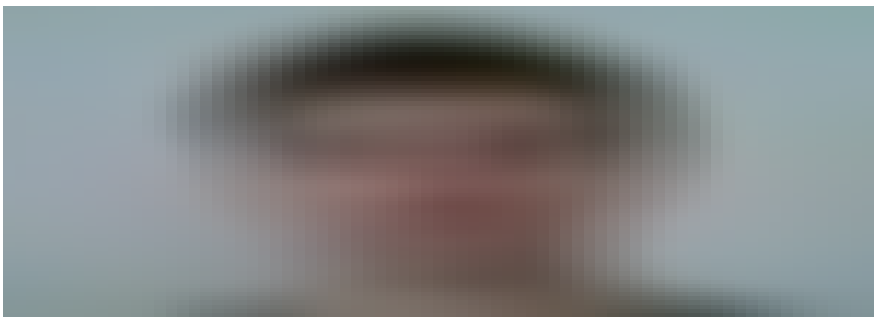
Contacts

To stay in touch with people, you met on the app, you have to be able to add them to contacts. We're missing this functionality right now. Also having a contacts feature opens a lot of possibilities to create other features that only users who are accepted as a contact could perform.

Generate a `Contact` model

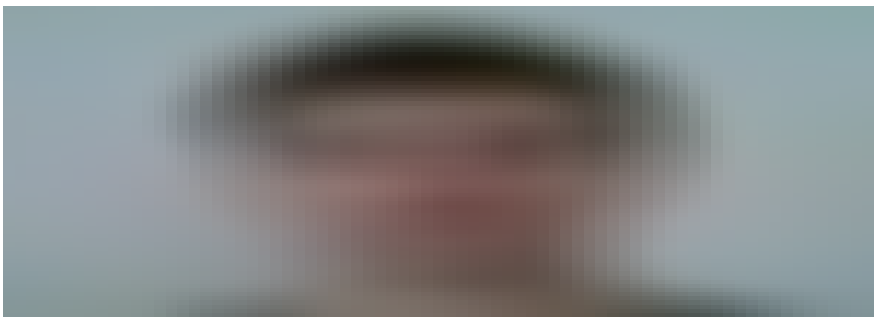
```
rails g model contact
```

Define associations, validation and a method to find a contact record by providing users' ids.



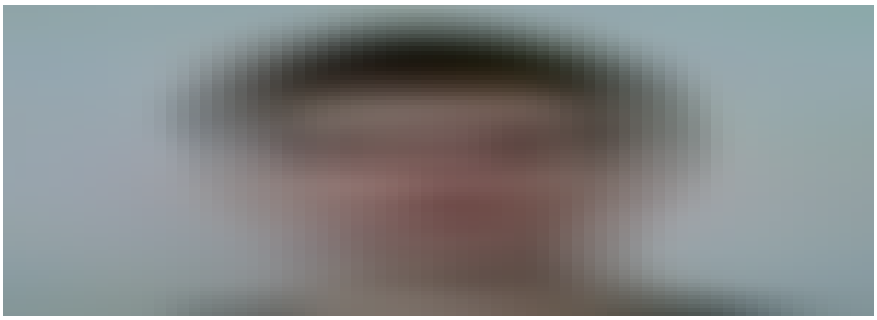
models/contact.rb

Define the contacts table



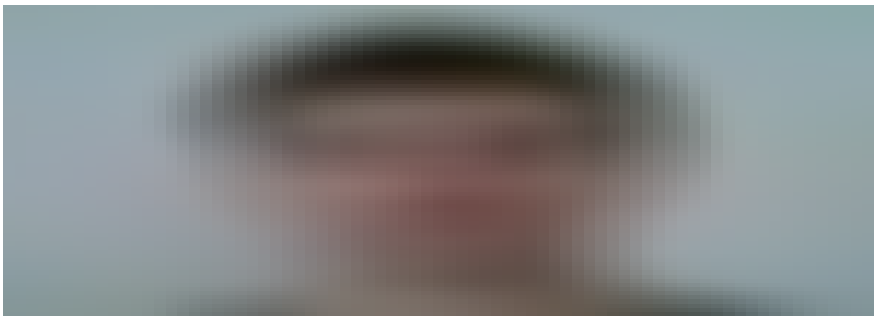
db/migrate/CREATION_DATE_create_contacts.rb

A factory for contacts is going to be needed. Define it:



spec/factories/contacts.rb

Write specs to test the model

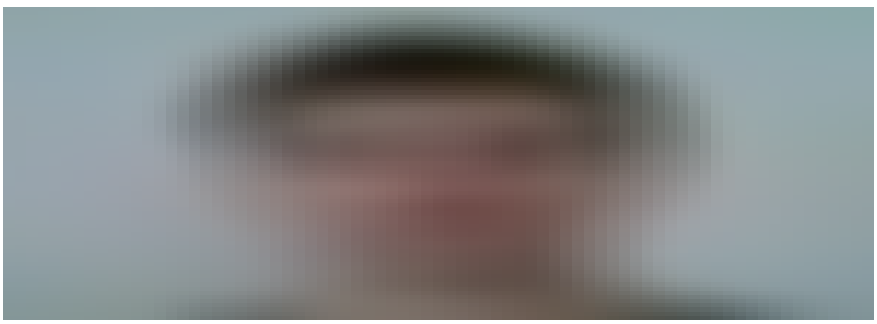


spec/models/contact_spec.rb

Commit the changes

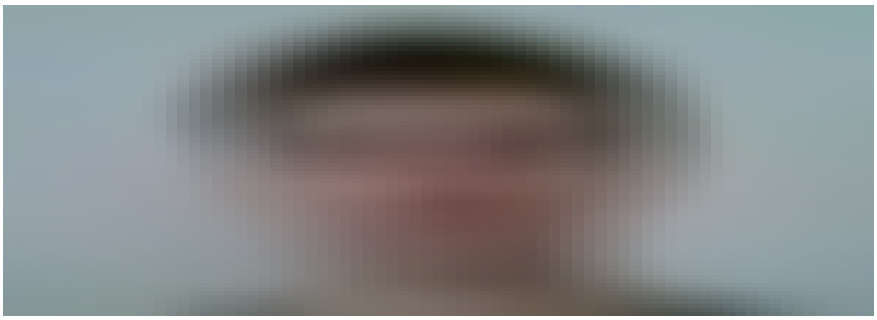
```
git add -A
git commit -m "Create a Contact model and write specs for
it"
```

Inside the `User` model's file, we have to define appropriate associations and also define some methods to help with contacts' queries.



models/user.rb

Cover associations and methods with specs



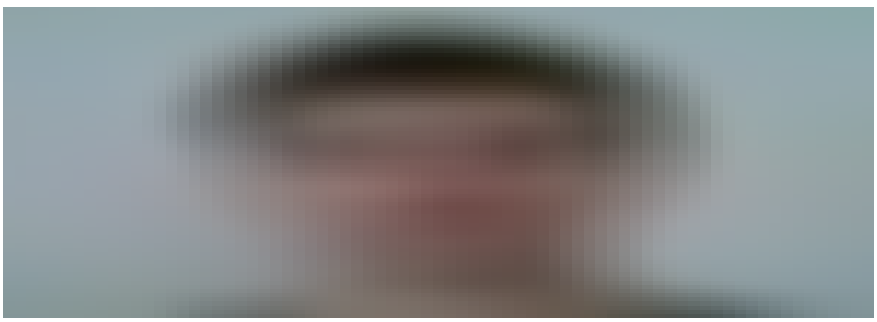
spec/models/user_spec.rb

Commit the changes

```
git add -A
git commit -m "Add associations and helper methods to the
User model

- Create relationship between the User the Contact models
- Methods help query the Contact records"
```

Generate a `Contacts` controller and define its actions

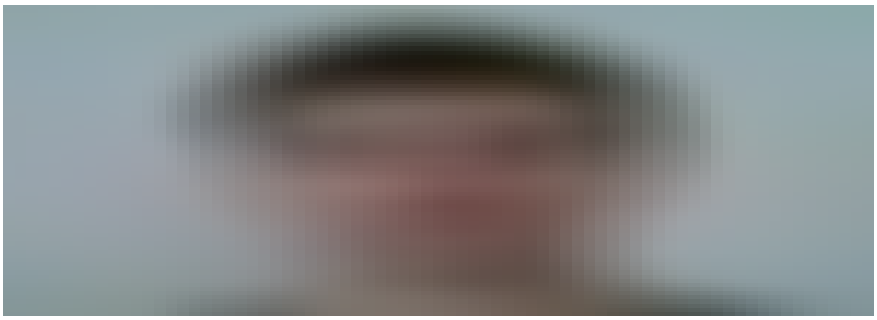
```
rails g controller contacts
```



controllers/contacts_controller.rb

As you see, users will be able to create a new contact record, update its status (accept a user to their contacts) and remove a user from their contact list. Because all actions are called via AJAX and we don't want to render any templates as a response, we respond with a success response. This way Rails doesn't have to think what to respond with.

Define the corresponding routes:



routes.rb

Commit the changes.

```
git add -A
git commit -m "Create a ContactsController and define routes
to its actions"
```

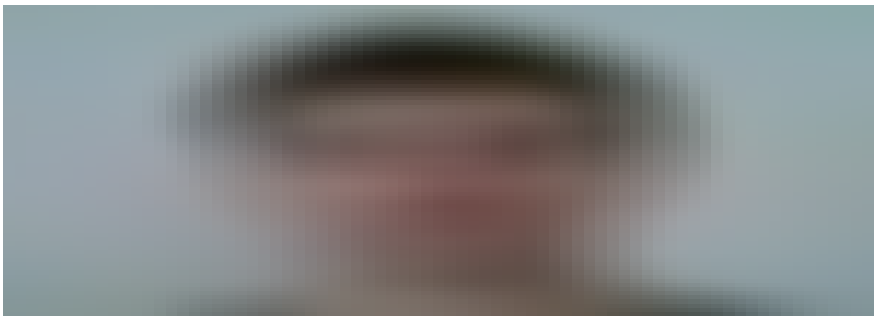
Private conversation's window update

The way users are going to be able to send and accept contact requests is through a private conversation's window. Later we'll add an extra way to accept requests through a navigation bar's drop down menu.

Create a new `heading` directory

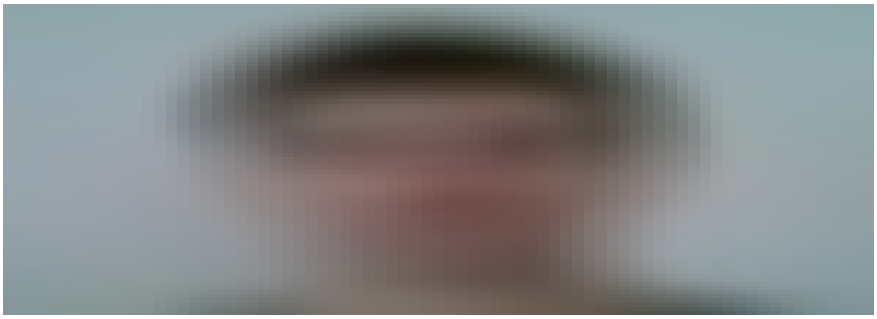
```
private/conversations/conversation/heading
```

This is where we'll keep extra options for a private conversation's window. Inside the directory, create a `_add_user_to_contacts.html.erb` file



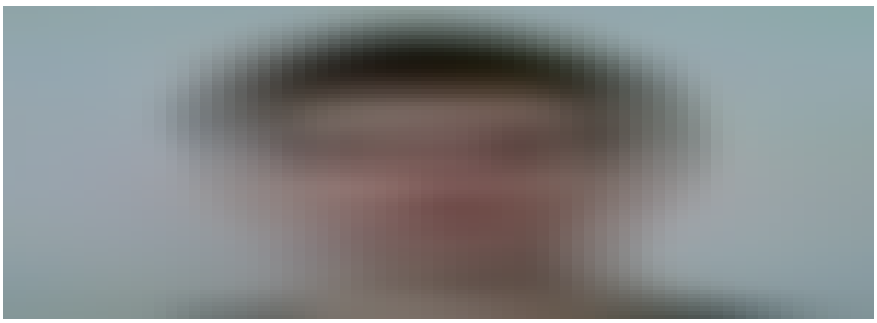
private/conversations/conversation/heading/_add_user_to_contacts.html.erb

At the bottom of the `_heading.html.erb` file, render the option to add the conversation's opposite user to contacts:



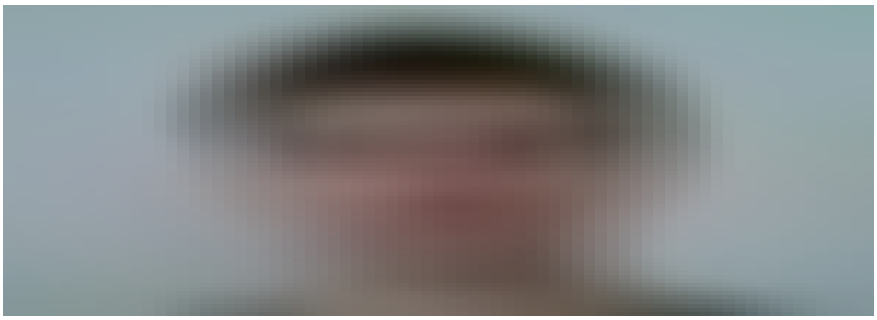
private/conversations/conversation/_heading.html.erb

Define the helper method and additional methods within a private scope



helpers/private/conversations_helper.rb

Write specs for these helper methods

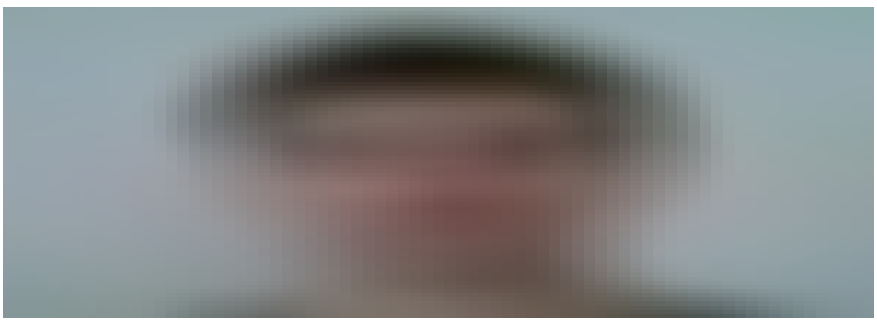


spec/helpers/private/conversations_helper_spec.rb

The `instance_eval` method is used to test methods within a private scope.

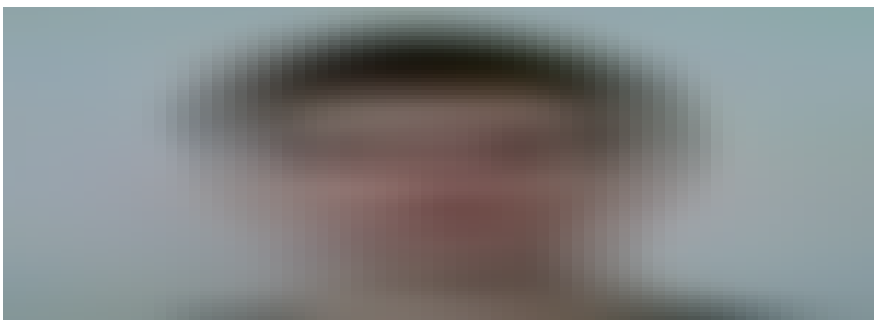
Because we're going to display options on the conversation window's heading element, we have to make sure that additional options fit perfectly on the heading. Inside the `_heading.html.erb` file, replace the `conversation-heading` class with `<%= conv_heading_class(@contact) %>`, to determine which class to add.

Define the helper method



helpers/private/conversations_helper.rb

Write specs for the method

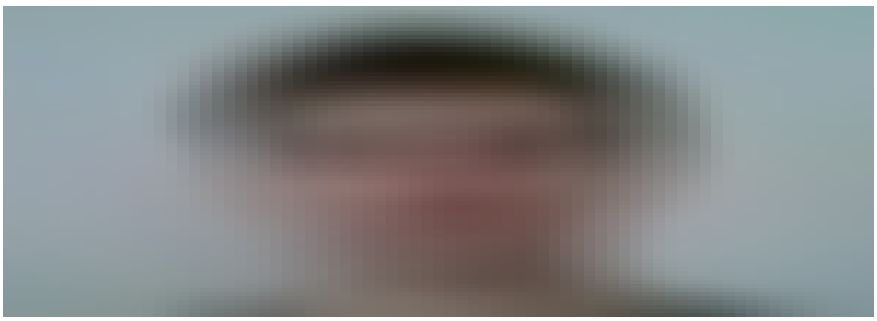


spec/helpers/private/conversations_helper_spec.rb

The options, to send or accept a contact request, won't be shown yet. More elements need to be added. Open the `_conversation.html.erb` file

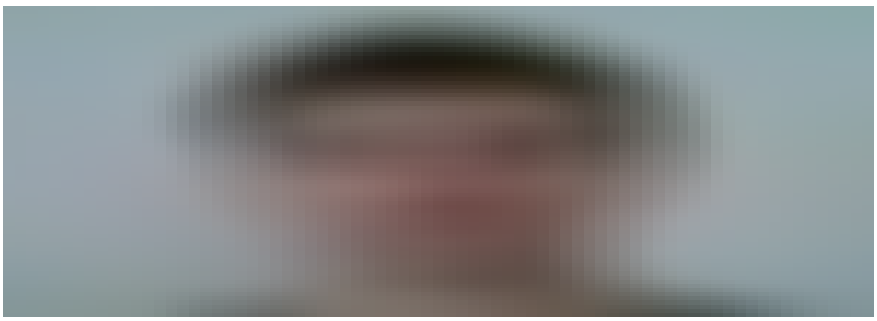
```
private/conversations/_conversation.html.erb
```

At the top of the file, define a `@contact` instane variable, so it is accessible across all partials



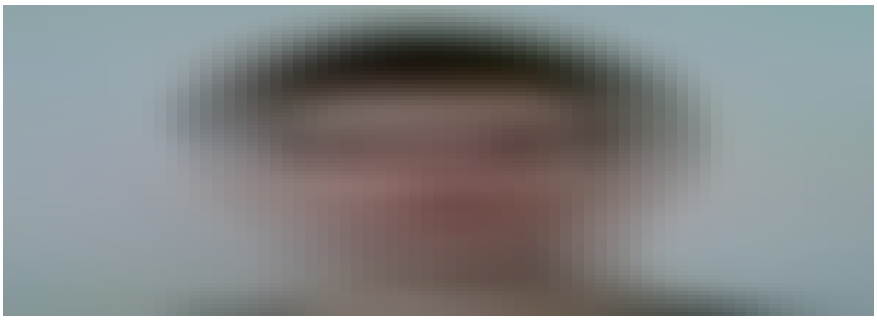
private/conversations/_conversation.html.erb

Define the `get_contact_record` helper method



helpers/private/conversations_helper.rb

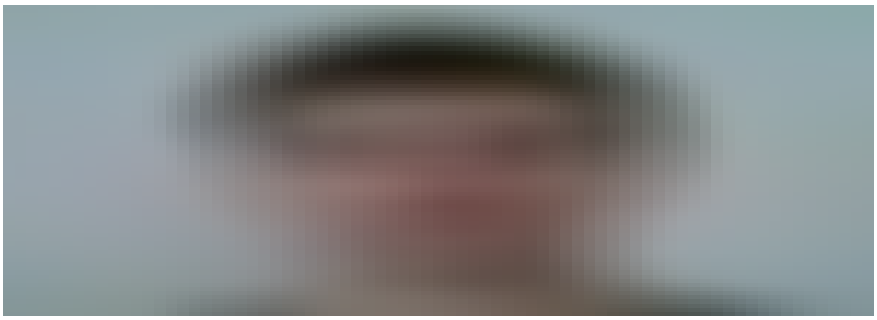
Cover the method with specs



spec/helpers/private/conversations_helper_spec.rb

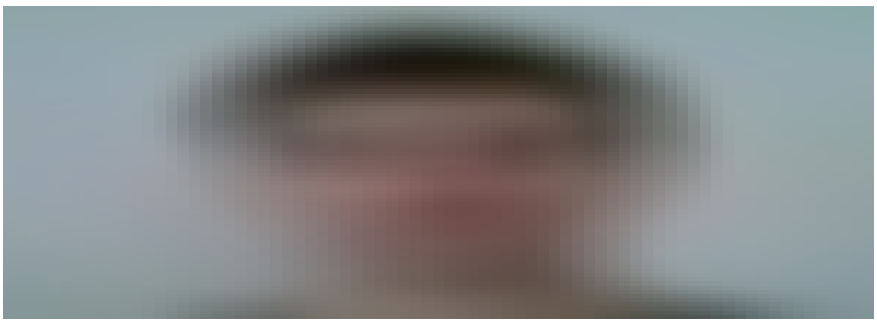
Previously, we used `current_user` and `recipient` `let` methods only within a private scope's context. Now we need access to them on both private and public methods. So cut and place them outside of private scope's context.

At the top of the `.panel-body` element, render a partial file which will show an extra message window to accept or decline a contact request



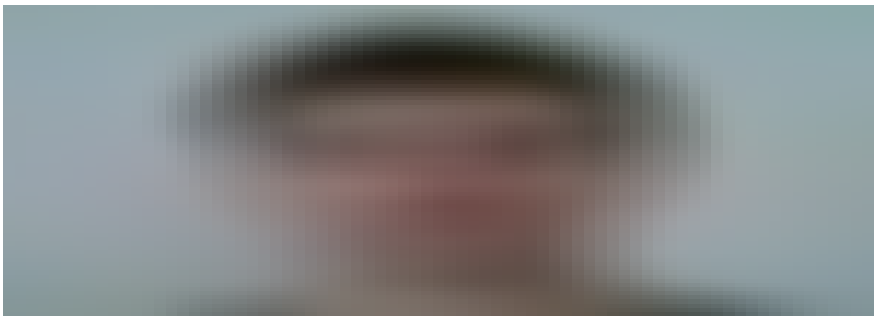
private/conversations/_conversation.html.erb

Create the `_request_status.html.erb` file



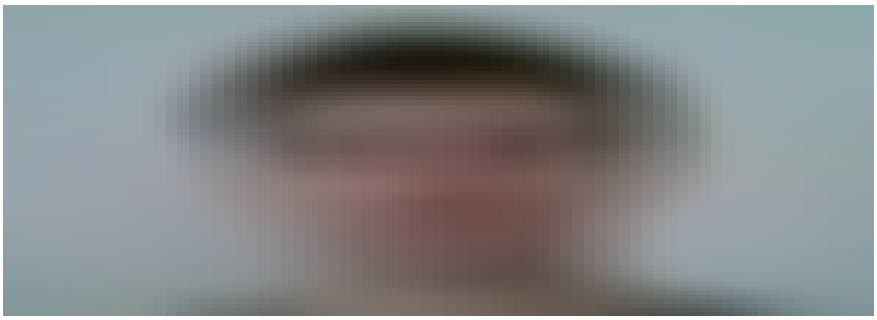
private/conversations/conversation/_request_status.html.erb

Define the needed helper methods



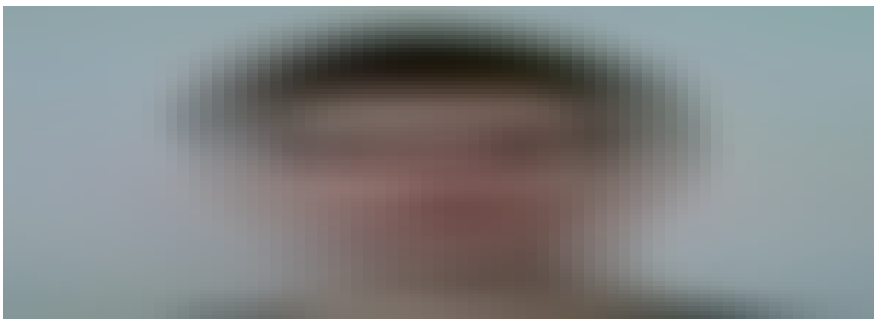
helpers/private/conversations_helper.rb

Writes specs for the helper methods

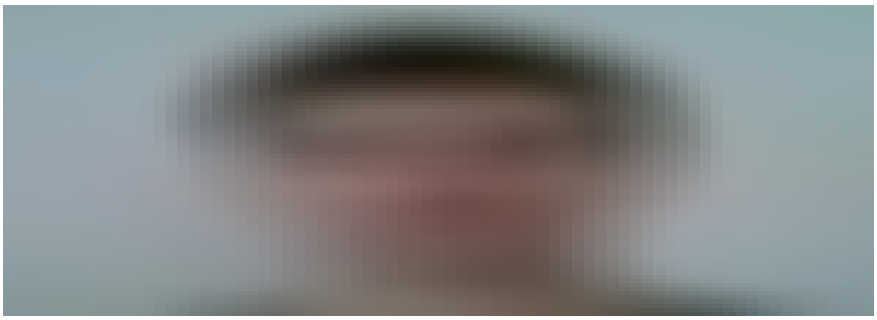


spec/helpers/private/conversations_helper_spec.rb

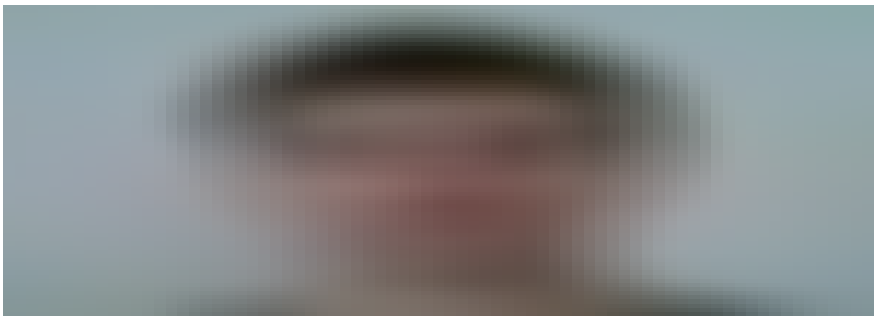
Create the `request_status` directory and then create `_send_request.html.erb`, `_sent_by_current_user.html.erb` and `_sent_by_recipient.html.erb` partial files



private/conversations/conversation/request_status/_send_request.html.erb



private/conversations/conversation/request_status/_sent_by_current_user.html.erb

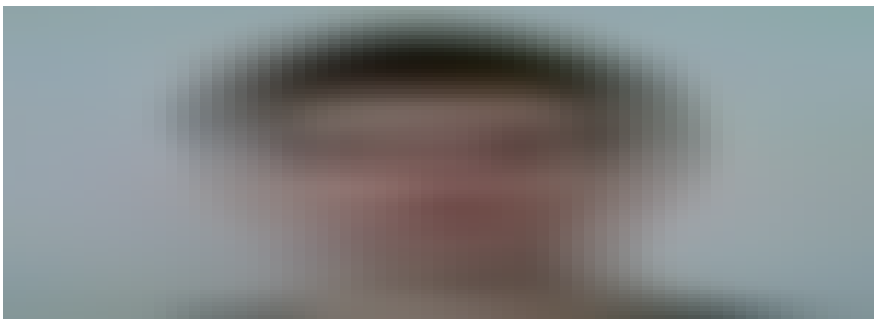


private/conversations/conversation/request_status/_sent_by_recipient.html.erb

Commit the changes

```
git add -A
git commit -m "Add a button on private conversation's window
to add a recipient to contacts"
```

Implement design changes and take care of styling issues which appear due to extra elements on the conversation window. Add CSS to the `conversation_window.scss` file

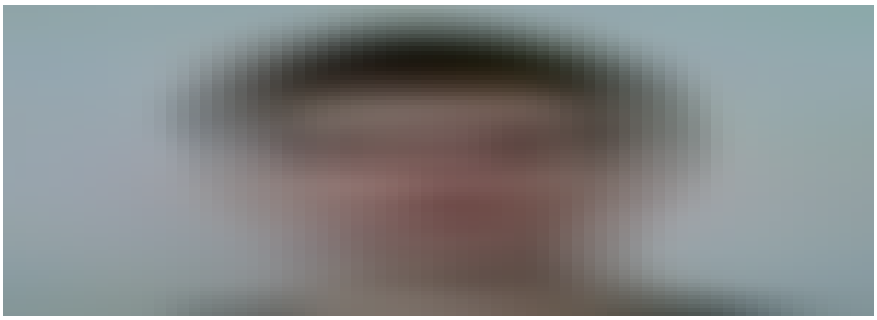


stylesheets/partials/conversation_window.scss

Commit the changes

```
git add -A
git commit -m "Add CSS to conversation_window.scss to style
option buttons"
```

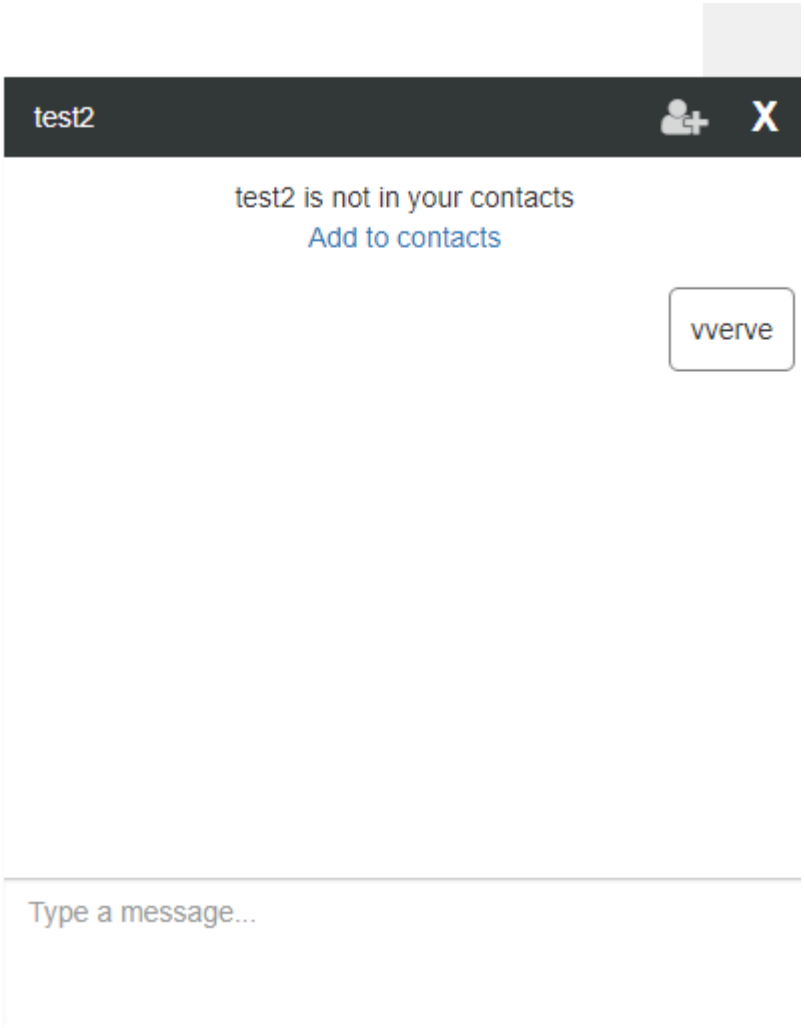
When a conversation window is collapsed, it would be better to not see any options. It's a more convenient design to see options only when a conversation window is expanded. To achieve it, inside the `toggle_window.js` file's toggle function, just below the `messages_visible` variable, add



assets/javascripts/conversations/toggle_window.js

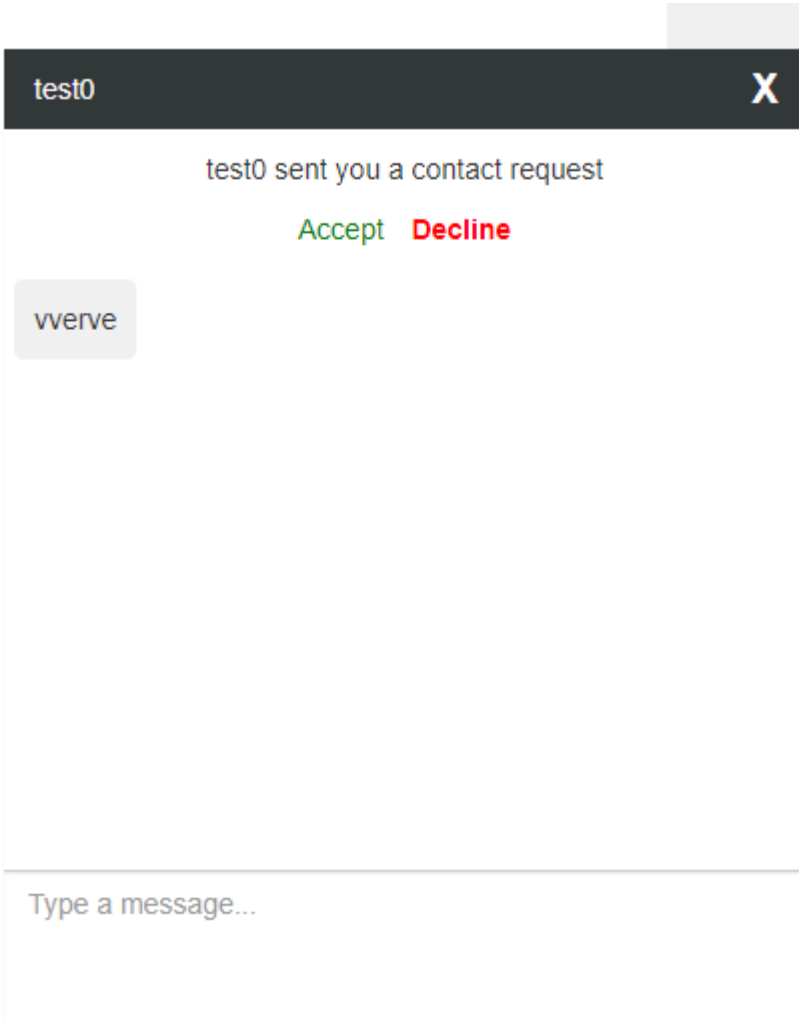
Now the collapsed window looks like this, it has no visible options

The expanded window has an option to add a user to contacts. Also there's a message which suggests to do that



Actually, you can send and accept a contact request right now by clicking an icon on the conversation's header or clicking the `Add to contacts` link. For now, there isn't any response after you click on those links and buttons. We'll add some feedback and real time notification system a little bit later. But technically, you can add users to your contacts, it is just not highly user friendly yet.

After you send a contact request, the opposite user's side looks like this



Commit the changes

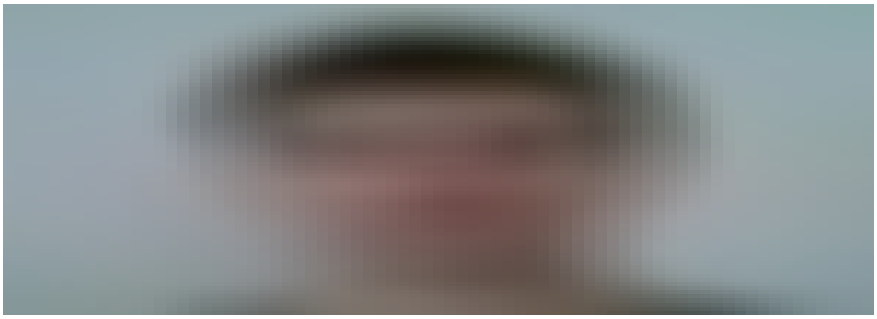
```
git add -A
git commit -m "Add JS inside the toggle_window.js to show
and hide additional options"
```

Currently users are able to talk privately, have one on one conversations. Since the app is about collaboration, it would be logical to have group conversations too.

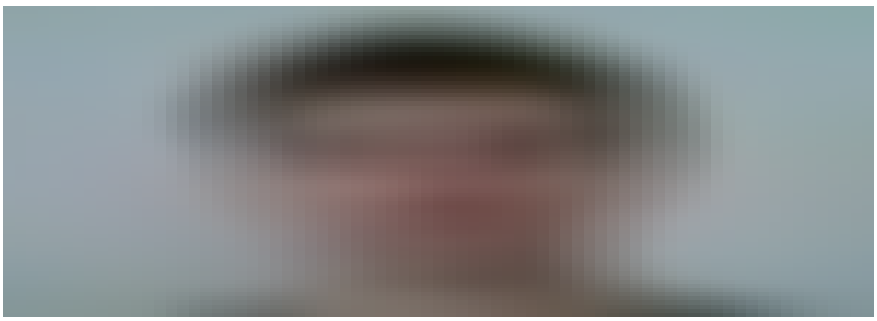
Start by generating a new model

```
rails g model group/conversation
```

Multiple users will be able to participate in one conversation. Define associations and the database table



models/group/conversation.rb

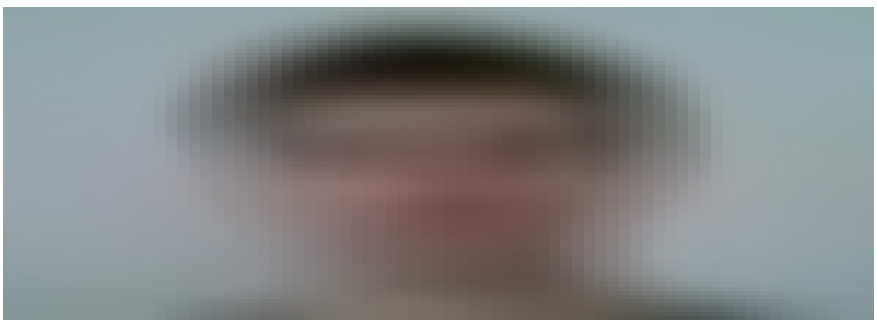


models/user.rb

A join table is going to be used to track who belongs to which group conversation

Then generate a model for messages

```
rails g model group/message
```

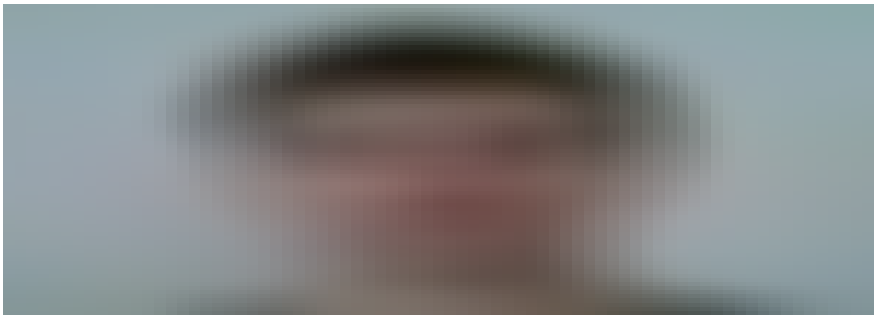


models/group/message.rb

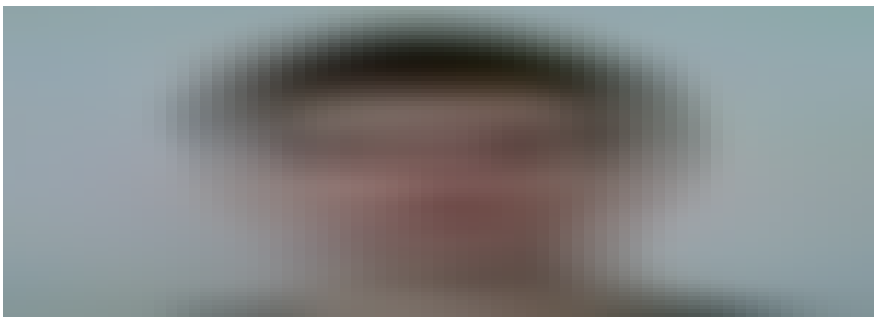
We'll store users' ids who have seen a message into an array. To create and manage objects, such as array, inside a database column, a serialize method is used. A default scope, to minimize the amount of queries, and some validations are added.

The way we're building group conversations is pretty similar to private conversations. In fact, styling and some parts are going to be in common between both types of conversations.

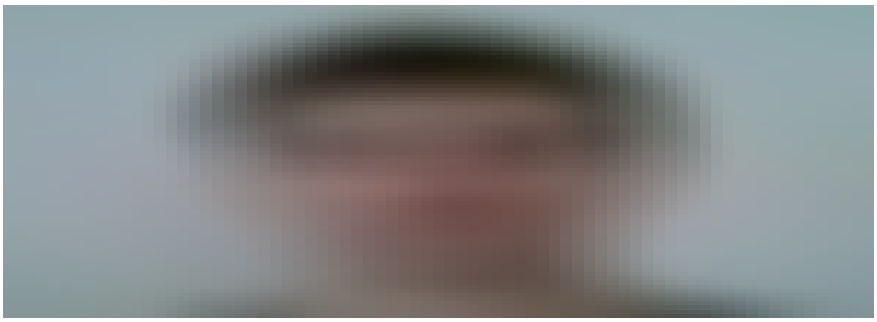
Write specs for the models. Also a factory for group messages is going to be needed



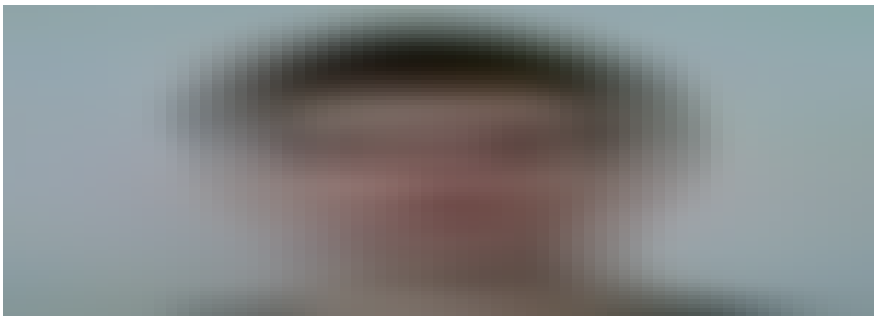
spec/factories/group_messages.rb



spec/models/user_spec.rb

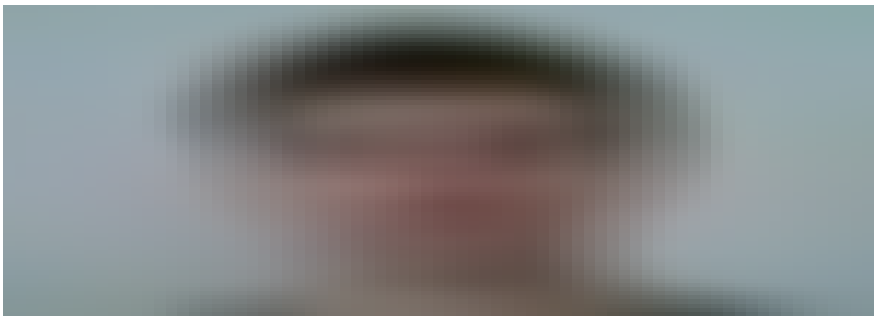


spec/models/group/conversation_spec.rb

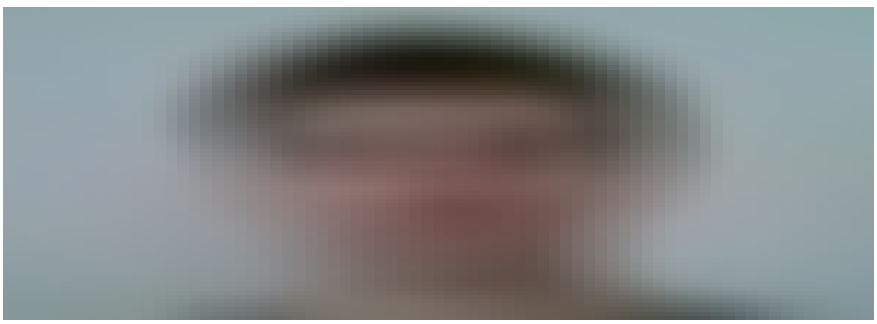


spec/models/group/message_spec.rb

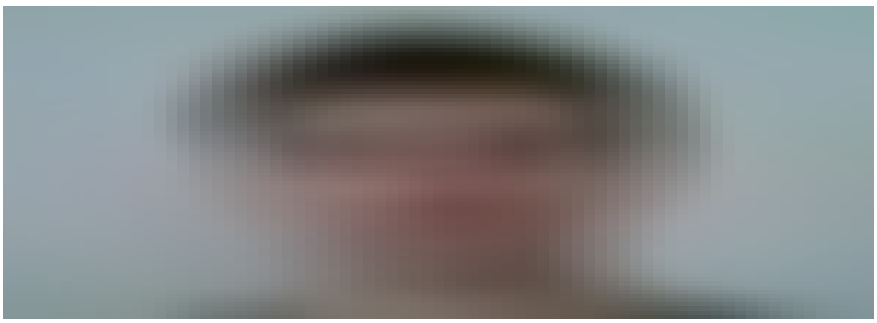
Define the migration files



db/migrate/CREATION_DATE_create_group_conversations.rb



db/migrate/CREATION_DATE_create_group_conversations_users_join_table.rb



db/migrate/CREATION_DATE_create_group_messages.rb

The fundamentals of the group conversation are set.

Commit the changes

```
git add -A
git commit -m "Create Group::Conversation and Group::Message
models

- Define associations
- Write specs"
```

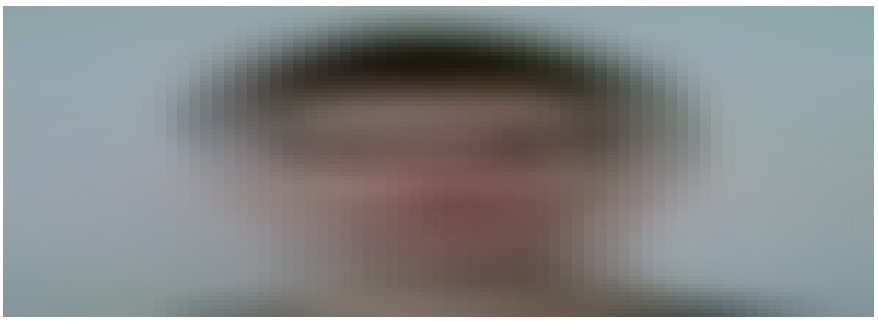
Create a group conversation

As mentioned before, the process of creating the group conversation feature is going to be similar to what we did with the private conversation. Start by creating a controller and a basic user interface.

