Programming Reference for PauseLab's BeCville Application

Rails Basics

This application was built with Ruby on Rails 5; if you are unfamiliar with the Rails framework then you may get started with the tutorial available from RailsGuides (among many others). Rails is an MVC framework, so the bulk of the work goes into the `models`, `controllers`, and `views` folders.

User Roles

The application is designed around six different roles that a user can take. The roles are mutually exclusive, in that a user can only occupy one at a time.

- Admins: Site administrators who have the highest privileges, including creation (and deletion) of other users in other roles. They control and maintain the system.
- Moderators: Trusted users who are permitted to manage any data in the system unrelated to users (e.g. ideas, proposals, blog posts). They revise and filter site content.
- Steering Committee: Users who are permitted to see most data in the system, even if it not accessible to residents (see below), and are able to use this data to communicate with each other. They direct PauseLab's focus and draw insights from the community.
- Artists: Users who propose public art projects to address community needs.
- Super Artists: A subset of artists who are selected to implement projects and report on their progress. They carry out the will of the community.
- Residents: Users to view the system's published content and may submit ideas for improvements to their area. They are community representatives.

Models

This system has a number of models implemented through a relational database schema.

- Idea: Submitted by residents in the first phase of the application. Contains freeform text about the idea itself, a category that the idea belongs to, a geographic location (i.e. latitude and longitude), and some personal information about the resident submitting the idea, including name, email, and phone number. When an idea is first submitted, it has an "unchecked" status, and is not publicly available. Only when an admin or moderator approves the idea does it become visible.

    - Category: Created by moderators as a means of grouping ideas together. Contains a short piece of text. Some example categories include: “Health and Safety,” “Art and Culture,” “Infrastructure,” and “Other.”
- Proposal: Submitted by artists in the second phase of the application. Contains freeform text about the proposal itself, a title, a link to the artist’s portfolio. Similar to ideas, proposals must be approved by admins or moderators before being publicly visible. Proposals may also be marked as “funded” to indicate to users which ones were selected in the voting phase.
    - Proposal Budget: Contains cost fields for various parts of the plan. This encourages artists to think about how they will allocate their funds if their project is selected.
    - Proposal Comment: Contains text for internal comments between members of the steering committee regarding a proposal.
- Vote: Submitted by residents during the last phase of the application. Has a many-to-many relationship with proposals, as well as personal information on the voter such as name, email, and phone number.
- Blog: Created by super artists or moderators during any phase of the application, but particularly after voting is complete, to report any news about PauseLab. Contains a title and freeform text body that may be embedded with formatting, images, or hyperlinks.
- Iteration: Captures the notion of cycles of the application. PauseLab can conceivably operate the ideas-proposals-voting process for community projects at least once per year, so each time it needs to reset it creates a new iteration. Each iteration has many ideas, proposals, and votes, and when each of these three models is shown on the site they are enumerated according to the current iteration.
- Landing page: Created by moderators for specific but mutable page content. Contains freeform text that may be embedded with formatting, images, or hyperlinks. Landing pages are exposed on the About page, the Ideas collection home page, and the pages that artists and steering committee are redirected to once they log in.
- Mass Email: Created by moderators for sending emails to certain role groups. For example, a moderator may want to send out an email to all artists reminding them to send in proposals before that phase of the application is over.
- User: Contains the profile of a users who authenticate into the system, including name, email, password, phone number, role, and an avatar image. Users may be associated with proposals, proposal comments, and blogs. However, they are not associated with ideas or votes because those submissions would typically be made by “one-time” users; the personal information collected in those submissions reside in those models themselves.
- Ability: Defines the permissions for each user role. Roles generally fall into a hierarchy where those further down the chain possess a subset of permissions belonging to those higher up. Admins are the top, followed by moderators, steering committee, super artists, artists, and residents, in that order.
    - Admins: Can manage (i.e. create, read, update, destroy (CRUD)) any type of data
    - Moderators: Like admin, but cannot manage user data
    - Steering Committee: Can create proposals, proposal comments, ideas, votes, and manage anything that they created. Can read all ideas, regardless of approval status. Can only read approved proposals.
    - Super artists: Like steering committee, but can create blogs, and cannot create proposal comments. Can only read approved ideas and proposals.
    - Artists: Like super artists, but cannot create blogs.
    - Residents: Like artists, but cannot create proposals.

EXTENDING THE SYSTEM

Customizing User Roles and Permissions

User roles are defined in an enum `role` in the `User` model. To create a new role, add a new symbol to the enum array. You will also need to add a translation for that role (i.e. what the name of that role looks like on the site) in `config/locales/en.yml` under `en.activerecord.attributes.user.roles.<your_new_role>`. Now that new role will show up in an admin's create/edit user prompt.

To change or add permissions for a user role, edit the `Ability` model. In the `initialize` method, you will find a set of conditional branches for each user role. Within each branch are a set of statements regarding that role's permissions (we use CanCanCan, which has a document on defining abilities on their GitHub repository). See the existing code for a reference on current roles' permissions. If no `can?` statement explicitly defines what a role can do, then by default it will not have that permission.

In some cases, CanCanCan may not be enough for defining permissions. For example, it cannot do per-field authorization, which would be useful for the Idea model, which has fields like `first_name` and `phone` that should only be shown to admins and not to residents. We get around this with Devise -- in `ApplicationController`, we define helper methods to check if the user is under a certain set of roles (e.g. `user_has_admin_access`), which we can then use in controller or template code to filter certain fields out or show certain elements on a page.

Creating New Phases and Changing the Homepage

Phases are defined in an enum `status` in the `Iteration` model. To create a new phase, add a new symbol to the enum array. You will also need to add a translation for that role (i.e. what the name of the phase looks like on the site) in `config/locales/en.yml` under `en.activerecord.attributes.iteration.statuses.<your_new_status>`. Now the new phase will show up in an admin's change iteration page.

Functionally, phases only affect the redirection path of the root page, how the navbar looks, and permissions. To change/add a new redirection path, edit the `go_home` method in `PagesController` -- change/add the switch statement there. To change how the navbar looks, add conditional statements that reference `Iteration.get_current.status` (Rails has handy methods for checking if the status is a certain value, e.g. `Iteration.get_current.ideas?` will return true if `status` is `ideas`). To change permissions, edit the `Ability` model using CanCanCan code.

Creating Landing Pages

Landing pages are the editable wysiwyg content blocks that admins and moderators use to help users navigate the site and give information about PauseLab. The "About Us" and the homepage are examples of pages that have an editable landing page that only admins and moderators can edit. To

add a new landing page put the below code in the view, and change the title to the new landing page's title.

```
<% if Landingpage.exists?(title: "About Us") %>
 <% @landingpage = Landingpage.where(title: "About Us").first%>
 <h2><%= @landingpage.title%> </h2>
 <p><%=  @landingpage.description.html_safe %> </p>
  <% if can? :edit, Landingpage %>
 <p> <%= link_to (t 'pages.edit_about'), edit_landingpage_path(@landingpage) %></p>
 <% end %>
 <% else %>
 <p><%= t 'pages.no_post' %></p>
 <% if can? :create, Landingpage %>
  <p> <%= link_to (t 'pages.create_about'), new_landingpage_path %></p>
<% end %>
<% end %>
```

In views/landingpages/_form.html.erb add the new title to the list of options in the dropdown menu. In landingpages_controller.rb change the update method to include a correct redirection to the correct route.

Adding New Fields to the Idea Submission Page, Proposal Submission Page, or Voting Page

Summary:

To add new fields to the _forms found in the *views/* folder (*views/ideas/* , *views/proposals/ views/votes/*), you must go through Ruby's MVC. The three simple steps are:

1. Add new columns to the table you wish to edit via rails migrations
2. Add new fields to the private functions found in the ideas, proposals, or votes controller
3. Add new fields to the _form found in the respective *views/* folder

Example:

Let's say we wanted to add the new field **Address** to the voting form to confirm voter identification.

Step 1: Add new columns to the table you wish to edit via rails migrations
Make a new migration in the console

```
$ bin/rails generate migration AddAddressToVotes
```

Add the new column Address to the new file in db/migrate

```
class AddAddressToVotes < ActiveRecord::Migration[5.0]
 def change
   add_column :votes, :address, :string
 end
end
```

Rake the migration in the console

```
$ rake db:migrate
```

Step 2: Add new fields to the private functions found in the ideas, proposals, or votes controller

In our current example, we would add **Address** to the private votes_controllers.rb method

```
private
def vote_params
  params.require(:vote).permit(
  :first_name,
  :last_name,
  :phone,
  :email,
  :address,
  :proposal_ids => []
)
end
```

Step 3: Add new fields to the _form found in the respective views/ folder

```
<%= simple_form_for @vote, :url => { :action => "create" } do |vote| %>
 <h2 class="text-lg-left"><%= t 'votes.personal_information' %></h2>
 <%= vote.input :first_name, placeholder: (t 'common.first_name_placeholder') %>
 <%= vote.input :last_name, placeholder: (t 'common.last_name_placeholder') %>
 <%= vote.input :phone, placeholder: (t 'common.phone_placeholder') %>
 <%= vote.input :email, placeholder: (t 'common.email_placeholder') %>
 <%= vote.input :address, placeholder: (t 'common.address_placeholder') %>
```

Configuring the Map During Idea Collection

The map displayed on the homepage during the idea collection phase is highly configurable and can be extended to support geolocation. Since the map exists solely on the client side, it was all written in javascript and the code can be found in `pages.coffee` file in the assets directory.

The `**initialize**` method is responsible for pulling all ideas and their respective categories from the database and then loading the google maps object from the maps sdk.

