Project - Online Forum

Description

This application provides a forum for users to make posts and to comment on each other's posts.

My Configuration Details

Operating System

OS X 10.15.7

Browser Used

Chrome Version 104.0.5112.79 (Official Build) (x86_64)

Version of Ruby

ruby 3.1.2p20 (2022-04-12 revision 4491bb740a) [x86_64-darwin19]

Version of PostgreSQL

psql (PostgreSQL) 14.4

Secondary Configuration Details

After completion of development this was also tested on a system with:

- Windows 11 / Ubuntu 20.04.4 via WSL 2
- Microsoft Edge 104.0.1293.63 (Official build) (64-bit)
- ruby 3.1.2p20 (2022-04-12 revision 4491bb740a) [x86_64-linux]
- (PostgreSQL) 12.12 (Ubuntu 12.12-0ubuntu0.20.04.1)

Installation | Configuration | Running the Application

Installation

Unzip the archive, change into the root directory and run the following command to make the shell files executable:

```
chmod +x *.sh
```

(alternately the shell files can be run without making them executable via `sh <filename>.sh`)

Configuration

The application can be installed on a POSIX-compliant system (Linux, Mac OS X, etc.) by running `./setup.sh forum`, which will `bundle install` the necessary gems, create the `forum` PostgreSQL database, and then seed the database with some sample data.

NOTE: If you wanted to configure the application to start with no users, comments, or posts, you could run `./setup_empty.sh forum` instead.

Running tests

To set up the database for testing, run `./setup.sh forum_test`. This will create a `forum_test` PostgreSQL database, and seed it with some sample data.

Then tests can be run with `./test.sh`.

Running the Application

- The application can be run with `./run.sh` (or `bundle exec ruby forum_app.rb`)
- The application runs on port `8889` and listens on all interfaces.
- All users have the password `whatsup`, the recommended user is `admin`.
- Navigate back to the home page by clicking the "Online Forum" link at the top-left of the page.

Code Structure

Code Organization

- The main application code is in `forum_app.rb`.
- Helper functions to be used in views are in `view_helper_functions.rb`, and helper functions used in routes are in `route_helper_functions.rb`.
- Code to interface with the database is in `database_persistence.rb`.
- SQL code to create the database is in the `db` folder:
  - `schema.sql` contains SQL to create the database and populate it with sample data.
  - `schema_no_data.sql` contains SQL to create the database but not populate it with sample data.
  - `seed_data.sql` contains SQL to populate the database with sample data.
- Code to test the application functionality is in `test/forum_test.rb`.
- ERB files are in the `views` folder.
- The `public` folder contains the static files for the application.
  - The `stylesheets` subfolder contains the stylesheet `application.css`
  - The `javascripts` subfolder contains the JavaScript files `application.js` and `jquery-3.6.0.js`.
  - The `icons` subfolder contains the icon assets created for this application

Requirements

The application must use at least two kinds of related data where one of the data types is a collection of objects of the other type.

In this forum application, users can make posts and comment on each other's posts. Each post is in a one-to-many relationship with a set of comments.

The application must provide CRUD capabilities (create-read-update-delete). In other words, the user should be able to use the application to create, display, update, and delete collections and objects. All changes (creation, updates, and deletes) should be reflected in the database.

Create

Posts can be created via a `POST` to the `/posts/new` route. Similarly comments can be created via a `POST` to the `/comments/new` route.

The form to create a new posts is viewable at `/posts/new`. The form to create a new comment for the post with id `:post_id` is viewable at the bottom of the page for that post at `/posts/:post_id`

Read

The index page displays a list of posts, and each row has the avatar and name of the user who made the post, the post title, total comments on that post, and how long ago there was activity on that post.

An individual post can be viewed at the `/posts/:post_id` route. Similarly, an individual comment can be viewed at the `/posts/:post_id/comments/:comment_id` route (though this view is not linked to from any pages in the application).

Update

Posts can be updated via a `POST` to the `/posts/:post_id/edit` route. Similarly, comments can be updated via a `POST` to the `/posts/:post_id/comments/:comment_id/edit` route.

The pages to update a post are viewable at `/posts/:post_id/edit`. The pages to update a comment are viewable at `/posts/:post_id/comments/:comment_id/edit`.

Delete

Posts can be deleted via a `POST` to the `/posts/:post_id/delete` route. Similarly, comments can be deleted via a `POST` to the `/posts/:post_id/comments/:comment_id/delete` route.

Changes reflected in the database

The `DatabasePersistence` class, which is found in the `database_persistence.rb` file, is responsible for persisting changes to a PostgreSQL database named `forum`. An instance of this class is made available to the routes via the `@storage` variable.

Comments are created and updated via the `DatabasePersistence#create_comment` and `DatabasePersistence#update_comment` methods in `DatabasePersistence`. Similarly, posts are created and updated via the `DatabasePersistence#create_post` and `DatabasePersistence#update_post` methods.

Comments are deleted via the `DatabasePersistence#delete_comment` method, and posts are deleted via the `DatabasePersistence#delete_post` method.

The page used to update a collection or object must have a unique URL.

As stated above, the page to update a post is viewable at `/posts/:post_id/edit`, and the page to update a comment is viewable at `/posts/:post_id/comments/:comment_id/edit`.

When listing collections and objects, limit the amount of output per page to a maximum item count. If there are more than this number of items, the user should be able to scroll through the data in chunks of the maximum item count.

The number of items per page is configurable via the `MAX_ITEMS_PER_PAGE` constant found on line 16 of the `forum_app.rb` file.

For pagination, say we are on page `3` of our index displaying the posts. First all of the posts are retrieved and stored in `@all_posts`. Then we select the current page of posts by slicing the array `@all_posts` to return the posts for the current page using the `select_page` helper method.

A similar process is used to retrieve the comments appropriate for a particular page of a post.

When listing collections or objects, sort the items consistently. For instance: alphabetically, numerically, by date, and so on.

Posts are sorted in descending order by the last time they had a new comment, or if there are no comments, in descending order by the time they were created. Comments are sorted in ascending order by the time they were created.

Validate input data as needed. In particular, you must prevent SQL and JavaScript injection.

On input data the following validations are performed: Post title and content are required to be non-empty strings, and the post title must be less than 100 characters. Comment content must not be empty.

JavaScript injection is prevented by setting `escape_html` to `true` for `:erb` in the `configure` block of `forum_app.rb`, so that whenever input from the user is outputted in the views through `<%= %>`, the user input will be run through `Rack::Utils.escape_html` before being outputted to the HTML page.

SQL injection is prevented by using `exec_params` to execute queries with parameters. Parameterized statements separate the query statements from the data, and use stored queries that have markers (`$1`, `$2`, etc.) to indicate where the data should be inserted. Thus the only the stored query is executed, and user inputs are treated as data, not as statements.

Error messages should be displayed on the same page where they are raised and should be specific. If the page contains multiple inputs, you should preserve any valid data that the user has already entered.

In routes where user input is submitted, it is checked for validity and if invalid, the page is re-rendered with an error message and valid inputs preserved.

Display appropriate flash error messages when the user does something incorrectly. Don't fail silently, and don't rely on built-in errors from Sinatra.

When errors are detected, flash messages are put in the session and displayed on the next page displayed. They are kept at the top of the page until the page is reloaded or the user navigates away to another page. The error text attempts to be informative (`"That page doesn't exist"`) while not leaking implementation details (e.g., `"Page id must be an integer"`).

Validate URL parameters such as ID numbers and query strings.

In routes where a post_id is passed as a parameter, it is checked for validity with `require_valid_post_id`, and since a comment is bound to a post, both the post_id and comment_id are checked for validity with `require_valid_post_id_and_comment_id` for the `/posts/:post_id/comments/:comment_id/*` routes.

Page numbers are passed in as query strings.

The application must require login authentication. Suppose a user enters a URL for the application without logging in. In that case, you should first require authentication, then proceed to the requested page.

To implement retaining the page the user was on when they logged out, I used the session hash to store the path to the resource they were attempting to access, plus the query string (which tracks the page they were on if any, and whether they were viewing the results of a search). They are then redirected to the signin page.

If the user is authenticated, then logic in the signin page redirects them to the original page they had requested. The default layout.erb has a view helper function `clear_after_signin_session_variables` that clears the session variables that were used to track the page the user was on when they logged out. To make this work I had to make a custom layout without this helper function call just for the signin page.

It may be better to move this into an after filter which matches the signin page.

Design Choices

Navigate to Home

The homepage can be reached by clicking on a simple "Online Forum" link in the upper left corner of the page. This could be styled better in the future.

Signed in As / Sign Out

The current user who is signed in + a "Sign Out" button is displayed in the lower left corner of the page. In the future, this could be moved into the upper right corner of the page.

Sign Up

The 'signup' page is displayed if the user is not logged in and clicks the "Sign Up" button on the index page where they are prompted to sign in. Currently this page will display errors if the user tries to use an existing

username, their username is too short or too long, etc., but it would be better to have these guidelines displayed on the page.

Update Permissions

The application only allows users to update or delete their own posts and comments, so the "edit" and "delete" buttons appear only on posts or comments where the current user has that permission. Admin functionality was not implemented.

Sorting

Posts

The posts are sorted in descending order by: last comment time, or if there are no comments, the time the post was created. This allows the most recent posts to be displayed first.

I wasn't successful in implementing this sort in SQL: my attempts to do so can be found in the `scratch` directory in `posts_by_comment_date.sql` I was getting duplicates and ran into the limits of my SQL understanding.

So I returned all the posts with an additional field `last_comment_activity` generated by a subquery which found the most recent comment time if there was one, or returned the earliest possible UNIX timestamp (`to_timestamp(0)`) if there were no comments. I could then sort these posts with a Ruby helper function which compared posts either by the `last_comment_activity` if it was more recent than the post creation time, and otherwise by the post creation time.

Comments

The comments are sorted in ascending order by the time they were created. This allows the reader to follow the chronological order of the comments, which makes sense if people are replying to previous comments.

Replying to a Post

On a post page, the user can click "Reply" on the post or any of the comments and this will scroll the page down to the reply form. I kept the reply form integrated with the post page instead of having a separate page for it so that the user could still scroll up and reference what they were replying to.

User Pages

I added a user page which allows one to view the posts the user has made, this is viewable at `/users/:user_id`. Comments the user has made do not appear here, that would perhaps be something to add in the future.

N+1 queries

I was having issues with the application making two queries per user per post when loading the index page. This is because (1) the index page loads all of the posts which contain a nested user object that is looked up, and (2) for each post it fetches the user who made the post to extract their initials and display with the post title. So `DatabasePersistence#find_user_by_id` was being called twice for each post and this was initiating a `SELECT * FROM users WHERE id = ?` query for each post.

I dealt with this by caching the users in a `@users` hash, which is a hash mapping user id to user hash objects, and then looking up the user in the hash when needed.

Similarly when retrieving all posts, I used a `COUNT` subquery to count the number of comments for each post and stored that as a value in the `comment_count` field of the post hash.

Search

Search as implemented only searches the title and body of the posts. It does not search the comments.

Paths not Taken

Admin User Functionality

I have a user with id 1 and username admin, but currently there's no functionality implemented for this user to have admin privileges. In theory the admin should have an interface where the edit or delete buttons are present for all posts and comments.

Markdown Support in Posts and Comments

I had wanted to allow the users of the application to enter markdown text for their posts and comments, but I decided to go with plain text. This is because of the requirement to not use any additional Rubygems if they provide the application-specific requirements.

Right now I'm using `Rack::Utils.escape_html` to escape any user input by default when I'm outputting with `<%=` in my erb view templates (since I set `:erb` to `{:escape_html => true}` in the `configure` block). If I could use an additional gem instead of `escape_html` I would have layered the `sanitize` gem on top of markdown to html output (through the `Redcarpet` gem), because that would allow me to whitelist the html tags which would come out of the markdown parser and to still meet the requirement of preventing JavaScript injection.

Post and Comment Likes

It would have been nice to have given users the ability to like posts and comments, but this would have required setting up a many-to-many relationship between posts, comments and the users who liked them, and ensuring only one like per user per post or comment. This seemed to exceed the requirements of the project and to introduce excessive complexity.