Generate a namespaced controller

```
rails g controller group/conversations
```

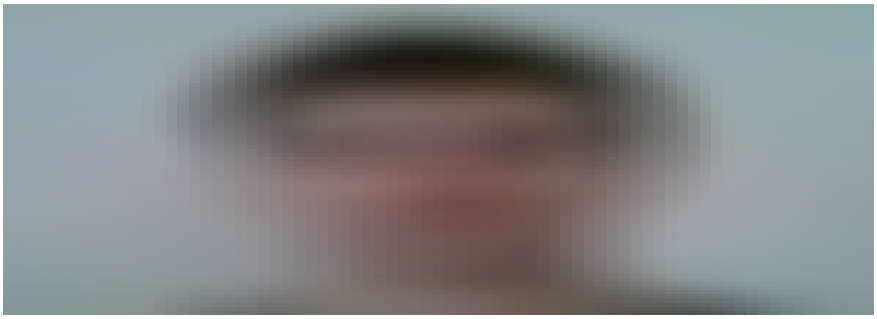
Inside the controller define a `create` action and `add_to_conversations`, `already_added?` and `create_group_conversation` methods within a private scope



controllers/group/conversations_controller.rb

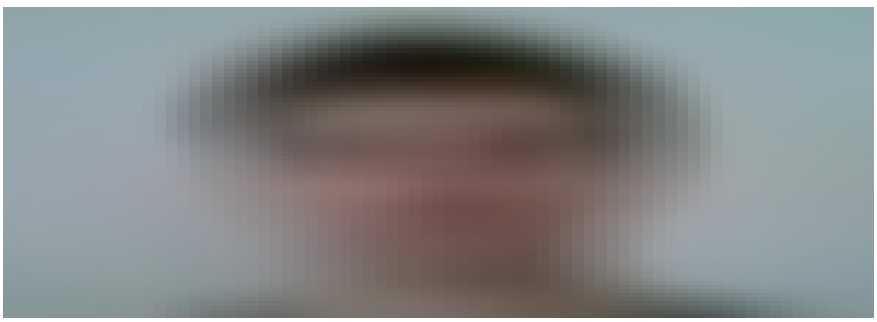
There is some complexity involved in creating a new group conversation, so we'll extract it into a service object. Then we have `add_to_conversations` and `already_added?` private methods. If you recall, we have them in the `Private::ConversationsController` too, but this time it stores group conversations' ids into the session.

Now define the `Group::NewConversationService` inside a new `group` directory



services/group/new_conversation_service.rb

The way a new group conversation is going to be created, is actually through a private conversation. We'll create this interface as an option on the private conversation's window soon. Before doing that, make sure that the service object functions properly by covering it with specs. Inside the `services`, create a new directory `group` with a `new_conversation_service_spec.rb` file inside

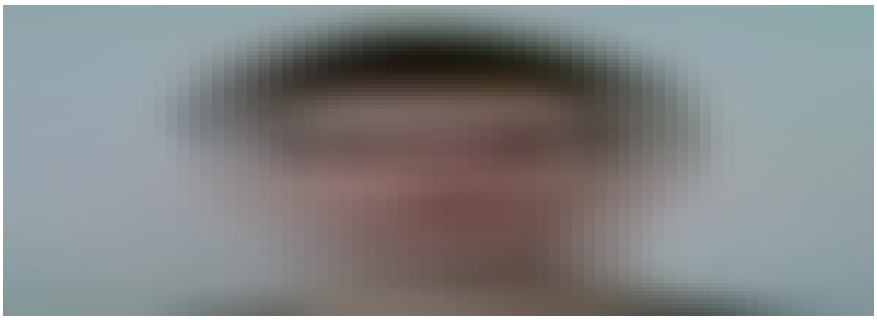


spec/services/group/new_conversation_service_spec.rb

Commit the changes

```
git add -A
git commit -m "Create back-end for creating a new group
conversation

- Create a Group::ConversationsController
  Define a create action and add_to_conversations,
  create_group_conversation and already_added? private
methods inside
- Create a  Group::NewConversationService and write specs
for it"
```

Define routes for the group conversation and its messages

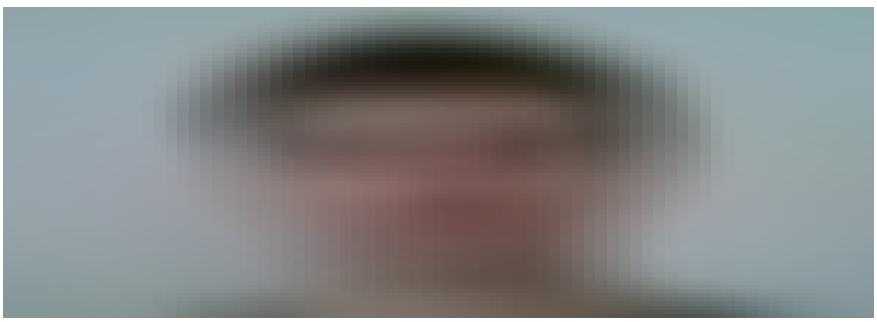


routes.rb

Commit the changes

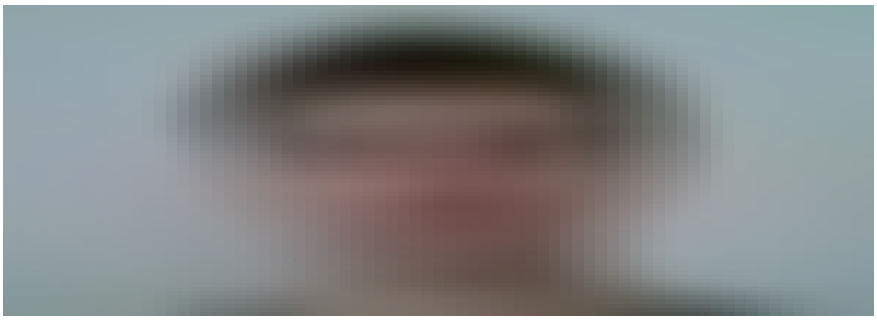
```
git add -A
git commit -m "Define specs for Group::Conversations and
Messages"
```

Currently we only take care of private conversations inside the `ApplicationController`. Only private conversations are being ordered and only their ids, after a user opens them, are available across the app. Inside the `ApplicationController`, update the `opened_conversations_windows` method



controllers/application_controller.rb

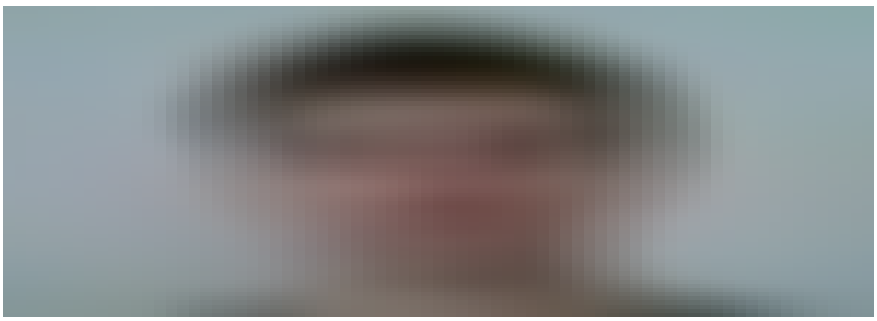
Because conversations' ordering happens with a help of the `OrderConversationsService`, we've to update this service



order_conversations_service.rb

Previously we only had the private conversations array and we sorted it by the latest messages' creation dates. Now we have private and group conversations arrays, then we join them together into one array and sort it the same way, as we did before.

Also update the specs

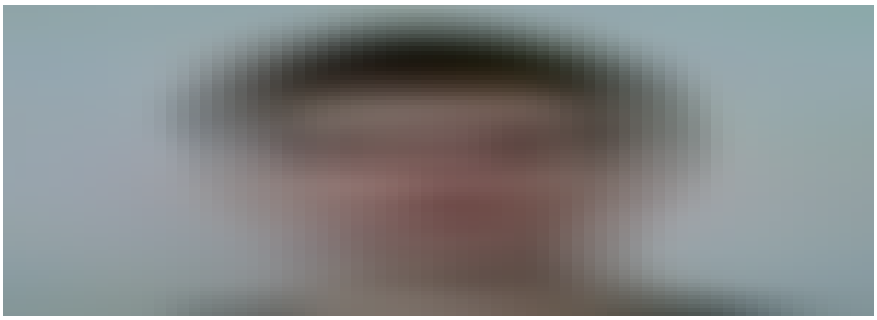


spec/services/order_conversations_service_spec.rb

Commit the changes

```
git add -A
git commit -m "Get data for group conversations in
ApplicationController

- Update the opened_conversations_windows method
- Update the OrderConversationsService"
```

In just a moment we'll need to pass some data from a controller to JavaScript. Luckily, we have already installed the `gon` gem, which allows us to do that easily. Inside the `ApplicationController`, within a `private` scope, add



controllers/application_controller.rb

Use the `before_action` filter to call this method

```
before_action :set_user_data
```

Commit the changes

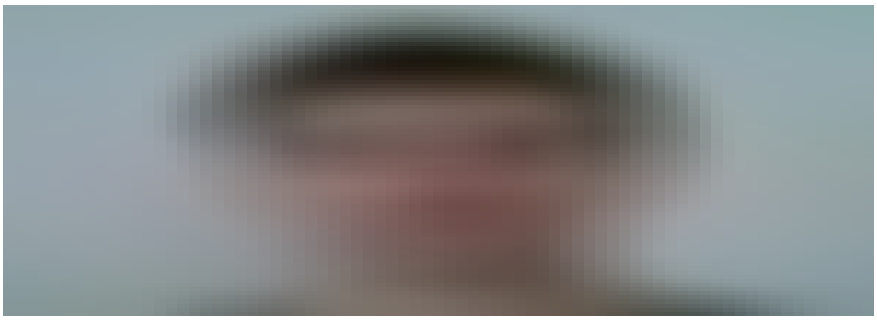
```
git add -A
git commit -m "Define a set_user_data private method in
ApplicationController"
```

Technically we can create a new group conversation now, but users have no interface to do that. As mentioned, they will do it through a private conversation. Let's create this option on the private conversation's window.

Inside the

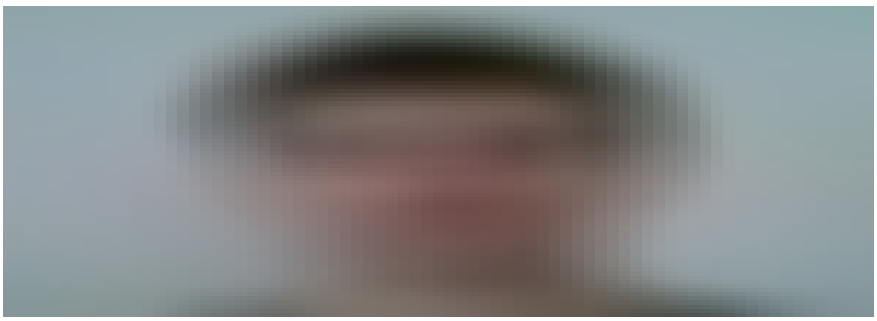
```
views/private/conversations/conversation/heading
```

directory create a new file



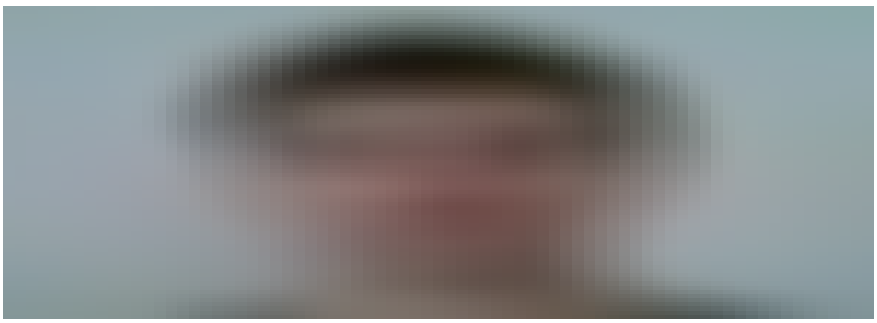
private/conversations/conversation/heading/_create_group_conversation.html.erb

A `collection_select` method is used to display a list of users. Only users who are in contacts are going to be included in the list. Define the `contacts_except_recipient` helper method



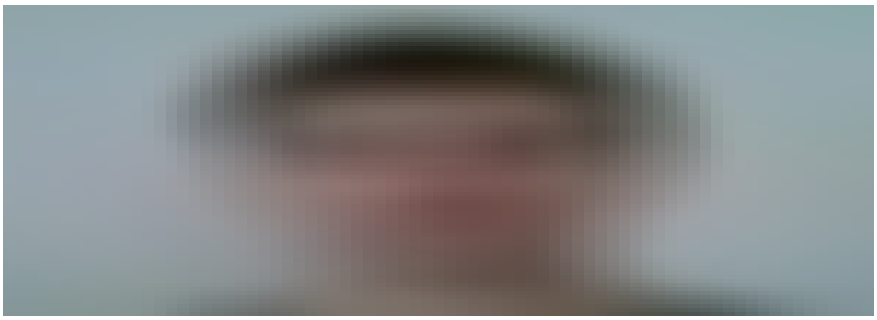
helpers/private/conversations_helper.rb

Write specs for the method



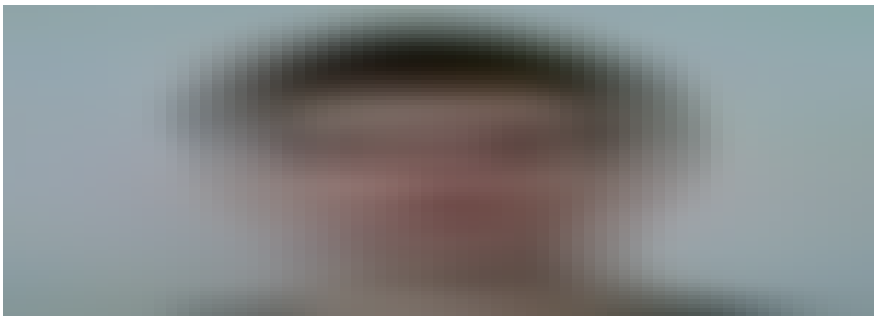
spec/helpers/private/conversations_helper_spec.rb

Render the partial at the bottom of the `_heading.html.erb`



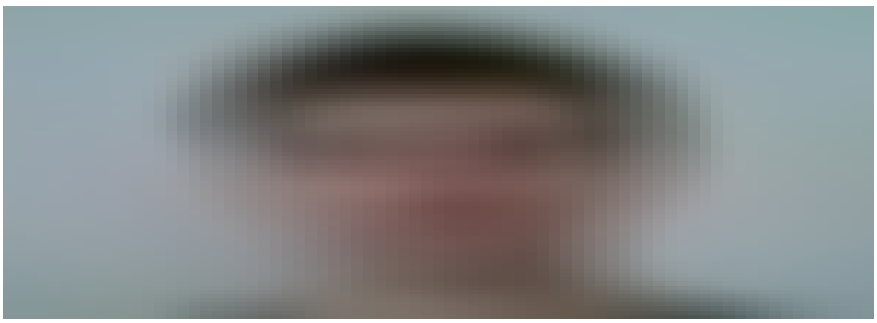
private/conversations/conversation/_heading.html.erb

Define the helper method



helpers/private/conversations_helper.rb

Wrap it with specs

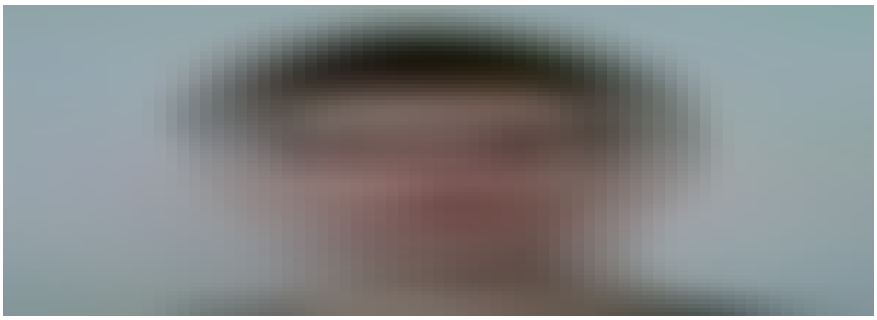


spec/helpers/private/conversations_helper_spec.rb

Commit the changes.

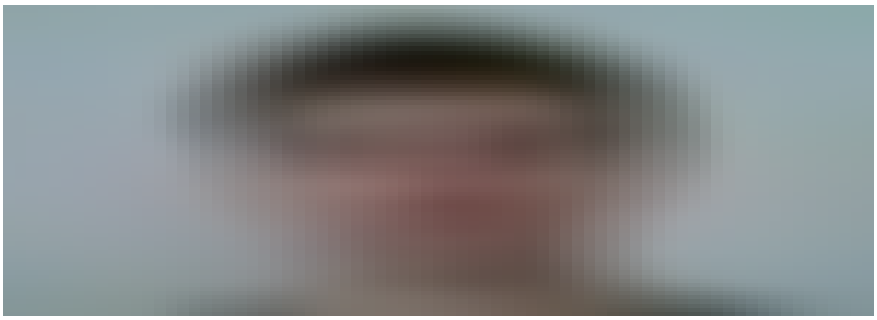
```
git add -A
git commit -m "Add a UI on private conversation to create a
group conversations"
```

Add CSS to style the component which allows to create a new group conversation



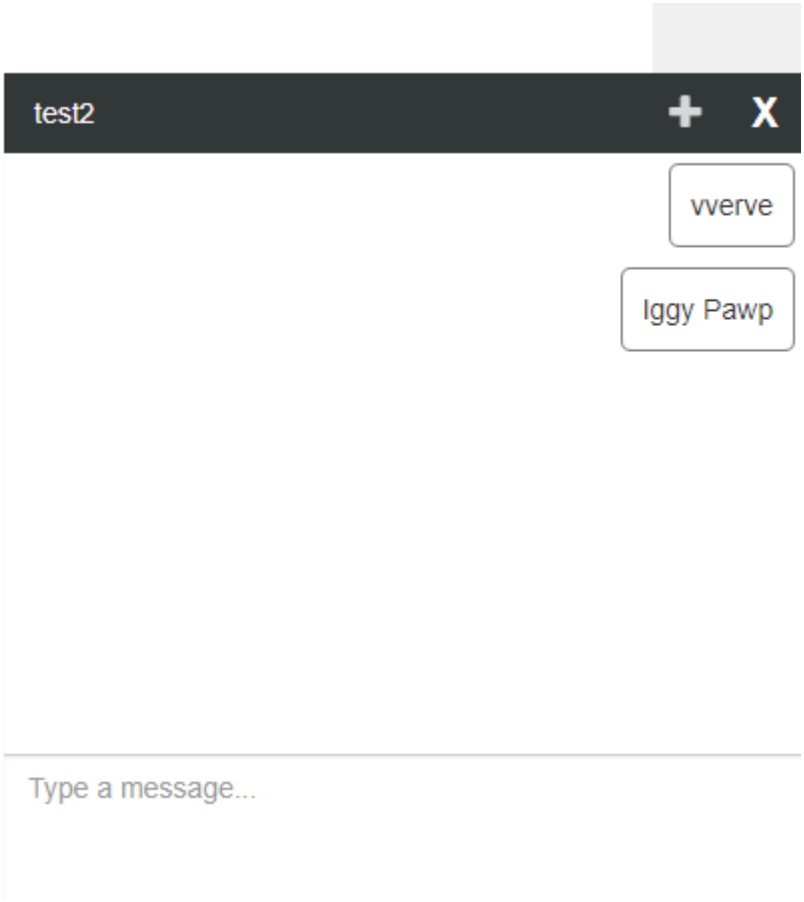
assets/stylesheets/partials/conversation_window.scss

A selection of contacts is hidden by default. To open the selection, a user has to click on the button. The button isn't interactive yet. Create an `options.js` file with JavaScript inside to make the selection list toggleable.

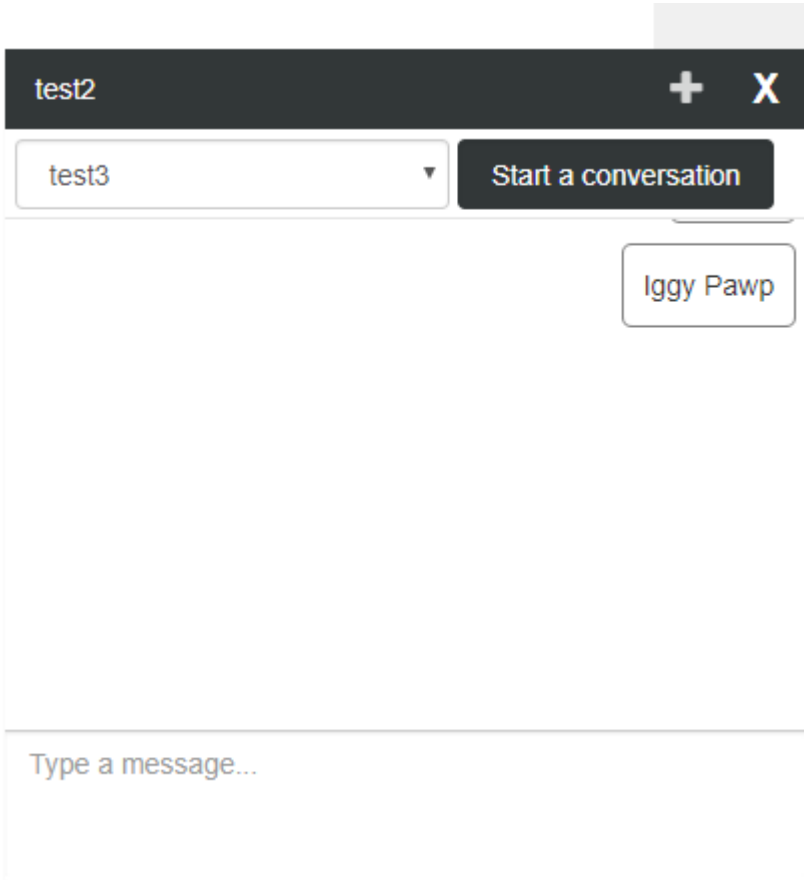


assets/javascripts/conversations/options.js

Now a conversation window with a recipient who is a contact looks like this



There is a button which opens a list of contacts, you can create a group conversation with, when you click on it

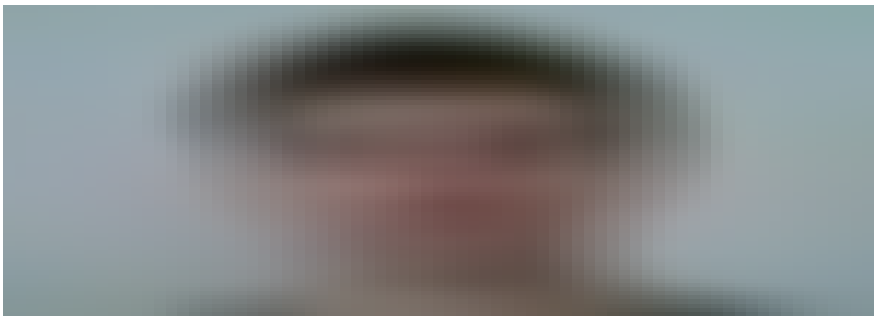


Commit the changes.

```
git add -A
git commit -m "
- Describe style for the create a group conversation option
- Make the option toggleable"
```

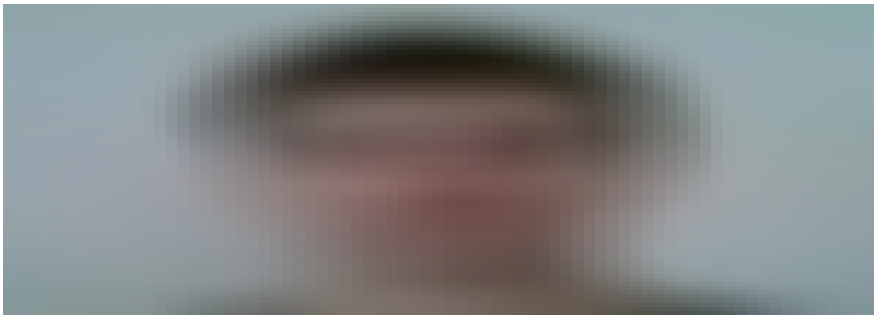
We have a list of ordered conversations, including group conversations now, which will be rendered on the navigation bar's drop down menu. If you recall, we specified different partials for different types of conversations. When the app tries to render a link, to open a group conversation, it will look for a different file than for a private conversation. The file isn't created yet.

Create a `_group.html.erb` file



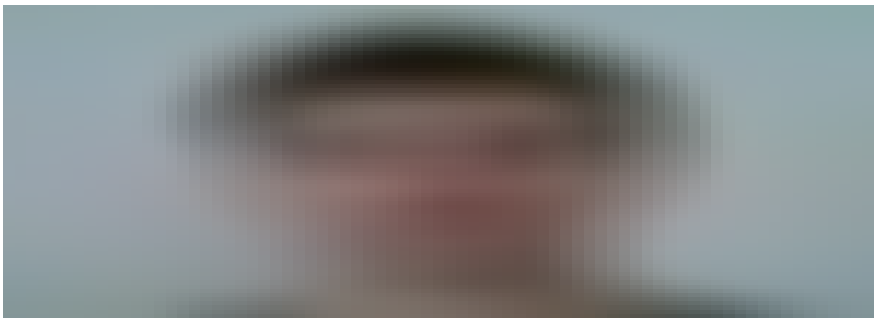
navigation/header/dropdowns/conversations/_group.html.erb

Define the `group_conv_seen_status` helper method inside the `Shared::ConversationsHelper`



helpers/shared/conversations_helper.rb

Write specs for the method

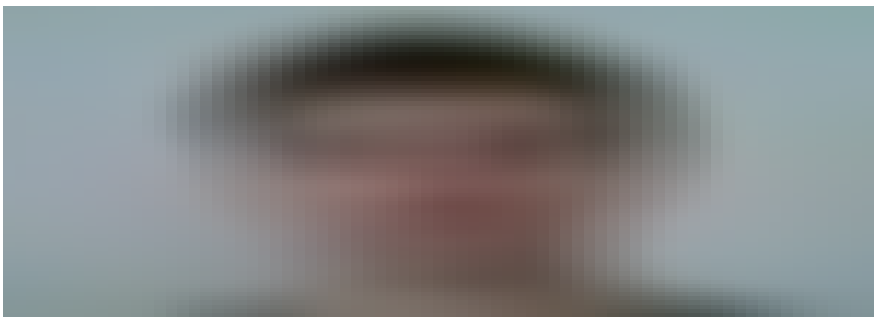


spec/helpers/shared/conversations_helper_spec.rb

Commit the changes.

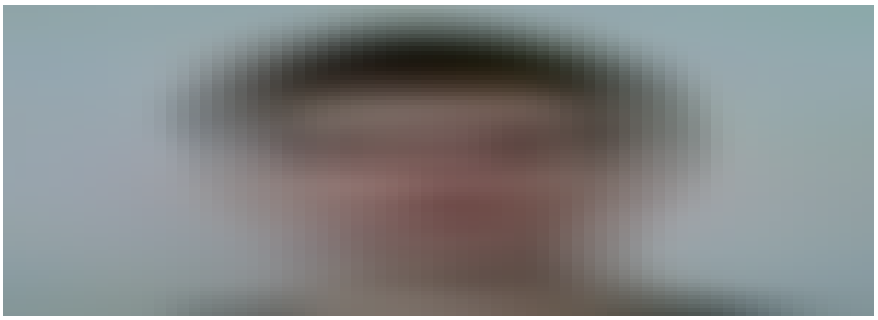
```
git add -A
git commit -m "Create a link on the navigation bar to open a
group conversation"
```

Render group conversations' windows on the app, the same way as we rendered the private conversations. Inside the `application.html.erb`, just below the rendered private conversations, add:



layouts/application.html.erb

Create the partial file to render group conversations' windows one by one:



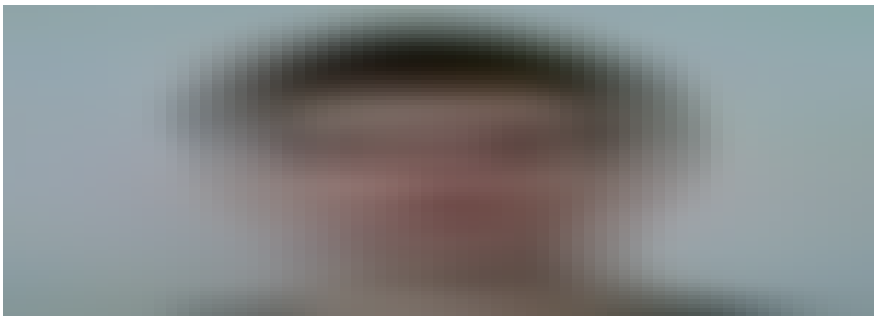
layouts/application/_group_conversations_windows.html.erb

Commit the change.

```
git add -A
git commit -m "Render group conversations' windows inside
the
application.html.erb"
```

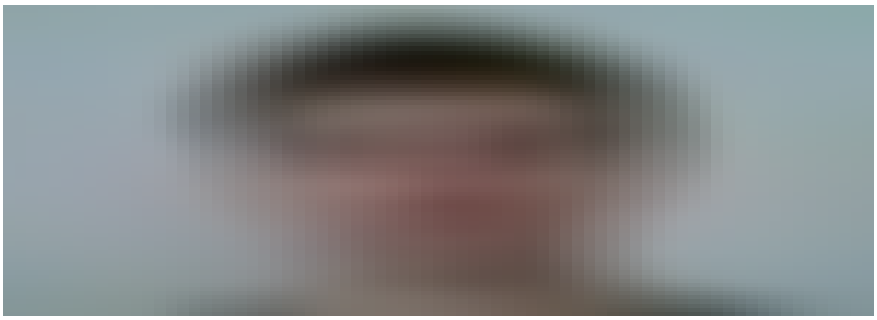
We have a mechanism how group conversations are created and rendered on the app. Now let's build a conversation window itself.

Inside the `group/conversations` directory, create a `_conversation.html.erb` file.



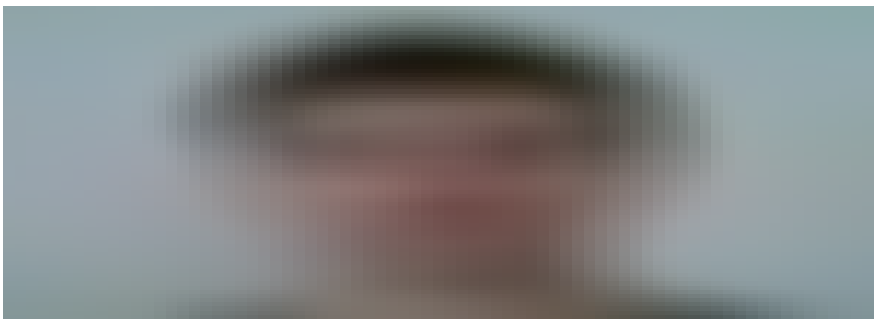
group/conversations/_conversation.html.erb

Define the `add_people_to_group_conv_list` helper method:



helpers/group/conversations_helper.rb

Write specs for the helper:



spec/helpers/group/conversations_helper_spec.rb

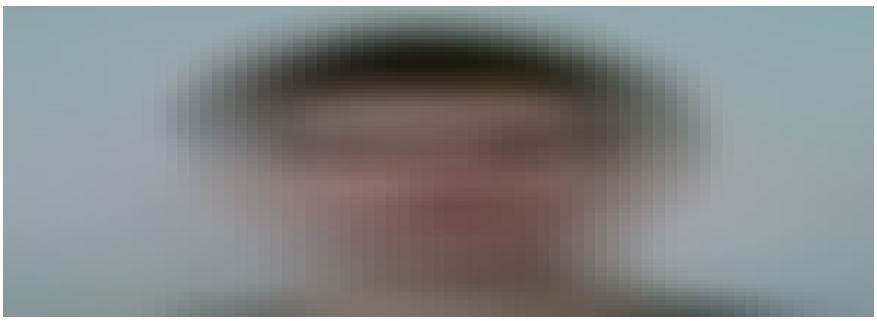
Just like with private conversations, group conversations are going to be accessible across the whole app, so obviously, we need access to the `Group::ConversationsHelper` methods everywhere too. Add this module inside the `ApplicationHelper`

```
include Group::ConversationsHelper
```

Commit the changes.

```
git add -A
git commit -m "
- Create a _conversation.html.erb inside the
group/conversations
- Define a add_people_to_group_conv_list and write specs for
it"
```

Create a new `conversation` directory with a `_heading.html.erb` file inside:

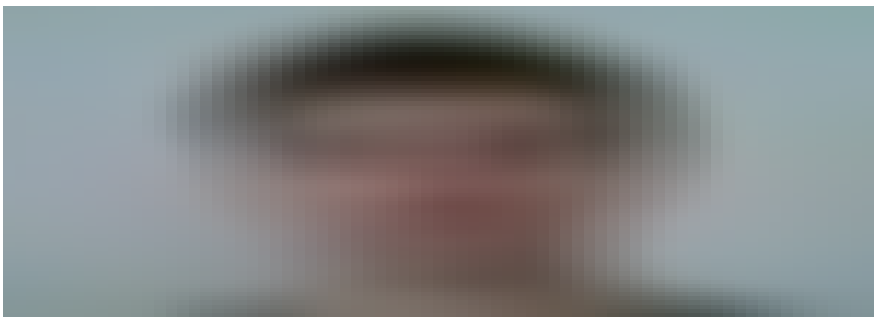


group/conversations/conversation/_heading.html.erb

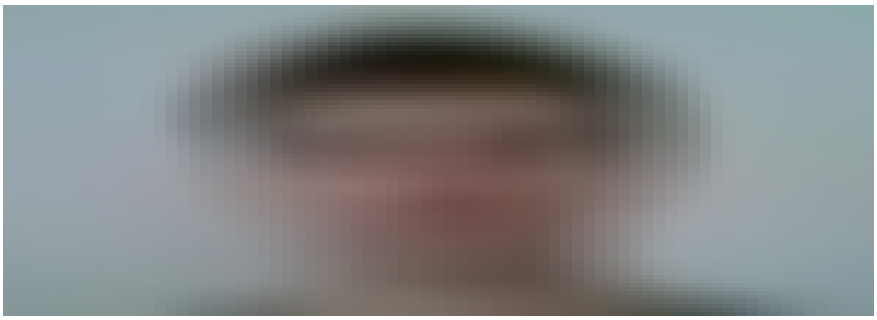
Commit the change.

```
git add -A
git commit -m "Create a _heading.html.erb inside the
group/conversations/conversation"
```

Next we have `_select_user.html.erb` and `_messages_list.html.erb` partial files. Create them:

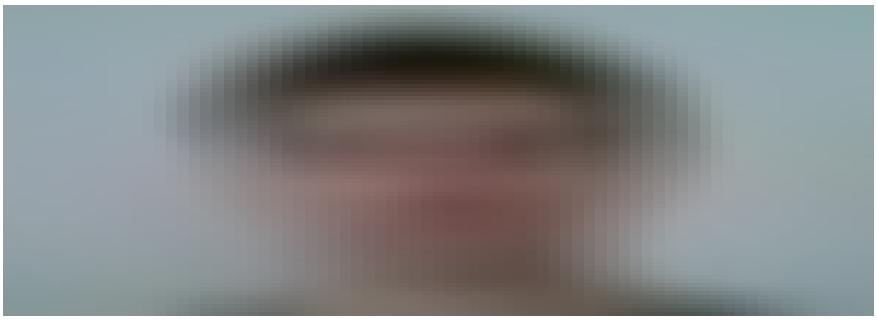


group/conversations/conversation/_select_users.html.erb



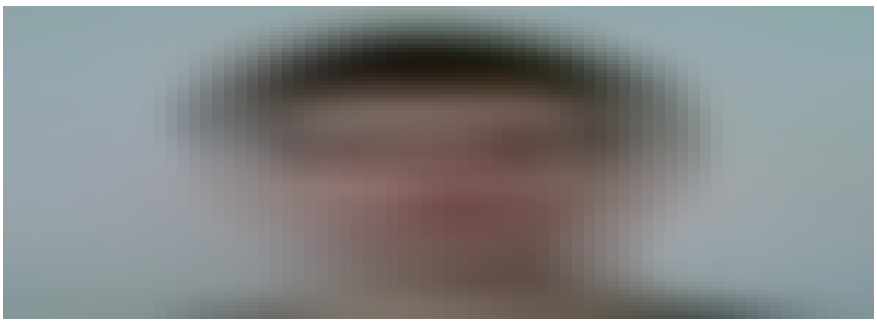
group/conversations/conversation/_messages_list.html.erb

Define the `load_group_messages_partial_path` helper method:



helper/group/conversations_helper.rb

Wrap it with specs:

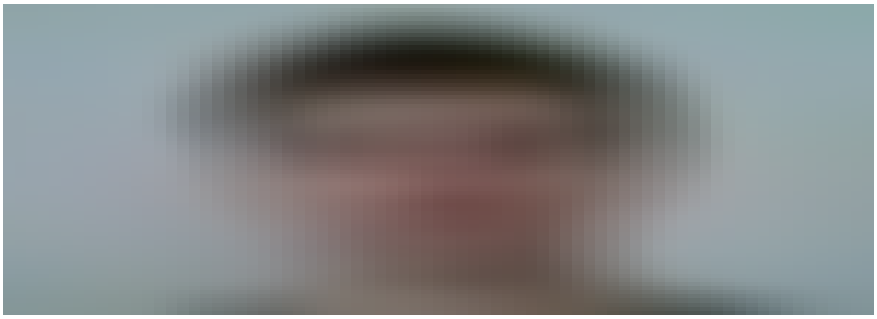


spec/helpers/group/conversations_helper_spec.rb

Commit the changes.

```
git add -A
git commit -m "
- Create _select_user.html.erb and _messages_list.html.erb
inside
  group/conversations/conversation
- Define a load_group_messages_partial_path helper method
  and write specs for it"
```

Create a `_link_to_previous_messages.html.erb` file, to have a link which loads previous messages:

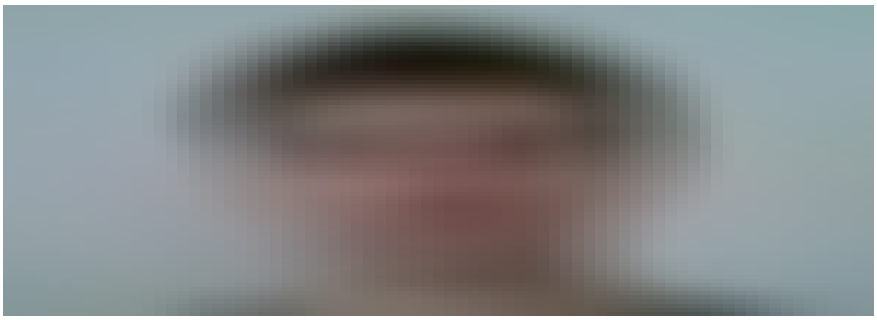


group/conversations/conversation/messages_list/_link_to_previous_messages.html.erb

Commit the change.

```
git add -A
git commit -m "Create a _load_messages.html.erb inside the
group/conversations/conversation/messages_list"
```

Create a new message form

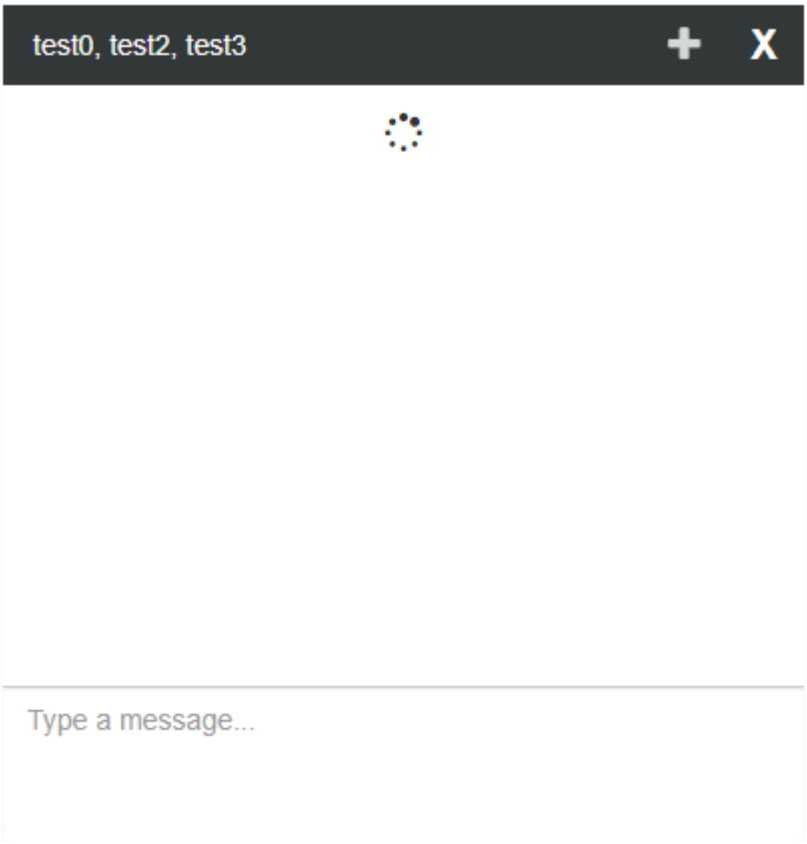


group/conversations/conversation/_new_message_form.html.erb

Commit the change.

```
git add -A
git commit -m "Create a _new_message_form.html.erb inside
the
group/conversations/conversation/"
```

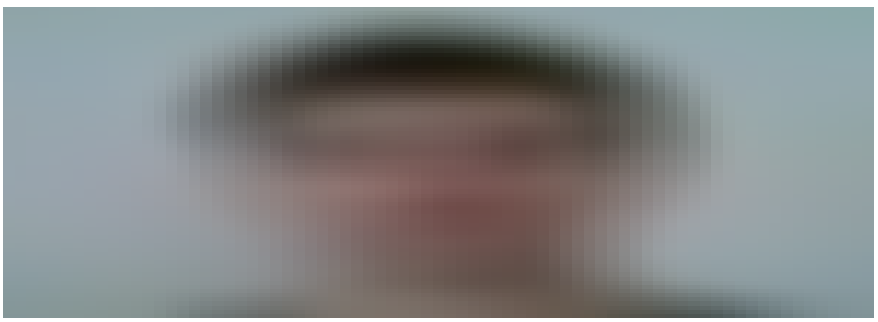
The application is now able to render group conversations' windows too.



But, they aren't functional yet. First, we need to load messages. We need a controller for messages and views. Generate a `Messages` controller:

```
rails g controller group/messages
```

Include the `Messages` module from `concerns` and define an `index` action:

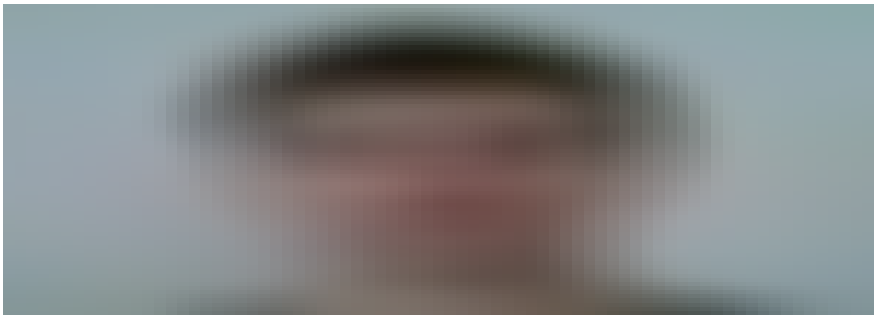


controllers/group/messages_controller.rb

Commit the changes.

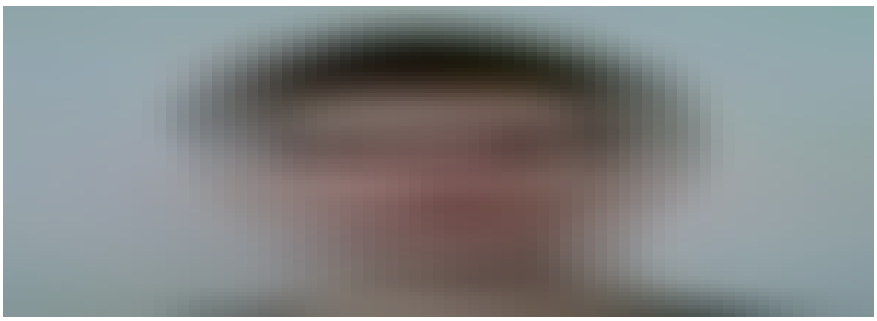
```
git add -A
git commit -m "Create a Group::MessagesController and define
an index action"
```

Create a `_load_more_messages.js.erb`



group/messages/_load_more_messages.js.erb

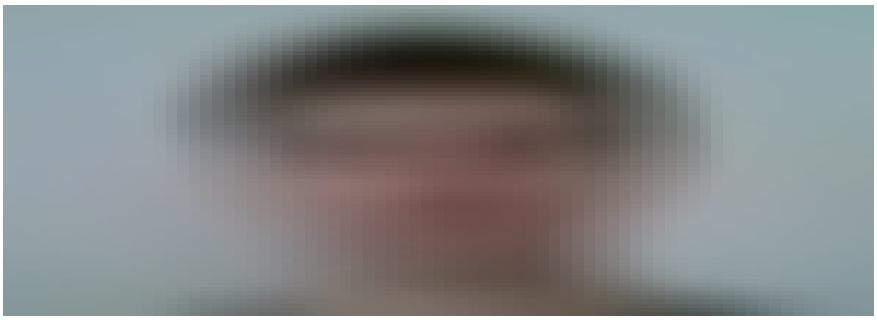
We've already defined the `append_previous_messages_partial_path` and `remove_link_to_messages` helper methods earlier on. We only have to define the `replace_link_to_group_messages_partial_path` helper method



helpers/group/messages_helper.rb

Again, this method, just like on the private side, is going to become more "intelligent", once we develop the messenger.

Create the `_replace_link_to_messages.js.erb`



group/messages/load_more_messages/window/_replace_link_to_messages.js.erb

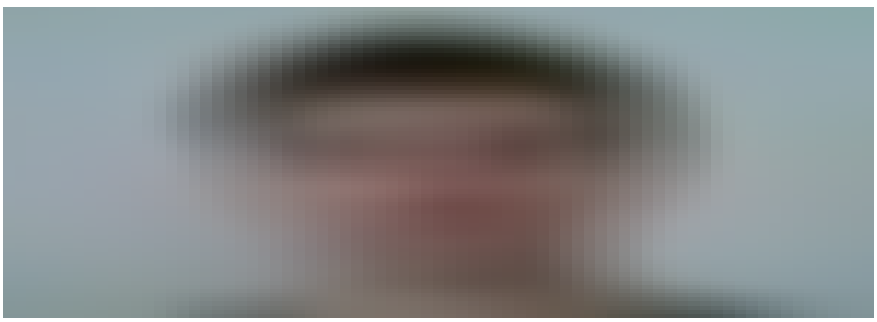
Also add the `Group::MessagesHelper` to the `ApplicationHelper`

```
include Group::MessagesHelper
```

Commit the changes.

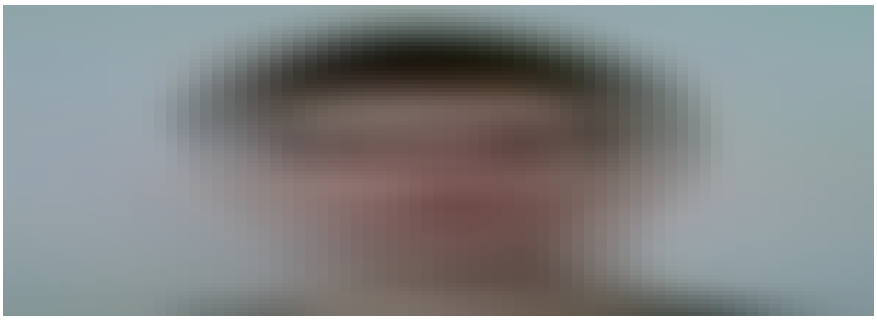
```
git add -A
git commit -m "Create a _load_more_messages.js.erb inside
the group/messages"
```

The only way group conversations can be opened right now is after their initialization. Obviously, this is not a thrilling thing, because once you destroy the session, there is no way to open the same conversation again. Create an `open` action inside the controller.



controllers/group/conversations_controller.rb

Create the `_open.js.erb` partial file:



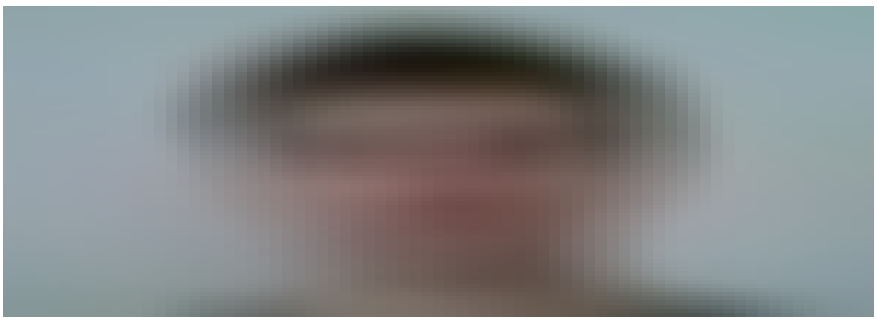
group/conversations/_open.js.erb

Now we're able to open conversations by clicking on navigation bar's drop down menu links. Try to test it with feature specs on your own.

Commit the changes.

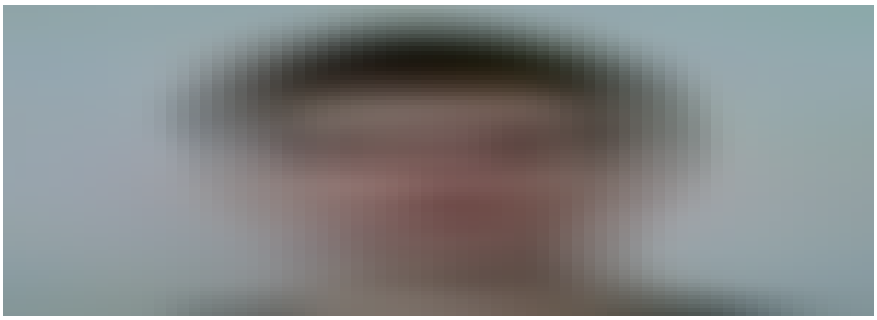
```
git add -A
git commit -m "Add ability to open group conversations

- Create an open action in the
Group::ConversationsController
- Create an _open.js.erb inside the group/conversations"
```

The app will try to render messages, but we haven't created any templates for them. Create a `_message.html.erb`



group/messages/_message.html.erb

Define `group_message_date_check_partial_path`, `group_message_seen_by` and `message_content_partial_path` helper methods.



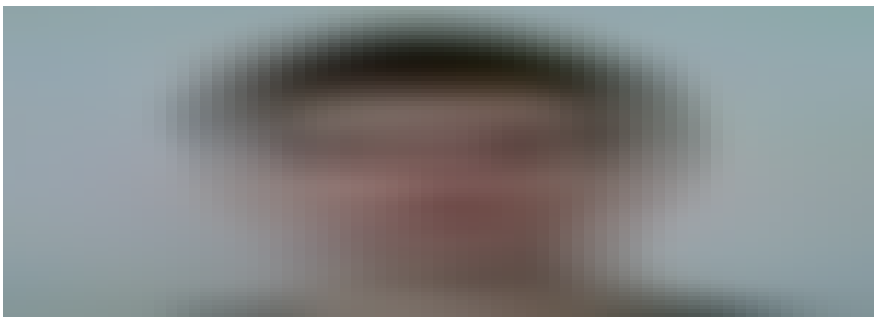
helpers/group_message_helper.rb

The `group_message_seen_by` method will return a list of users who have seen a message. This little information allows us to create extra features, like show to conversation participants who have seen messages, etc. But in our case, we'll use this information to determine if a current user has seen a message, or not. If not, then after the user sees it, the message is going to be marked as seen.

Also we'll need helper methods from the `Shared` module. Inside the `Group::MessagesHelper`, add the module.

```
require 'shared/messages_helper'
include Shared::MessagesHelper
```

Wrap helper methods with specs:



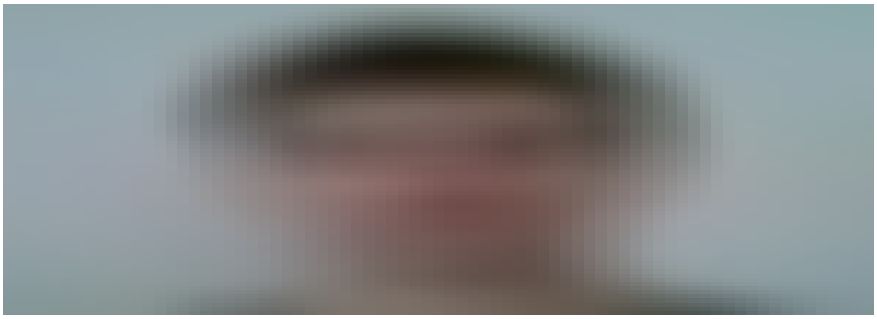
spec/helpers/group/messages_helper_spec.rb

Commit the changes.

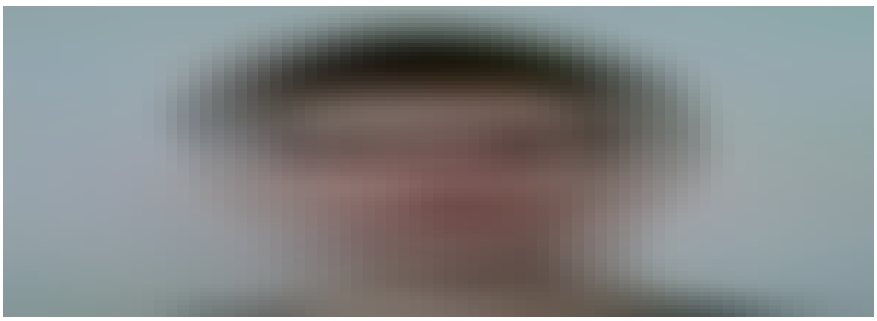
```
git add -A
git commit -m "Create a _message.html.erb inside
group/messages
```

```
- Define group_message_date_check_partial_path,
  group_message_seen_by and message_content_partial_path
helper
  methods and write specs for them"
```

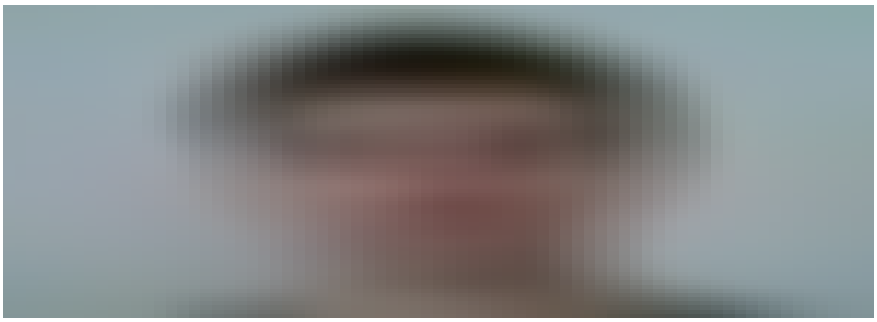
Create partial files for the message:



group/messages/_new_date.html.erb



group/messages/message/_different_user_content.html.erb



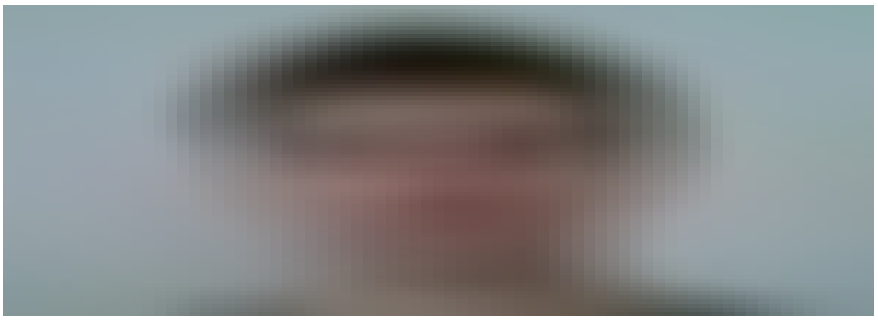
group/messages/message/_same_user_content.html.erb

Commit the changes.

```
git add -A
git commit -m "Create _new_date.html.erb,
_different_user_content.html.erb and
```

```
_same_user_content.html.erb
inside the group/messages/message/"
```

Now we need a mechanism which will render messages one by one. Create a `_messages.html.erb` partial file:

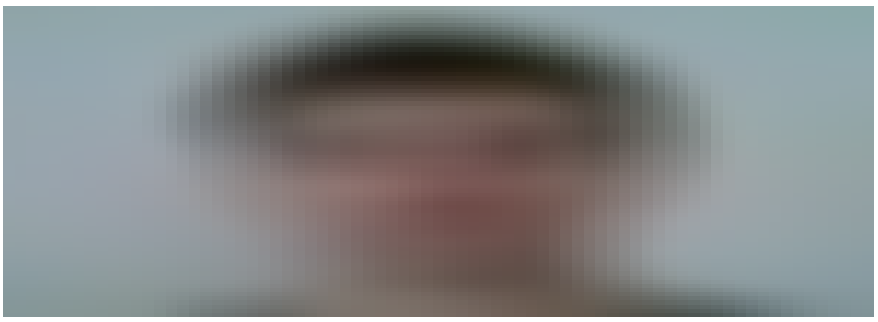


group/conversations/_messages.html.erb

Commit the change.

```
git add -A
git commit -m "Create _messages.html.erb inside
group/conversations"
```

Add styling for group messages:

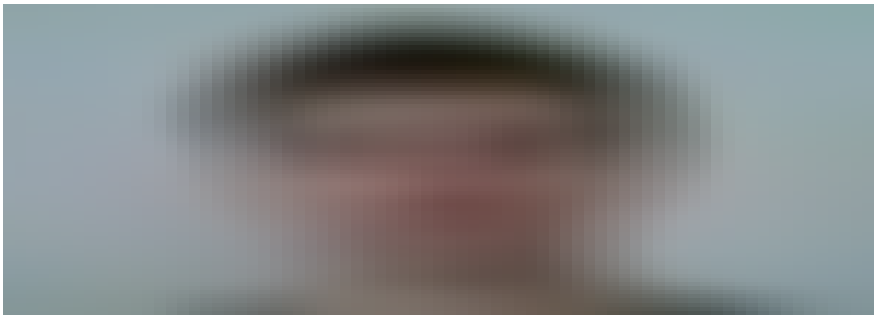


assets/stylesheets/partials/conversation_window.scss

Commit the change.

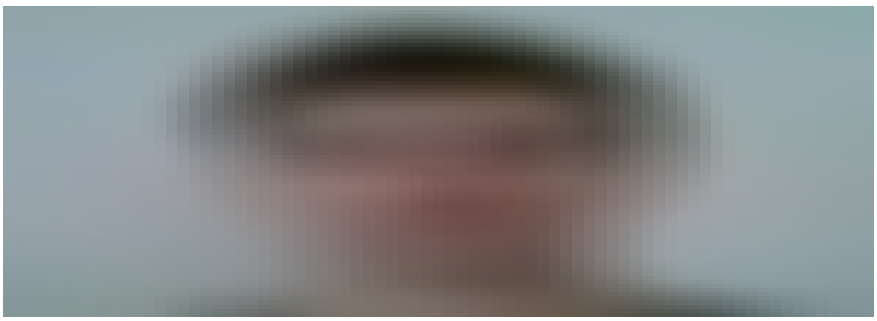
```
git add -A
git commit -m "Add CSS for group messages in
conversation_window.scss"
```

Make the close button functional by defining a `close` action inside the `Group::ConversationsController`



controllers/group/conversations_controller.rb

Create the corresponding template file:



group/conversations/close.js.erb

Commit the changes.

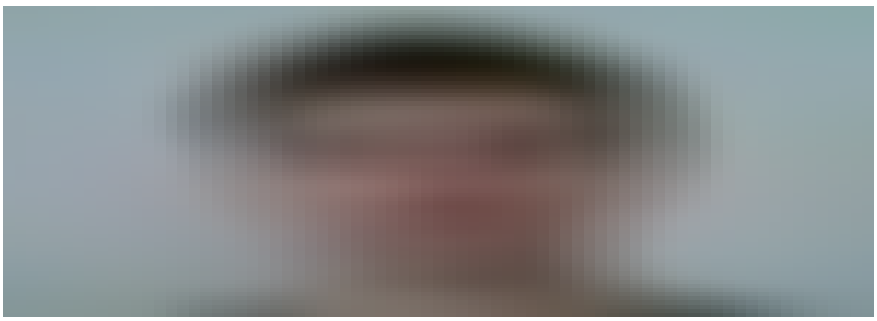
```
git add -A
git commit -m "Add a close group conversation window
functionality

- Define a close action inside the
Group::ConversationsController
- Create a close.js.erb inside the group/conversations"
```

Communicating in real time

Just like with private conversations, we want to be able to have conversations in real time with multiple users at the same window. The process of achieving this feature is going to be pretty similar to what we did with private conversations.

Generate a new channel for group conversations

```
rails g channel group/conversation
```



channels/group/conversation_channel.rb

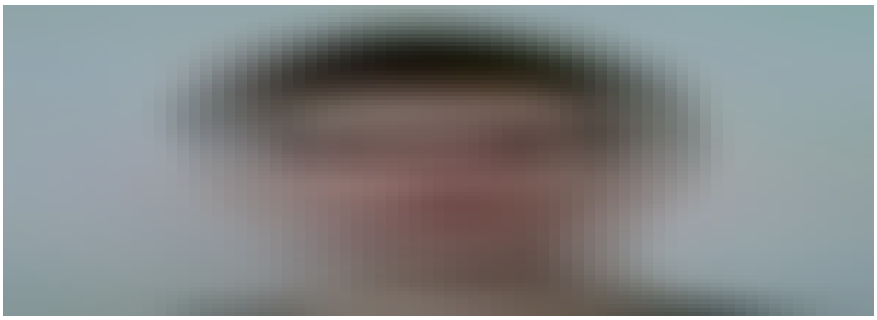
This time we check if a user belongs to a conversation, before establishing the connection, with the `belongs_to_conversation` method. In private conversations we streamed from a unique channel, by providing the `current_user`'s id. In a case of group conversations, an id of a conversation is passed from the client side. With the `belongs_to_conversation` method we check if users didn't do any manipulations and didn't try to connect to a channel which they don't belong to.

Commit the change

```
git add -A
git commit -m "Create a Group::ConversationChannel"
```

Create the `Group::MessageBroadcastJob`

```
rails g job group/message_broadcast
```

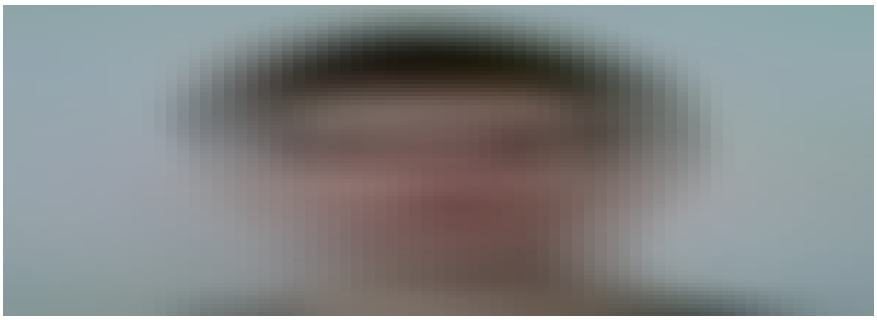


jobs/group/message_broadcast_job.rb

Commit the change.

```
git add -A
git commit -m "Create a Group::MessageBrodcastJob"
```

The last missing puzzle piece left—the client side:



assets/javascripts/channels/group/conversation.js

Essentially, it's very similar to the private conversation's `.js` file. The layout of the code is a little bit different. The main difference is an ability to pass conversation's `id` to a channel and a loop at the top of the file. With this loop we connect a user to all its group conversations' channels. That is the reason why we have used the `belongs_to_conversation` method on the server side. Id's of the conversations are passed from the client side. This method on the server side makes sure that a user really belongs to a provided conversation.

When you think about it, we could have just created this loop on the server side and wouldn't have to deal with all this confirmation process. But here's a reason why we pass an id of a conversation from the client side. When new users get added to a group conversation, we want to connect them immediately to the conversation's channel, without

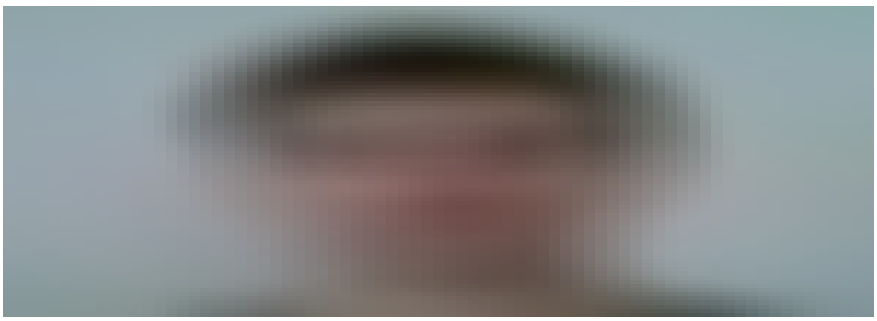
requiring to reload a page. The passable conversation's id allows us to effortlessly achieve that. In the upcoming section we'll create a unique channel for every user to receive notifications in real time. When new users will be added to a group conversation, we'll call the `subToGroupConversationChannel` function, through their unique notification channels, and connect them to the group conversation channel. If we didn't allow to pass a conversation's id to a channel, connections to new channels would have occurred only after a page reload. We wouldn't have any way to connect new users to a conversation channel dynamically.

Now we are able to send and receive group messages in real time. Try to test the overall functionality with specs on your own.

Commit the changes.

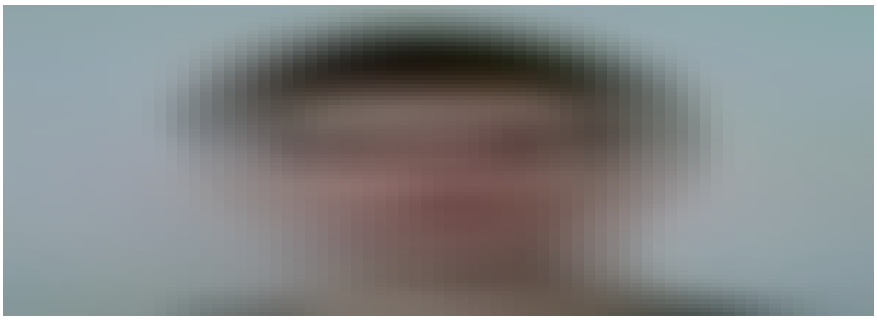
```
git add -A
git commit -m "Create a conversation.js inside the
assets/javascripts/channels/group"
```

Inside the `Group::ConversationsController` define an `update` action



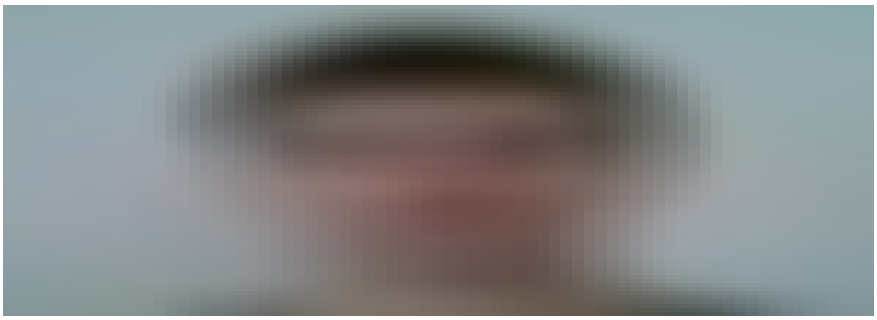
controllers/group/conversations_controller.rb

Create the `Group::AddUserToConversationService`, which is going to take care that a selected user will be added to a conversation



services/group/add_user_to_conversation_service.rb

Test the service with specs:



spec/services/group/add_user_to_conversation_service_spec.rb

We have working private and group conversations now. A few nuances are still missing, which we will implement later, but the core functionality is here. Users are able to communicate one on one, or if they need, they can build an entire chat room with multiple people.

Commit the changes.

```
git add -A
git commit -m "Create a Group::AddUserToConversationService
and test it"
```

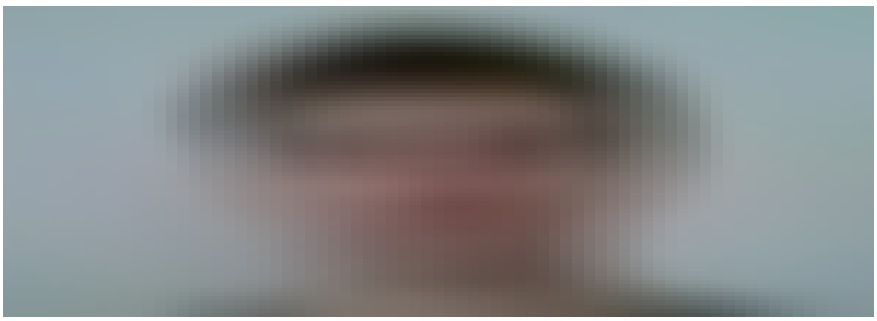
Messenger

What is the purpose of having a messenger? On mobile screens instead of opening a conversation window, the app will load the messenger. On bigger screens, users could choose where to chat, on the conversation window or on the messenger. If the messenger is going to fill the whole browser's window, it should be more comfortable to communicate.

Since we'll use the same data and models, we just need to open conversations in a different environment. Generate a new controller to

handle requests to open a conversation inside the messenger.

