Group 4 ESOF 423 Portfolio

Adam Saylor, Alex Sutherland, and Ryan Pallman

Section 1: Program

Site: https://acommentapi.com/

Cypress Dashboard: Dashboard

Repo: https://github.com/aalleexxss/ESOF432

- Additional information about our program is provided in the README of the above repo.

Section 2: Teamwork

Team member 1 work: Responsible for initial user documentation, creating a UML diagram of our API, returning comments specific to a post, and rendering comments to an appropriate view. Specifically, member 1 had the backlog items:

- Make user documentation E(estimated):1 A(actual):1
- Make UML diagram E:2 A:2
- Update portfolio E:2 A:2
- Delete comment verification E:3 A:3
- Add instructions to developer documentation E:1 A:1
- Add swagger documentation to user documentation E:1 A:1

Team member 2 work: Responsible for setting up the database and swagger for use by other team members throughout the project. Team member 2 also worked on documenting the project via swagger by adding the necessary portions in the codebase. Team member 2 had the backlog items:

- Make developer documentation E:1 A:1
- Set up CI testing E:2 A:2
- Set up bug tracking E:2 A:2
- Create swagger documentation E:.33 A:.33
- Create view with JSON output E:2 A:2
- Create Like button E:4 A:2
- Validate that the comment fields are filled out E:2 A:1

Team member 3 work: Responsible for setting up and maintaining the server which our project runs on, creating the front end interface so that our API can be tested, including adding functionality for comments to be stored and deleted. I have also been responsible for connecting our database to the API and making sure we can store data in the database from the front end user input, as well as creating the initial project file. My specific backlog items have been:

- