

SE31520 Assignment 1 - Part 2: Forum for the CSA - version 1

Michał Wojciech Goly [mwg2]

30th November 2017

Contents

1	Introduction	2
2	The CSA application	2
2.1	Entity Relationship Diagram	2
2.2	The forum feature	3
2.3	Controllers Diagram	4
2.4	Models Diagram	5
2.5	Redesign for the REST API	6
2.6	REST API interface	6
2.6.1	Listing all threads	6
2.6.2	Get details about a specific thread	6
2.6.3	Creating new threads	6
3	The REST client	7
4	Cucumber testing	7
5	Flair	7
5.1	The overall look and feel	7
5.2	Docker	7
5.3	Build and extra testing	7
5.4	Heroku	8
6	Critical evaluation	8

2.2 The forum feature

Once the initial planning phase has finished, I have created a GitHub repository for the project, created work items, Dockerised the project, integrated it with Circle CI and continuously deployed the **master** branch to Heroku (I have covered the details in the *Flair* section of this document).

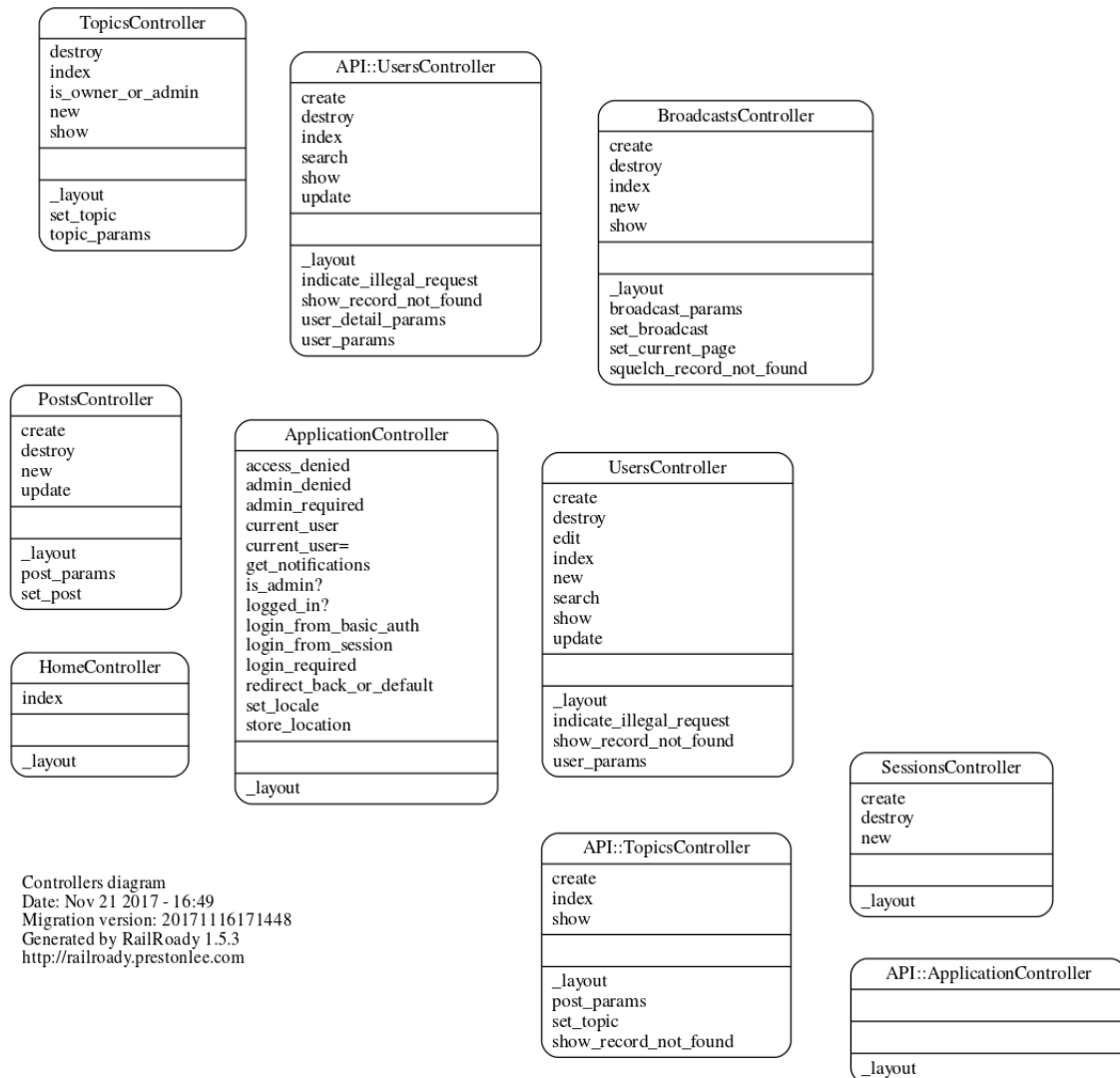
Having cleaned up the CSS and fixed the broken migrations provided (timestamps were in a wrong order), I have scaffolded the **Topic** migration, model, controller and views. I then added a *Forum* tab in the navbar and scaffolded the **Post** in a similar fashion. I wanted to generate as much as possible early on and then remove "dead code" once the forum functionality was implemented. The scaffolded code was not ideal and I had to manually define the **belongs_to** and **has_many** relations between the models. I also used self joins to make sure each **Post** could have a parent to represent posts' replies.

New topics can be created by calling the `/topics/new` route which will trigger the `TopicsController#new` action. Now, instead of creating a new **Topic** model and passing it to the view, I decided to create a **Post**, pass it to the view and use partial rendering to reuse the `/views/posts/_form.html.erb` view for both **Topic** creation and **Post** replies later on. Logically, **Topic** cannot exist on its own therefore such approach seemed optimal to minimise the amount of code written.

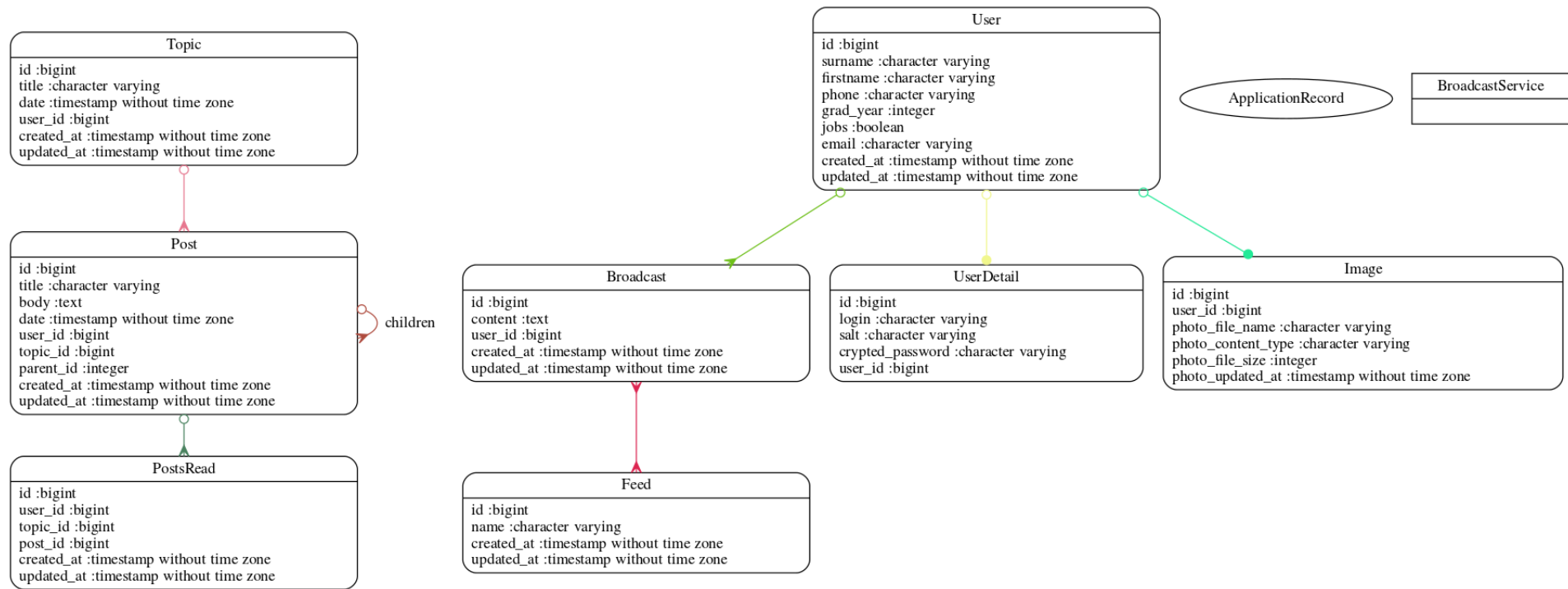
Implementation of the post replies and the associated indenting was arguably the most interesting part of the assignment. The `/views/topics/show.html.erb` view uses material design cards to render each post. The indentation was achieved using the Materialize CSS[1] grid system. The `Topic#post_wrappers` method iterates over the posts of a topic, recursively sorts them and calculates the offsets based on the amount of parents above a specific node. These wrappers are then returned as a simple array back to the view and rendered appropriately.

Finally, I have added the anonymous posting by treating `nil` valued **User** of a **Topic** or **Post** appropriately, the total count of posts for each **Topic** and the unread posts counter. This required the creation of a new table in the database with the associated model **PostsRead**. I took the simple approach of storing 3 foreign keys for a **User**, **Topic** and **Post**. This allowed me to add new records to this table each time user accessed the `/views/topics/show.html.erb` view and render the counter in the topics list.

2.3 Controllers Diagram



2.4 Models Diagram



2.5 Redesign for the REST API

In order to implement the REST API for the forum feature I have created a new controller under the api namespace called `API::TopicsController`. It contained three actions to manage incoming requests. `API::TopicsController#index`, `API::TopicsController#show` and `API::TopicsController#create` corresponded to `/api/topics`, `/api/topics/:id` and `/api/topics` routes respectively. Furthermore, I have removed the `format.json` calls from the `TopicsController`, created `json.jbuilder` views and updated the `config/routes.rb`. Finally, I have added integration tests to make sure my implementation was correct.

2.6 REST API interface

The REST API is using the Basic Auth and `$USERNAME` and `$PASSWORD` have to be exchanged for actual credentials in the examples below.

2.6.1 Listing all threads

GET `/api/topics`

1. Listing the 8 most recent threads in the forum.

```
curl http://localhost:3000/api/topics -u $USERNAME:$PASSWORD
```

2. Listing all threads in the forum.

```
curl http://localhost:3000/api/topics?all=true -u $USERNAME:$PASSWORD
```

2.6.2 Get details about a specific thread

GET `/api/topics/:id`

1. Get a single thread by its ID

```
curl http://localhost:3000/api/topics/40 -u $USERNAME:$PASSWORD
```

2.6.3 Creating new threads

POST `/api/topics`

1. Creating a new thread containing a single post titled "I need help" with body "What is Java?".

```
curl -d '{"post":{"title":"I need help", "body":"What is Java?"}}' -X POST
-H "Content-Type: application/json" http://localhost:3000/api/topics
-u $USERNAME:$PASSWORD
```

2. Creating the same post as an anonymous user.

```
curl -d '{"post":{"title":"I need help", "body":"What is Java?"}, "anonymous":"true"}'
-X POST -H "Content-Type: application/json" http://localhost:3000/api/topics
-u $USERNAME:$PASSWORD
```

3 The REST client

The REST client has been developed using Ruby and can be started by calling `ruby forum_rest_client.rb`. The implementation has been heavily inspired by the users REST client provided as part of the assignment. The end user has to login using one of the accounts in the system. The credentials are then used to authenticate with the REST API using Basic Authentication. User can list the first 8 threads, all threads, display a specific thread by id and create new ones using the command line interface.

4 Cucumber testing

I started by running the cucumber tests with the feature file provided and implementing missing step definitions. Implementing steps for the *Create a thread post* scenario provided as part of the assignment was a bit tricky with my implementation, as the threads table contains two additional columns: the date of the thread creation and the empty columns for the *Destroy* and *Show* links. Nevertheless, I managed to remove them from the hash returned by the view and successfully compare with the expected results without editing the scenario.

I also added the *Reply to the thread post* scenario to make sure replying to posts worked properly, and more importantly new posts were indented properly. Afterwards, I created the `features/forum-post-delete.feature` to ensure posts can only be deleted by owners or the admin. I also took the advantage of the cucumber background to give more context to all scenarios without repeating myself.

Finally, I created the `features/forum-pagination.feature` with the corresponding steps to make sure pagination behaved in the expected manner. This also involved testing if clicking a *Back* button takes user back to the expected page as required in the requirements specification of the assignment.

5 Flair

5.1 The overall look and feel

I have removed most of the CSS provided and replaced it with the MaterializeCSS[1] to make the user interface modern.

5.2 Docker

I have dockerised the CSA for both testing and development purposes. This involved removing the SQLite database provided and swapping the sqlite gem for a postgres one. I also had to fix the broken migrations provided, as the timestamps were in a wrong order and one of the tables was referenced before it even existed. To learn more investigate the `Dockerfile` and `docker-compose.yml` files.

5.3 Build and extra testing

Having dockerised the app I wanted to deploy it to production as soon as possible before even attempting to implement the forum feature. I'm a big fan of Continuous Integration

as it ensures my code can be constantly tested, deployed and enables me to perform heavy refactoring without worrying I break anything.

As mentioned before, I have created issues on GitHub with the aim of creating feature branches, running tests and making pull requests to close issues down. I have integrated my GitHub repository with Circle CI to continuously test and deploy my code to production. The `.circleci/config.yml` contains the build definition and is quite straightforward to follow. Circle CI starts by installing the docker-compose, it then builds the docker-compose network containing my dockerised CSA application and the postgres container for testing purposes. Afterwards the build agent sets up the database for testing, runs all the integration tests (these include REST API integration tests I have mentioned before), cucumber tests and sets up an SSH fingerprint to access Heroku for deployment.

When a feature-branch is pushed to GitHub all tests run to make sure code is correct. If a pull request is merged into the `master` branch on the other hand, the final step in the build runs and deploys the CSA to Heroku using the `heroku-container-registry`. Each deployment is followed by a `rake db:migrate` call to make sure the remote database schema is up to date.

5.4 Heroku

The application has been deployed to production and can be accessed here: <https://mwig2forum.herokuapp.com/>. The production postgresql database is provided as a Heroku add-on, because Heroku does not allow multi-container Docker environments to run together. Please see `/config/database.yml` to find out more.

6 Critical evaluation

- Could have added controller tests, but did not know how to mock sessions in rails 5 - Improve the look and feel of the rest of the app

References

- [1] *Materialize CSS*, materializecss.com [Online], Available: <http://materializecss.com/grid.html>, [Accessed: Nov. 22, 2017].