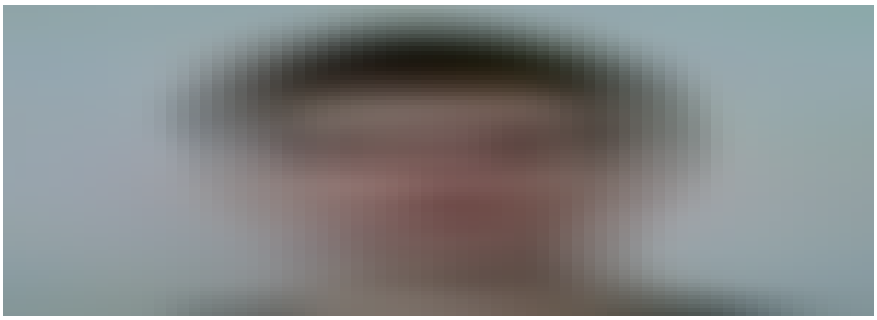
```
rails g controller messengers
```



controllers/messengers_controller.rb

`get_private_conversation` and `get_group_conversation` actions will get a user's selected conversation. Those actions' templates are going to append a selected conversation to the conversation placeholder. Every time a new conversation is selected to be opened the old one gets removed and replaced with a newly selected one.

Define routes for the actions:



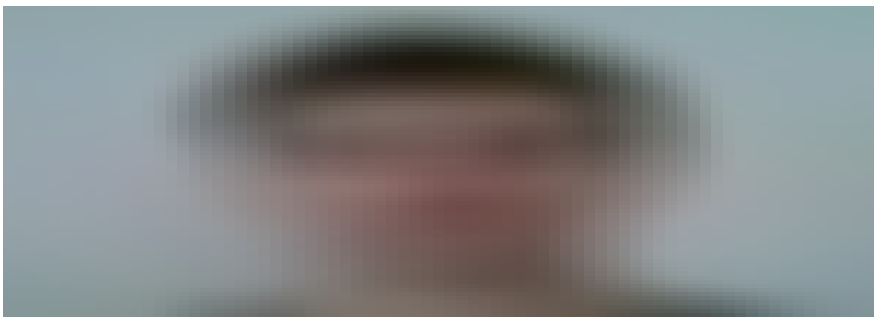
routes.rb

Commit the changes.

```
git add -A
git commit -m "Create a MessengersController and define
routes to its actions"
```

In the controller is an `open_messenger` action. The purpose of this action is to go from any page straight to the messenger and render a selected conversation. On smaller screens users are going to chat through messenger instead of conversation windows. In just a moment, we'll switch links for smaller screens to open conversations inside the messenger.

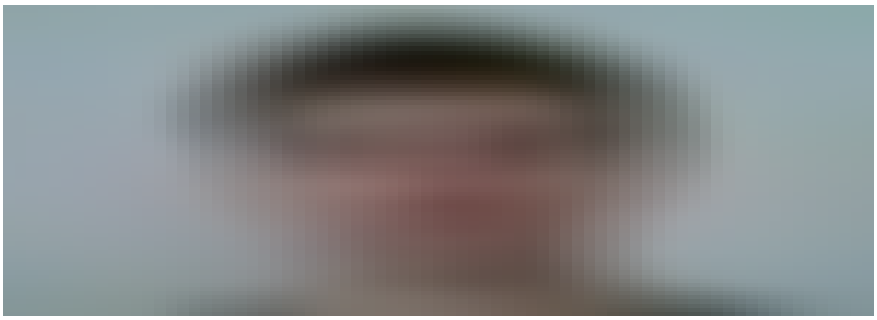
Create a template for the `open_messenger` action



messengers/open_messenger.html.erb

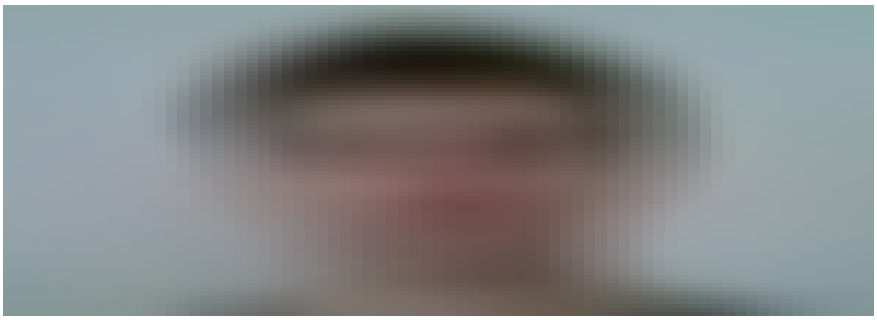
```
git add -A
git commit -m "Create an open_messenger.html.erb in the
/messengers"
```

Then we see the `ConversationForMessengerSerivce`, it retrieves a selected conversation's object. Create the service:



services/conversation_for_messenger_service.rb

Add specs for the service:

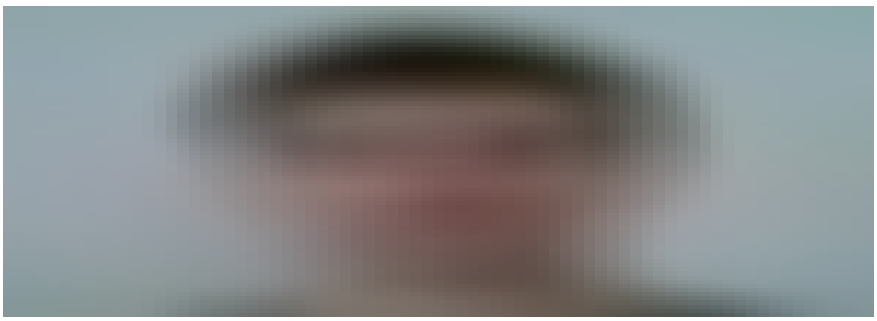


spec/services/conversation_for_messenger_service_spec.rb

Commit the changes.

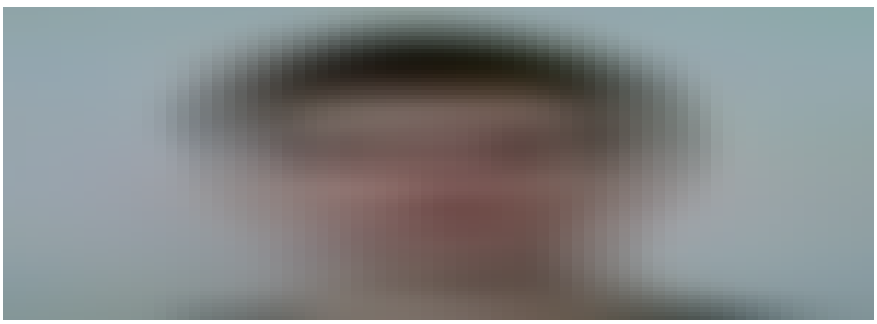
```
git add -A
git commit -m "Create a ConversationForMessengerSerivce and
add specs for it"
```

Create a template for the `index` action:



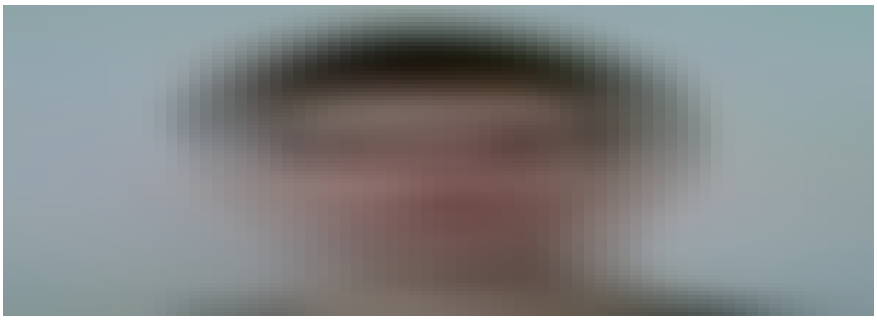
messengers/index.html.erb

This is going to be the messenger itself. Inside the messenger, we'll be able to see a list of user's conversations and a selected conversation. Create the partial files:



messengers/index/_conversations_list.html.erb

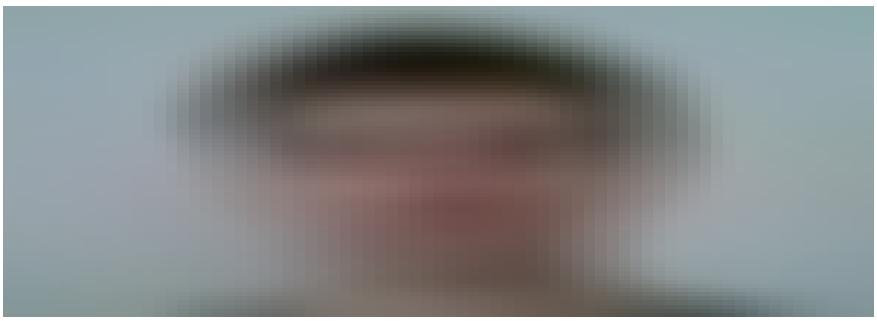
Define the helper method:



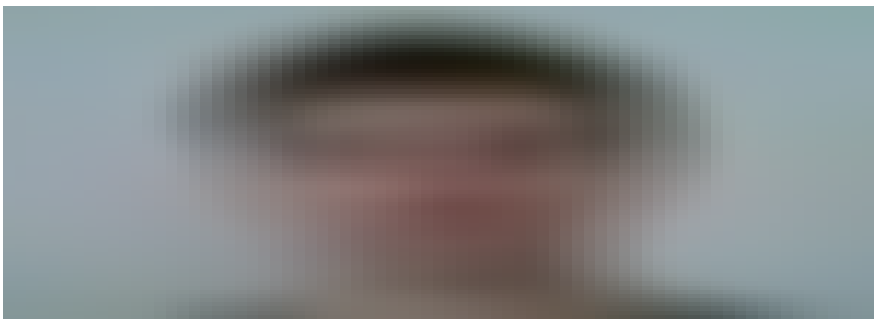
helpers/messengers_helper.rb

Try to test it with specs yourself.

Create the partial files for links to open conversations:

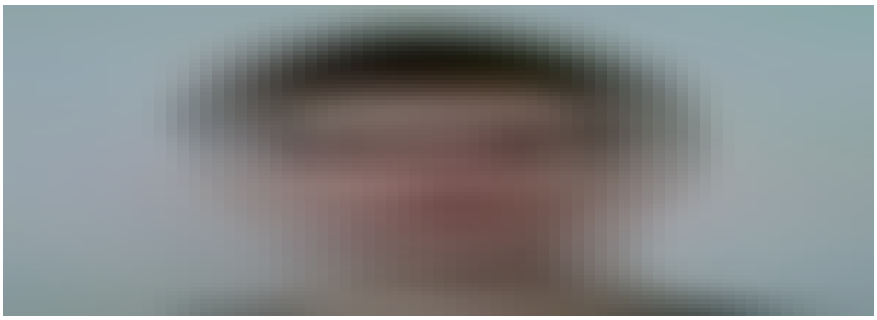


messengers/index/conversations_list_item/_private.html.erb



messengers/index/conversations_list_item/_group.html.erb

Now create a partial for the conversation space, selected conversations are going to be rendered there:

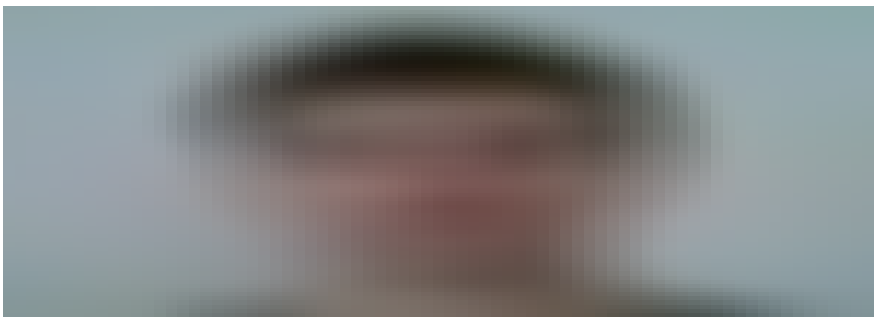


messengers/index/_conversation.html.erb

Commit the changes.

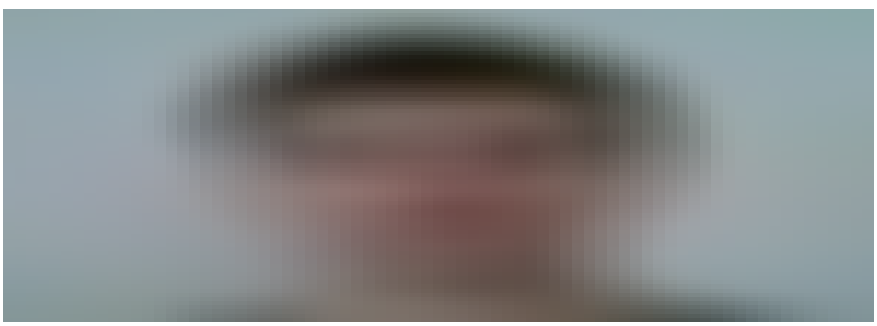
```
git add -A
git commit -m "Create a template for the
MessengersController's index action"
```

Create a template for the `get_private_conversation` action:



messengers/get_private_conversation.js.erb

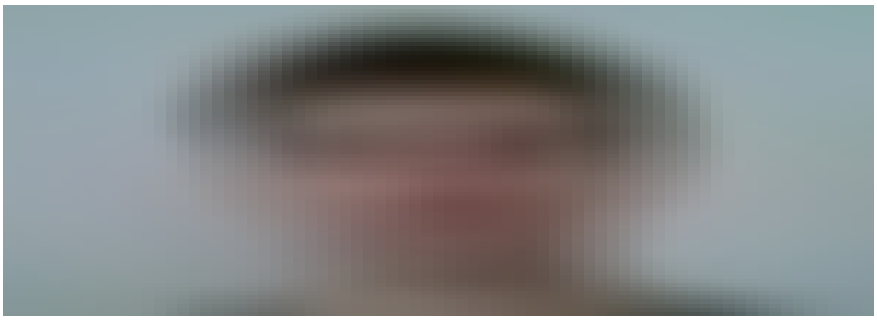
Create a `_private_conversation.html.erb` file:



messengers/_private_conversation.html.erb

This file will render a private conversation inside the messenger. Also notice that we reuse some partials from the private conversation views.

Create the `_details.html.erb` partial:



messengers/private_conversation/_details.html.erb

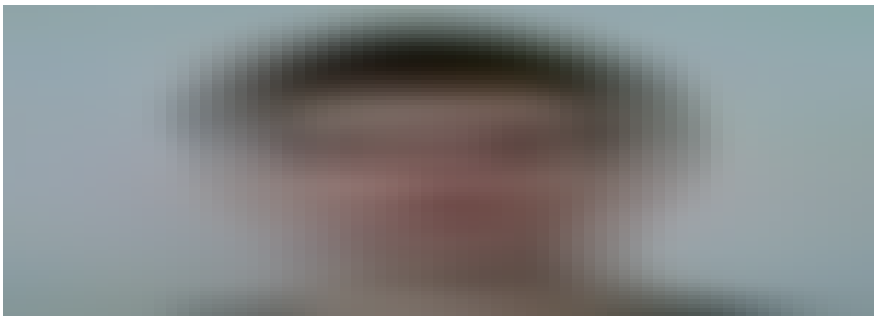
Commit the changes.

```
git add -A
git commit -m "Create a template for the
MessengersController's
get_private_conversation action"
```

When we go to the messenger, it's better to not see drop down menus on the navigation bar. Why? We don't want to render conversation windows inside the messenger, otherwise it would look chaotic. A conversation window and the messenger at the same time to chat with the same person. That would be a highly faulty design.

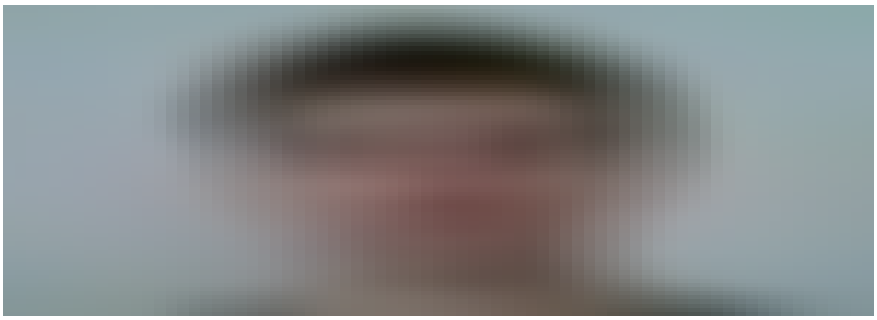
At first, forbid conversations' windows to be rendered on the messenger's page. Not that hard to do. To control it, remember how conversations' windows are rendered on the app. They are rendered inside the `application.html.erb` file. Then we have `@private_conversations_windows` and `@group_conversations_windows` instance variables. Those variables are arrays of conversations. Instead of just rendering conversations from those arrays, define helper methods to decide to give those arrays to users or not, depending on which page they are in. If users are inside the messenger's page, they will get an empty array and no conversations' windows will be rendered.

Replace those instance variables with `private_conversations_windows` and `group_conversations_windows` helper methods. Now define them inside the `ApplicationHelper`



helpers/application_helper.rb

Wrap them with specs



spec/helpers/application_helper_spec.rb

Commit the changes

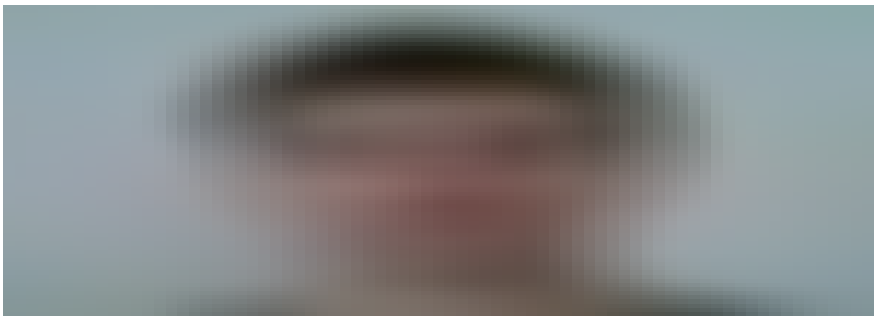
```
git add -A
git commit -m "
Define private_conversations_windows and
group_conversations_windows
helper methods inside the ApplicationHelper and test them"
```

Next, create an alternative partial file for navigation's header, so drop down menus won't be rendered. Inside the `NavigationHelper`, we've defined the `nav_header_content_partials` helper method before. It determines which navigation's header to render.

Inside the

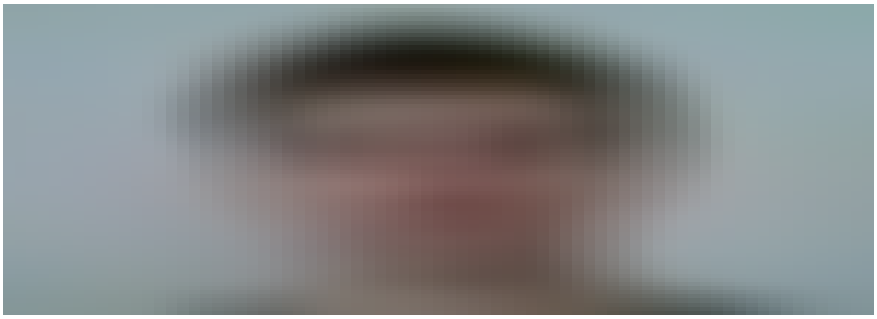
```
layouts/navigation/header
```

directory, create a `_messenger_header.html.erb` file



layouts/navigation/header/_messenger_header.html.erb

Style the messenger. Create a `messenger.scss` file inside the `partials` directory

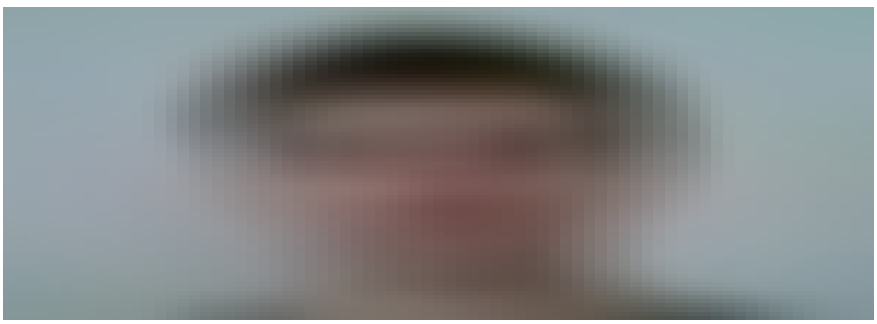


assets/stylesheets/partials/messenger.scss

Commit the change

```
git add -A
git commit -m "Create a messenger.scss inside the partials"
```

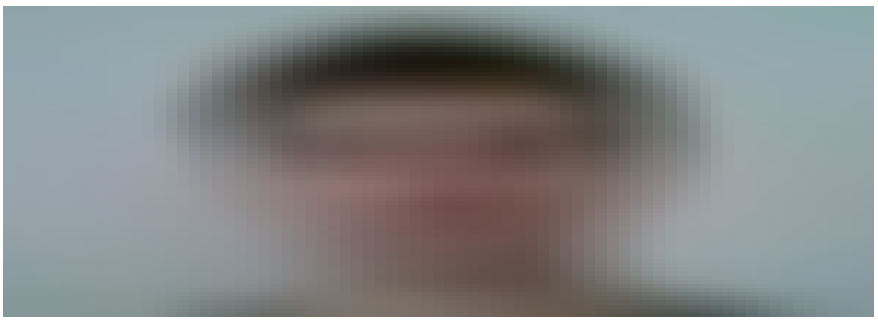
Inside the `desktop.scss`, within the `min-width: 767px`, add



assets/stylesheets/responsive/desktop.scss

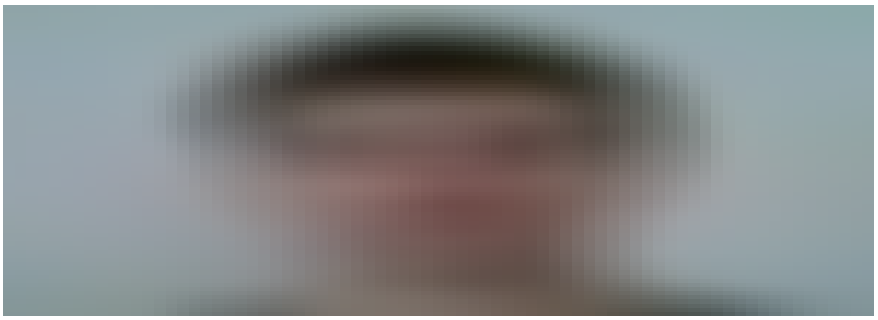
When we click on a conversation to open it, we want to be able to load previous messages somehow. We could add a visible link for loading

them. Or we can automatically load some amount of messages until the scroll bar appears, so a user could load previous messages by scrolling up. Create a helper method which will take care of it