The `**showMap**` function, which accepts the ideas and categories as arguments is responsible for creating a google maps object, this is where we can configure the map that will be displayed to the user. We can configure the map by passing in options relating to the center coordinate, displayed controls, zoom levels, whether the icons are clickable etc.

```coffeescript
  map = new (google.maps.Map)(document.getElementById('map'),
    zoom: 10
    center: g.sw
    clickableIcons: false
    mapTypeControl: false
    streetViewControl: false
  )
```

The `**fillMap**` function is responsible for creating markers for each idea received from the database, attaching a info box to each of the pins, and a click listener to open the info box when a pin is clicked. Finally, the map is populated with all of the created pins.

We adjust the bounds of maps relative to the all of the locations of the ideas such that the maps will essentially show all of the pins by default. This is done by taking the union of the bounds of each

idea location. An alternative would be to find an idea closest to the visitor's location and show nearby ideas.

The `**buildInfo**` function builds the info box for the give idea. This involves building a giant string composed of the idea's attributes and attaching it to a `div` element. This function is also responsible for creating social share buttons for facebook and twitter that will share a sample of the idea description. This is done by opening a new window with special urls that take in a string parameter consisting of the content to be shared, this can be configured by changing the following click listeners.

```javascript

  $(fb_btn).click ->
    window.open "https://www.facebook.com/sharer.php?u=" + window.location.host + '/ideas/' + idea.id, 'share to facebook', 'height=350,width=500'

  $(twtr_btn).click ->
    desc = encodeURIComponent Pages.format(idea)
    window.open "https://twitter.com/intent/tweet?text=" + desc, 'name', 'height=300,width=500'

```

Creating Event-Triggered Emails

If you would like to send out an email when a certain event occurs, such as when a new user is created or when a certain form is submitted. There are already Mailers in place that make performing such an action simple with only a few lines of code.

Step 1: Once you know what you would like to send out, add the strings to the specific language files, making sure to have a string for the body and the subject line. So, in the config/en.yml, you may add the following strings to a subsection user, with any outside parameters surrounded by "%{}" with the entire string containing the outside parameter surrounded in quotes ("").

```
users:
        confirm_subject: Your Account was Successfully Created
        confrm_body: "Thanks %{name} for joining our site!"
```

Step 2: Locate the controller file for the action you are trying to use as the trigger event for your email. For example, if you would like to send an email out when a new user is added, locate the controller file where the new user is successfully saved to the database.

Step 3: Located where the trigger action happens, in this case where the user is successfully added to the database, use the following lines as a template for what to add:

```
if user.save
        @to = user.email
        @name = user.first_name
        SlpMailer.email_custom_text(@to, (t 'users.confrim_subject', :name => @name), (t
'users.confrim_body', :name => @name)).deliver
end
```

In the above code, the first parameter of email_custom_text() is the email address being sent to, this can be stored in a string variable prior to being passed in. The subsequent parameter takes in a string containing the subject string and the last parameter takes in the body string.

Once you have completed the above steps, the email should be sent out whenever the specified event is triggered.

Adding New Translations

Localized strings are in the `config/locales` folder. Those in English are in `en.yml`. If you would like to add another language, create another .yml file whose name is the 2 letter country code (e.g. `es.yml` for Spanish). Then copy all the mappings from the English file into the new language file and replace the values with the appropriate translations. Make sure the top level key contains the right language code (e.g. `es` instead of `en`).

The app uses third party gems which use strings that need to be localized (e.g. `devise` and `simple_form`) -- these are kept in separate files. The appropriate translations will need to be included in the `config/locales` folder as well, but through some searching online you can find others who have already done that work.

Afterwards, in `config/routes.rb`, on the line that says

```
  scope "(:locale)", locale: /en/ do
```

Add the new language code in the section enclosed by forward slashes separated by a vertical pipe character. For example, to add Spanish, that line would be changed to

```
  scope "(:locale)", locale: /en|es/ do
```

Then, in `app/views/shared/_navbar.html.erb`, in the section containing the locale switcher (CTRL+F for `:locale`), add an element for the new language (if only one other language, it should be a single link; otherwise, you will likely want a multi-select element).

Adding a WYSIWYG Editor

The WYSIWYG editor allows for text fields to be replaced by a box that can format text. To use this, make sure that the editor's javascripts and stylesheets are properly located. Those will be found within assets/javascripts/application.js and assets/stylesheets/application.scss. In addition, make sure that the gem is installed as well.

Within the views, you'll want to add the following Javascript.

```
<script>
$('textarea').each(function() {
 $(this).trumbowyg({
     svgPath: '<%= image_path("icons.svg") %>',
     btns: [
         ['formatting'],
         ['insertImage'],
```

```
            'btnGrp-justify',
            'btnGrp-lists',
            ['horizontalRule'],
            ['fullscreen']
        ]
    });
});
</script>
```