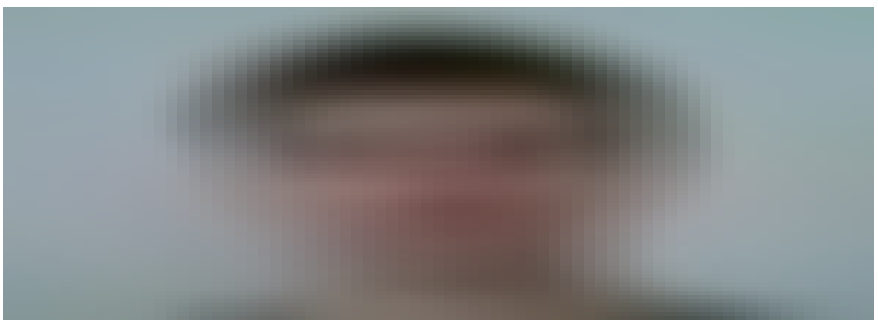


helpers/shared/messages_helper.rb

Test it with specs on your own. Create the partial files



shared/load_more_messages/messenger/_load_previous_messages.js.erb

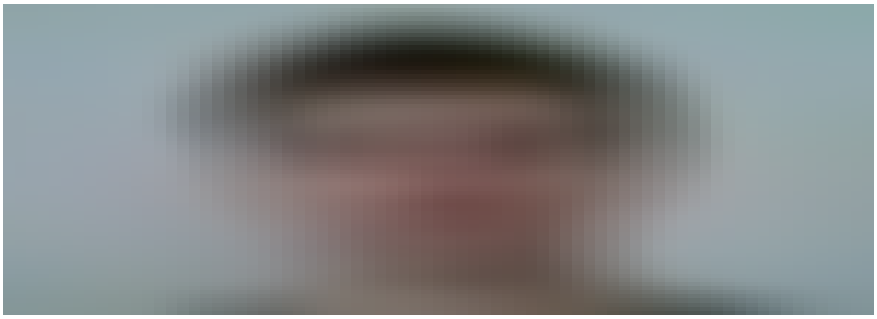


shared/load_more_messages/messenger/_remove_previous_messages_link.js.erb

Commit the changes

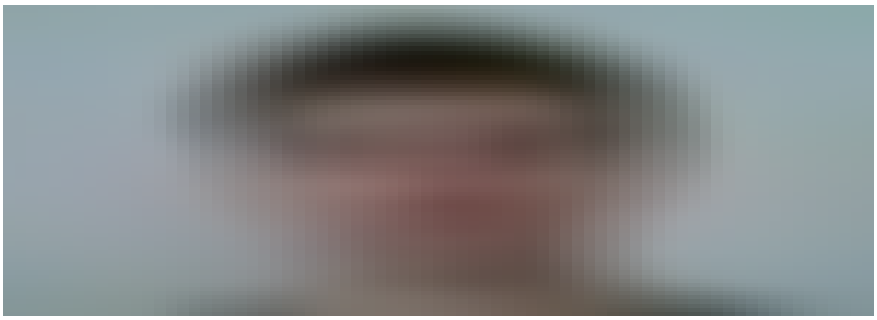
```
git add -A
git commit -m "Define an autoload_messenger_messages in the
Shared::MessagesHelper"
```

Use the helper method inside the `_load_more_messages.js.erb` file, just above the `<%= render remove_link_to_messages %>`



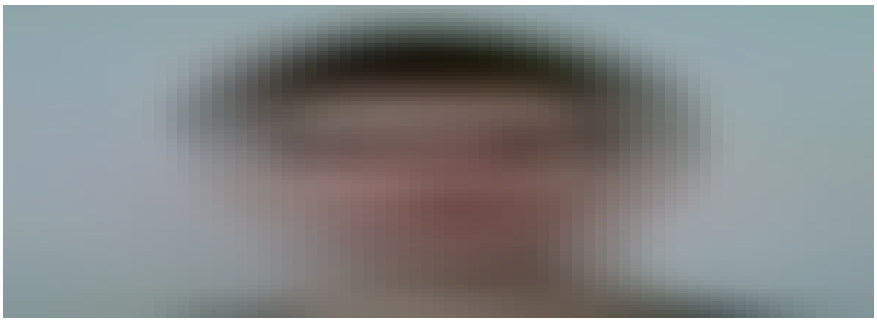
private/messages/_load_more_messages.js.erb

Now we have `append_previous_messages_partial_path` and `replace_link_to_private_messages_partial_path` helper methods which we should update, to make them compatible with the messenger



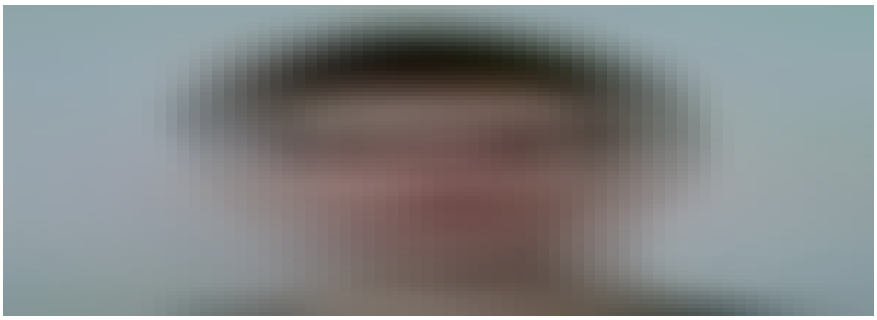
helpers/shared/messages_helper.rb

Create a missing partial file



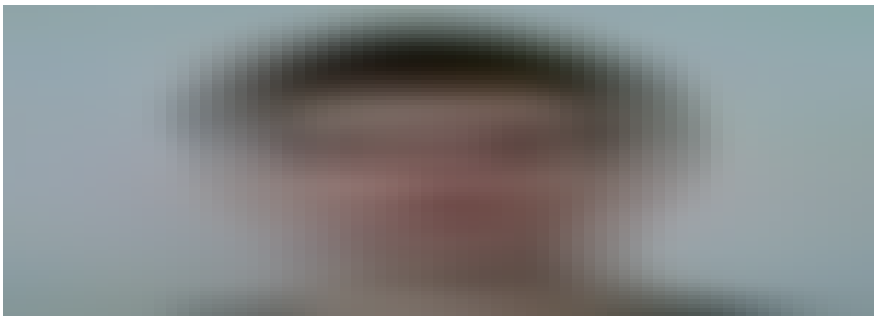
shared/load_more_messages/messenger/_append_messages.js.erb

Update another method



helpers/private/messages_helper.rb

Create the partial file



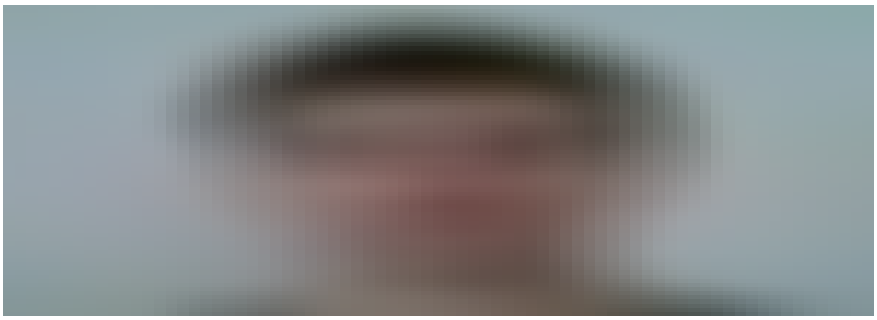
private/messages/load_more_messages/messenger/_replace_link_to_messages.js.erb

Test the helper methods with specs on your own.

Commit the changes

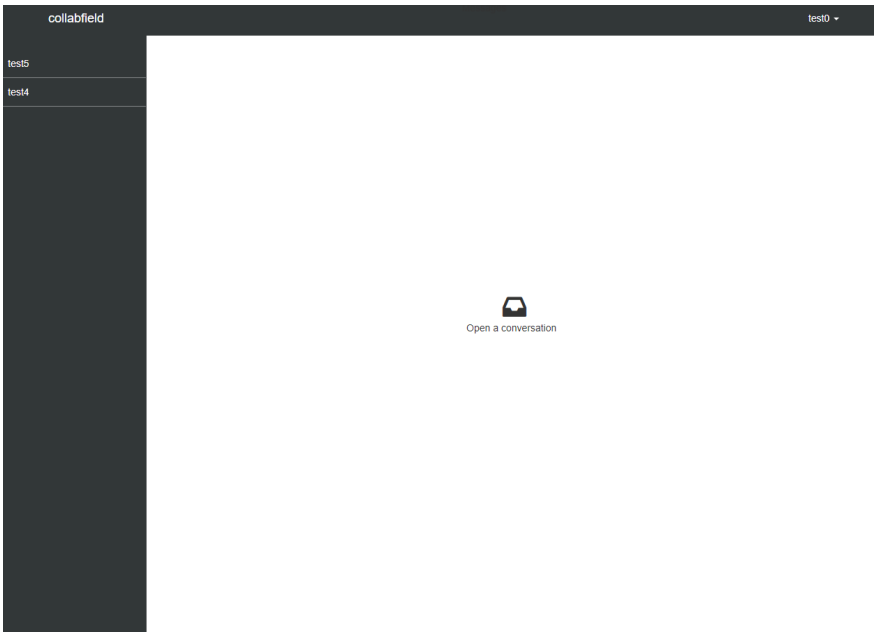
```
git add -A
git commit -m "
- Update the append_previous_messages_partial_path helper
method in
  Shared::MessagesHelper
- Update the replace_link_to_private_messages_partial_path
method in
  Private::MessagesHelper"
```

Now after an initial load messages link click, the app will automatically keep loading previous messages until there is a scroll bar on the messages list. To make the initial click happen, add some JavaScript:

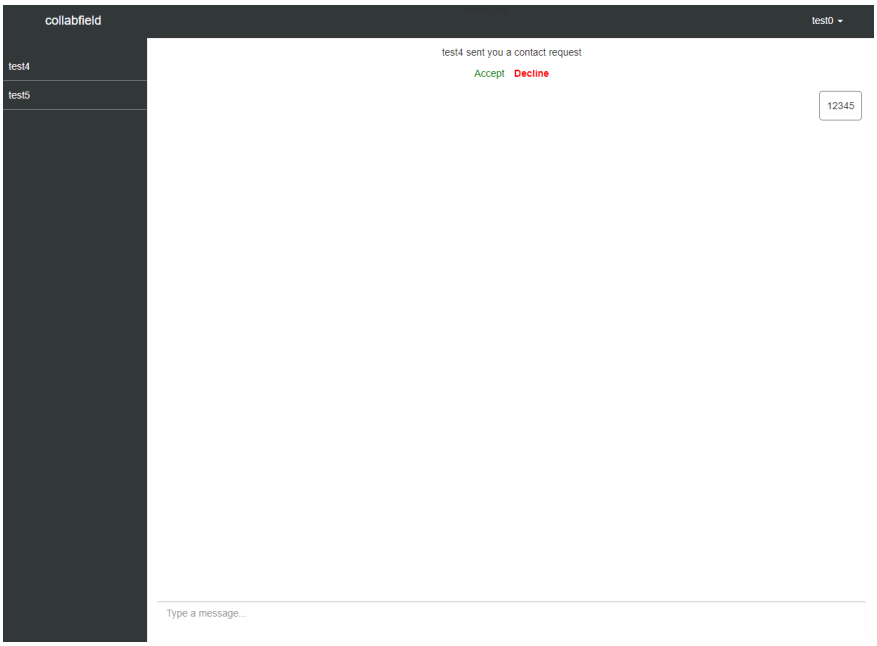


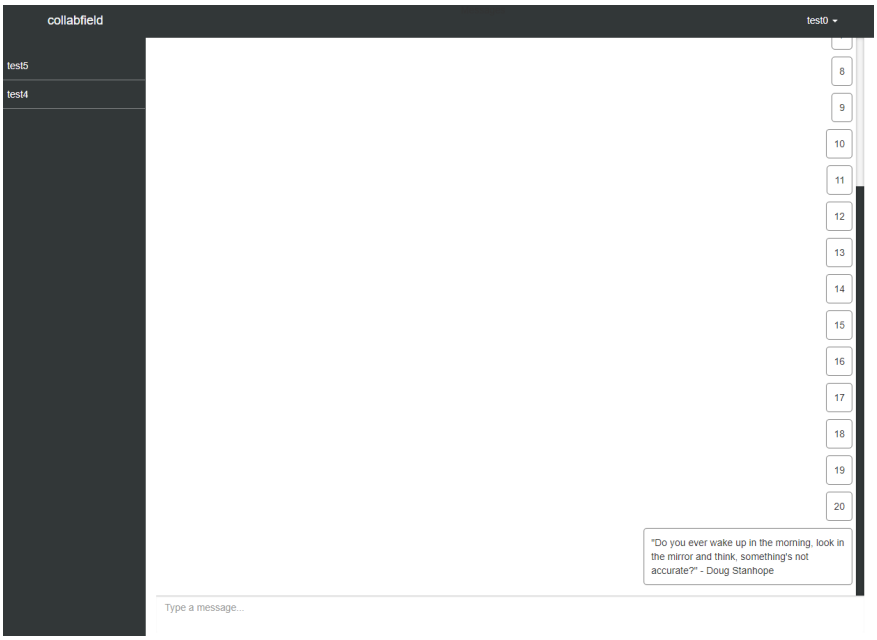
assets/javascripts/messenger.js

When you visit the `/messenger` path, you see the messenger:



Then you can open any of your conversations.



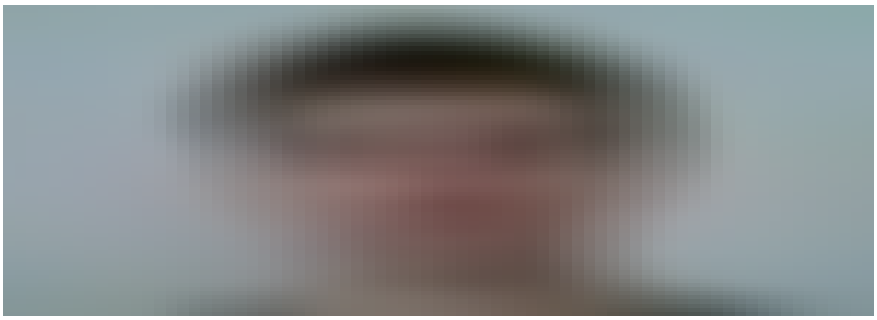


Commit the changes.

```
git add -A
git commit -m "Create a messenger.js"
```

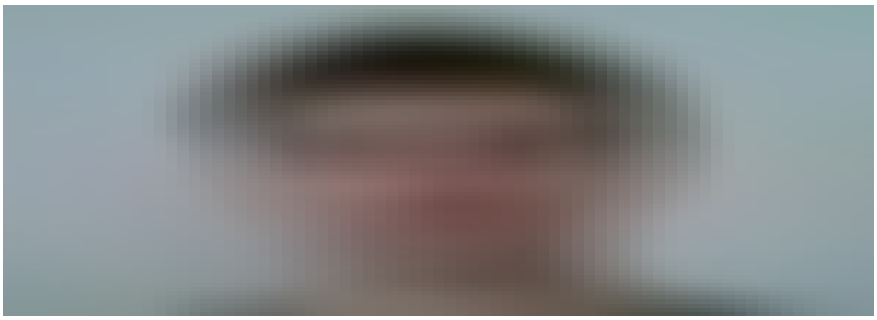
Now on smaller screens, when users click on the navigation bar's link to open a conversation, their conversation should be opened inside the messenger instead of a conversation window. To make this possible, we've to create different links for smaller screens.

Inside the navigation's `_private.html.erb` partial, which stores a link to open a private conversation, add an additional link for smaller screen devices. Add this link just below the `open_private_conversation_path` path's link in the file



layouts/navigation/header/dropdowns/conversations/_private.html.erb

On smaller screens, this link is going to be shown instead of the previous one, dedicated for bigger screens. Add an additional link to open group conversations too



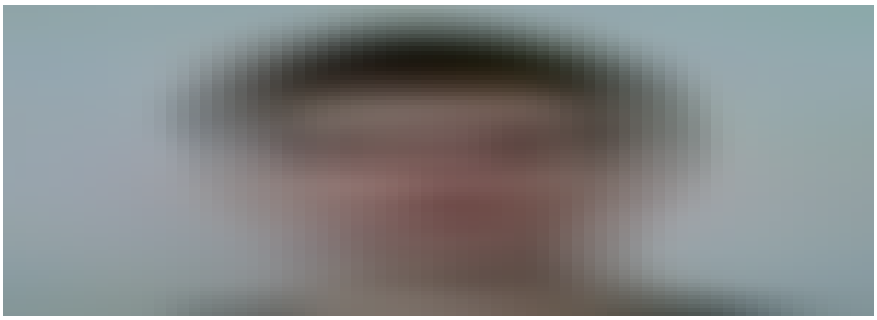
layouts/navigation/header/dropdowns/conversations/_group.html.erb

The reason why we see different links on different screen sizes is that previously we've set CSS for `bigger-screen-link` and `smaller-screen-link` classes.

Commit the changes.

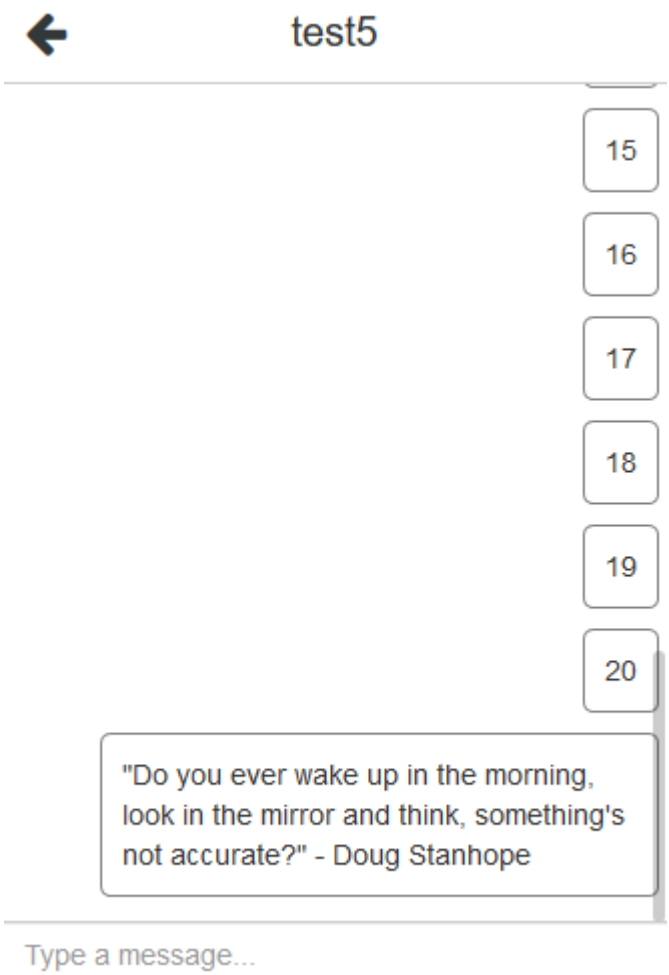
```
git add -A
git commit -m "Inside _private.html.erb and _group.html.erb,
in the
layouts/navigation/header/dropdowns/conversations, add
alternative
links for smaller devices to open conversations"
```

Messenger's versions on desktop and mobile devices are going to differ a little bit. Write some JavaScript inside the `messenger.js`, so after a user clicks to open a conversation, js will determine to show a mobile version or not.



assets/javascripts/messenger.js

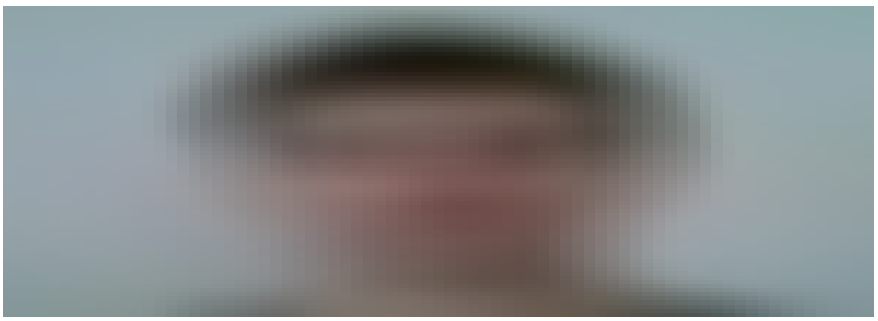
Now when you open a conversation on a mobile device, it looks like this



Commit the change.

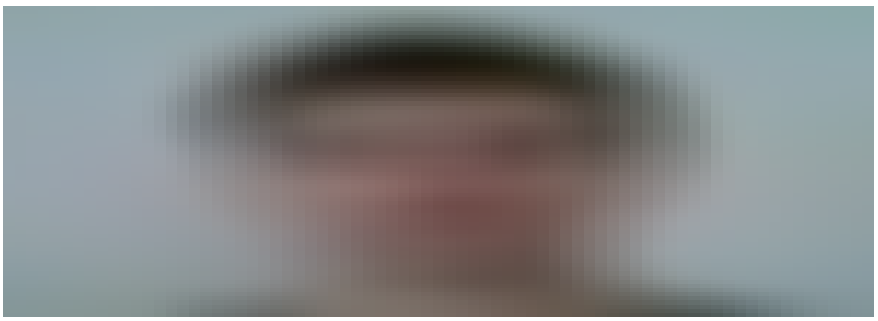
```
git add -A
git commit -m "Add JavaScript to messenger.js to show a
different
messenger's version on mobile devices"
```

Now make group conversations functional on the messenger. Majority of work with the messenger is already done, so setting up group conversations is going to be much easier. If you look back inside the `MessengersController`, we have the `get_group_conversation` action. Create a template file for it:



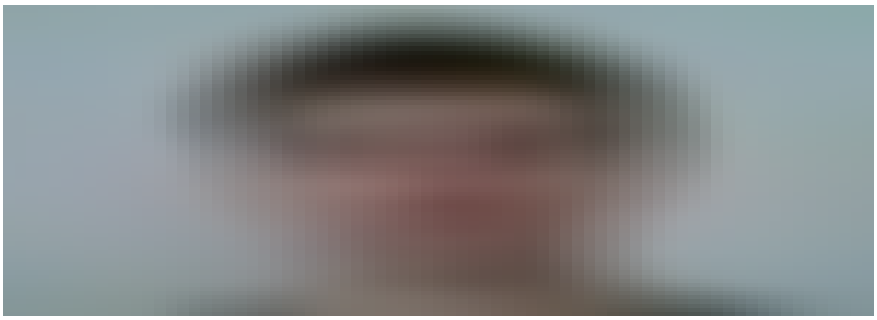
messengers/get_group_conversation.js.erb

Then create a file to render a group conversation in the messenger:

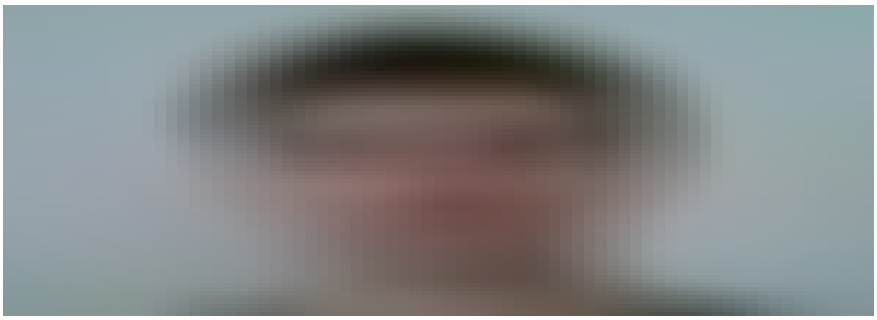


messengers/_group_conversation.html.erb

Create its partials:



messengers/group_conversation/_details.html.erb

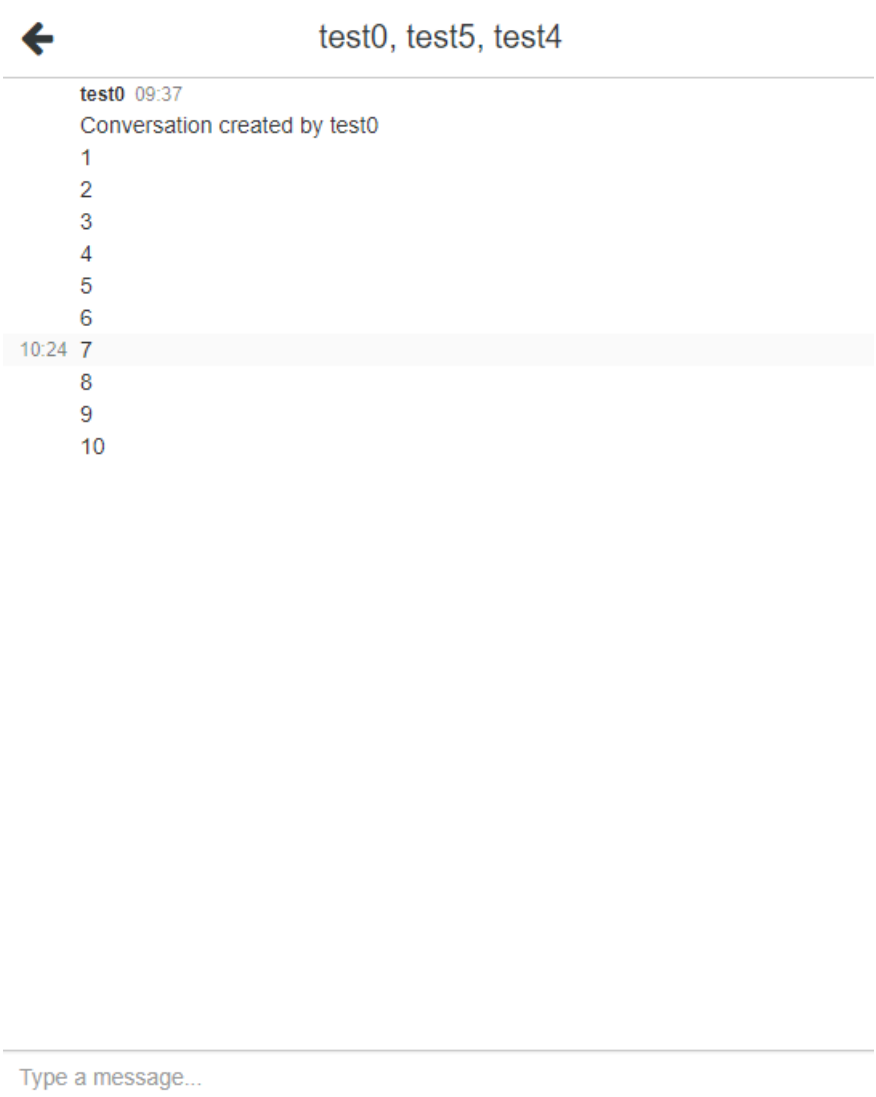


messengers/group_conversation/_new_message_form.html.erb

Commit the changes:

```
git add -A
git commit -m "Create a get_group_conversation.js.erb
template and
its partials inside the messengers"
```

That's what group conversations in the messenger look like:



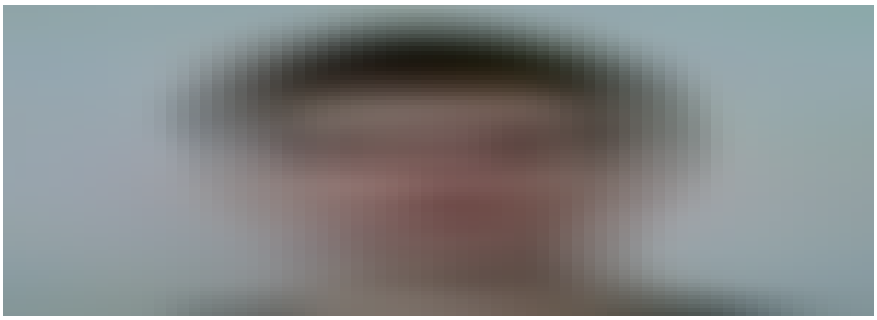
5. Notifications

The application already has all fundamental features ready. In this section we'll put our energy on enhancing those vital features. Instantaneous notifications, when other users try to get in touch with you, provide a better user experience. Let's make users aware whenever they get a contact request update or joined to a group conversation.

Contact requests

Generate a notification channel which will handle all user's notifications.

```
rails g channel notification
```



channels/notification_channel.rb

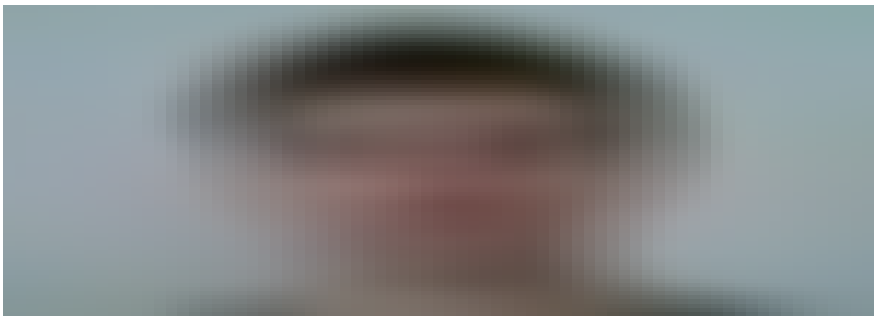
Commit the change.

```
git add -A
git commit -m "Create a NotificationChannel"
```

Every user is going to have its own unique notification channel. Then we have the `ContactRequestBroadcastJob`, which will broadcast contact requests and responses.

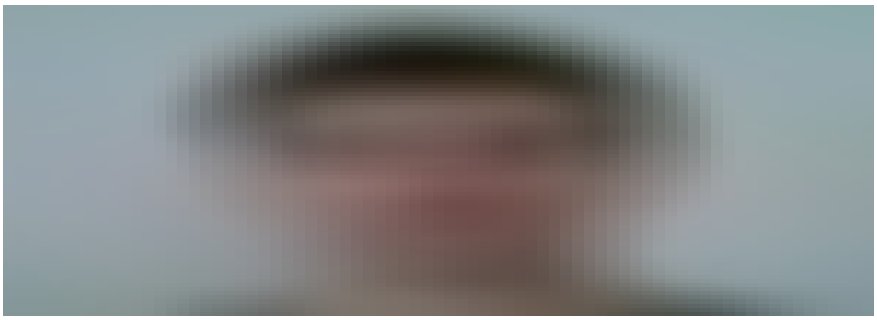
Generate the job.

```
rails g job contact_request_broadcast
```



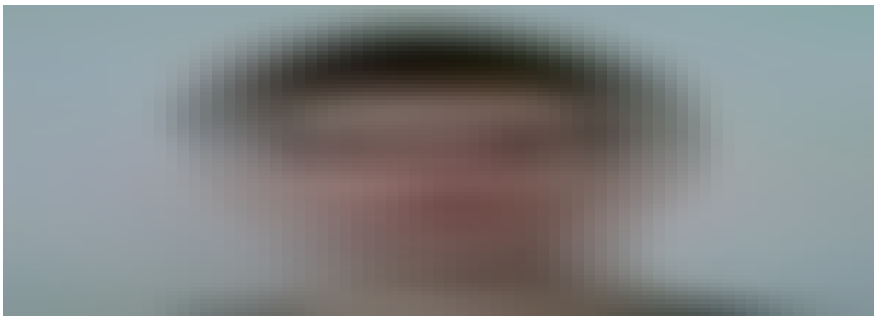
jobs/contact_request_broadcast_job.rb

Create a `_contact_request.html.erb` partial, which will be used to add contact requests in the navigation bar's drop down menu. In this case we'll add those requests dynamically with the `ContactRequestBroadcastJob`



contacts/_contact_request.html.erb

Fire the job every time a new `Contact` record is created:

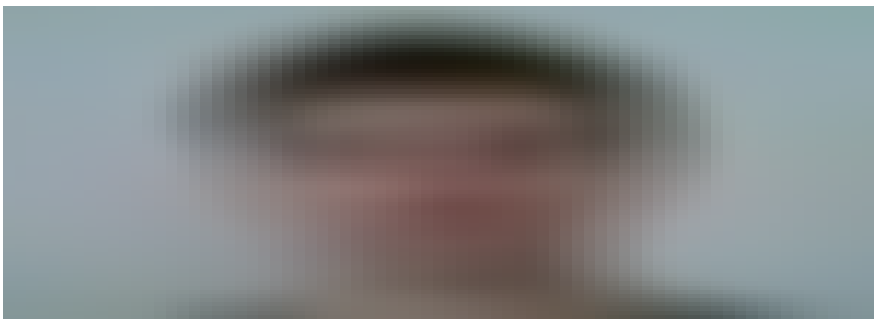


models/contact.rb

Commit the changes.

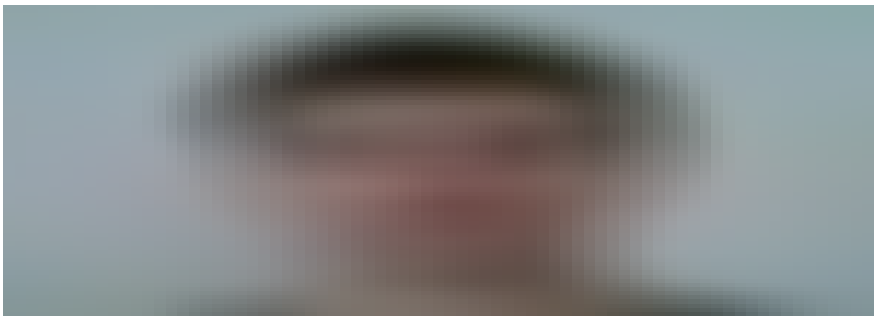
```
git add -A
git commit -m "Create a ContactRequestBroadcastJob"
```

Then create a drop down menu itself on the navigation bar:



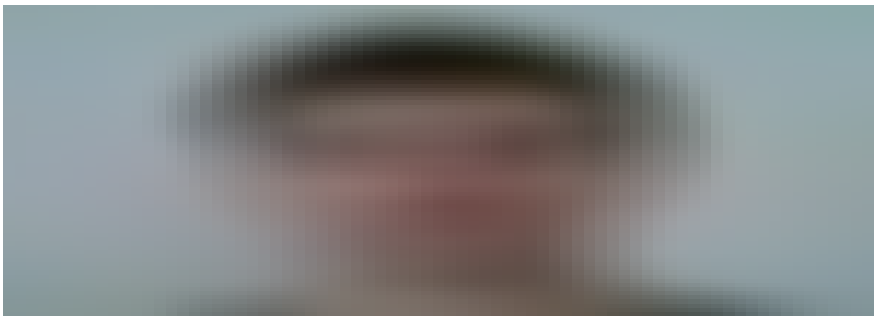
layouts/navigation/header/dropdowns/_contact_requests.html.erb

Define the `nav_contact_requests_partial_path` helper method:

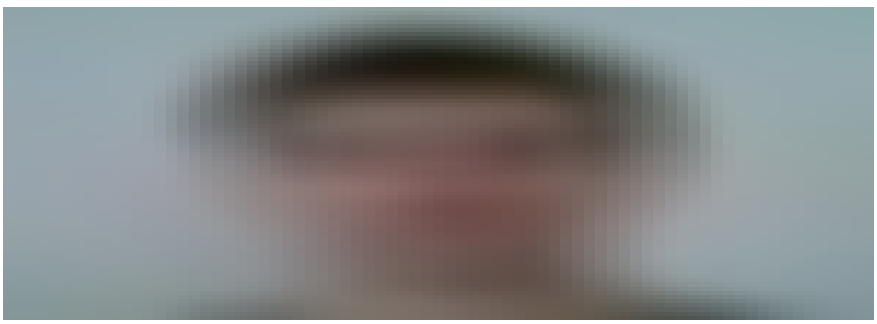


helpers/navigation_helper.rb

Wrap the method with specs and then create the partial files:

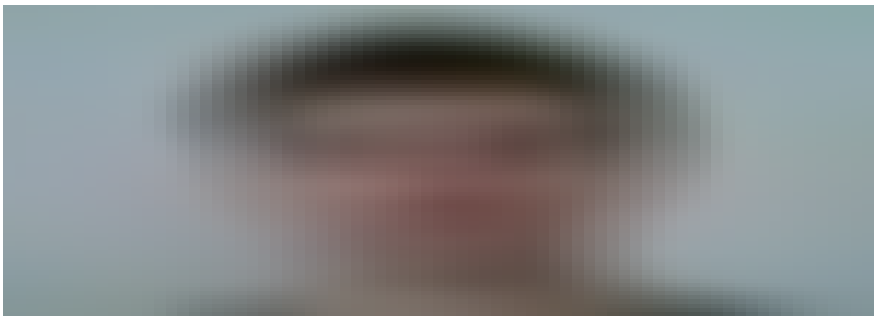


layouts/navigation/header/dropdowns/contact_requests/_requests.html.erb



layouts/navigation/header/dropdowns/contact_requests/_no_requests.html.erb

Inside the `_dropdowns.html.erb` file, render the `_contact_requests.html.erb`, just below the conversations. So we could see a drop down menu of contacts' requests on the navigation bar

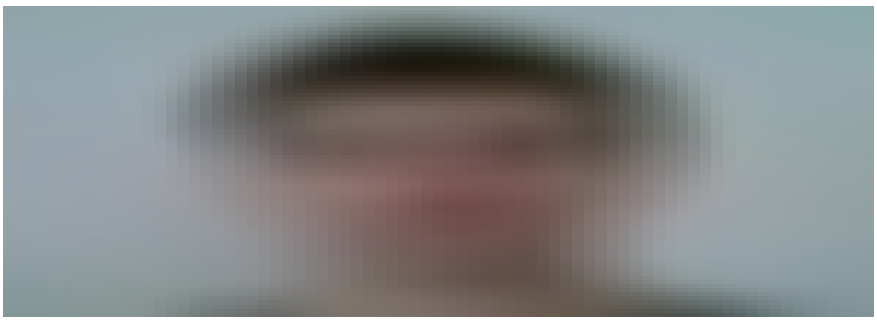


layouts/navigation/header/_dropdowns.html.erb

Commit the changes.

```
git add -A
git commit -m "
- Create a _contact_requests.html.erb inside the
  layouts/navigation/header/dropdowns
- Define a nav_contact_requests_partial_path in
NavigationHelper"
```

Also create a partial file for a single contact request:

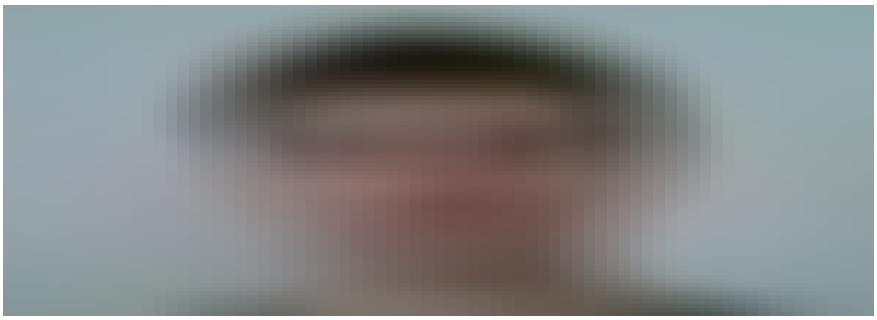


layouts/navigation/header/dropdowns/_request.html.erb

Commit the change.

```
git add -A
git commit -m "Create a _request.html.erb inside the
layouts/navigation/header/dropdowns"
```

Add CSS to style and position the contact requests' drop down menu:

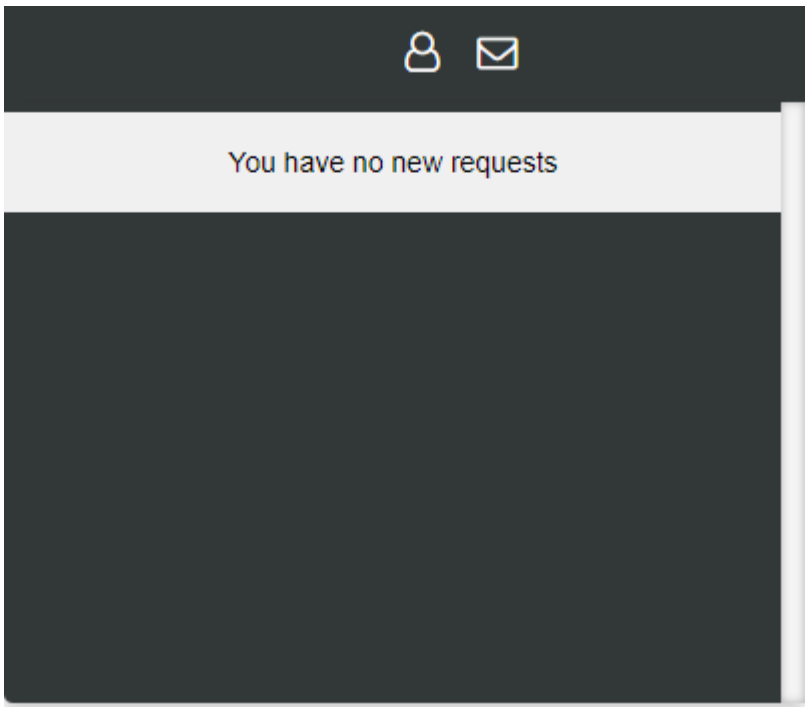


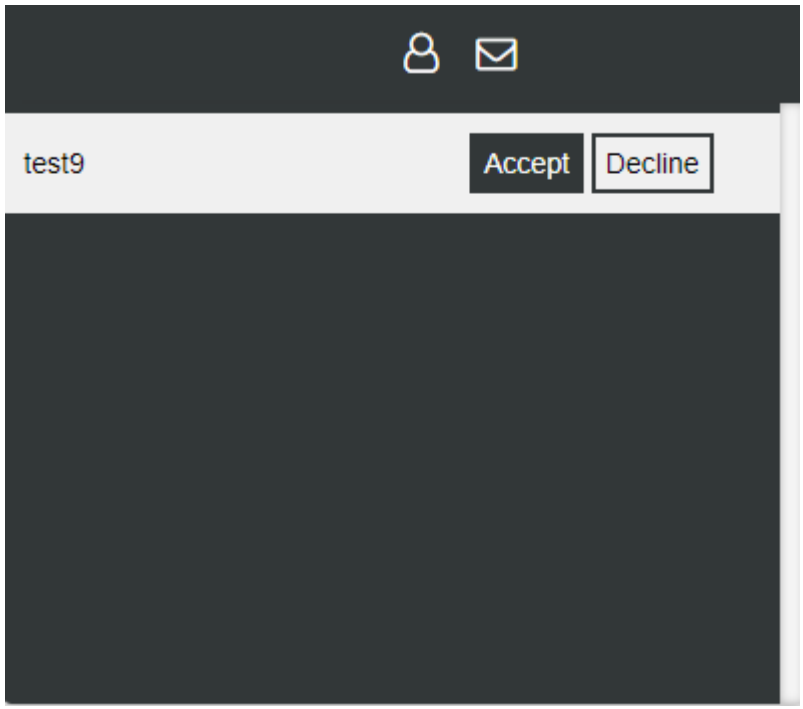
assets/stylesheets/partials/layout/navigation.scss

Commit the changes.

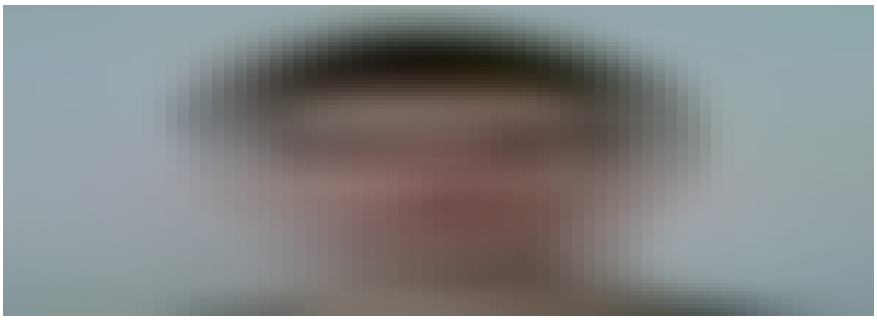
```
git add -A
git commit -m "Add CSS in navigation.scss to style and
position the
contact requests drop down menu"
```

On the navigation bar, we can see a drop down menu for contact requests now.





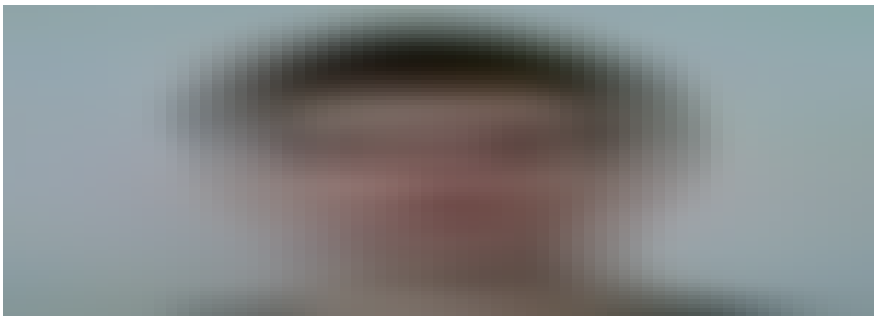
We have the notifications channel and the job to broadcast contact requests' updates. Now we need to create a connection on the client side, so users could send and receive data in real time.



assets/javascripts/channels/notification.js

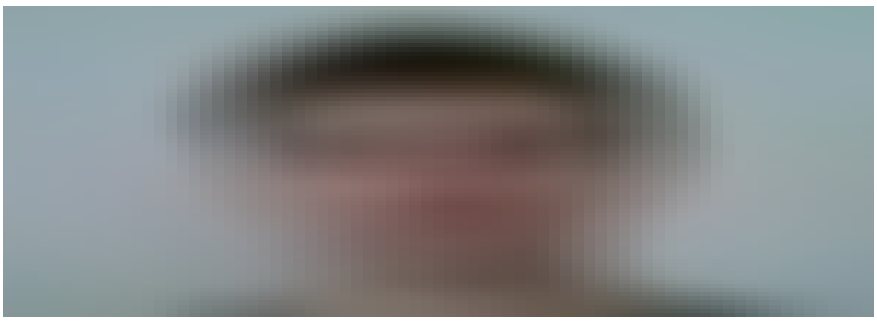
Notice that `if` statements, where a contact request was accepted and declined, have empty code blocks. You can play around and add your own code here.

Also create a `contact_requests.js` file to perform DOM changes after certain events and broadcast performed actions to the opposed user, using the `contact_request_response` callback function



assets/javascripts/contact_requests.js

Also after a new contact request is sent from a conversation window, remove the option to send the request again. Inside the conversation's `options.js` file, add the following:



assets/javascripts/conversations/options.js

Now contact requests are going to be handled in real time and the user interface will be changed after particular events.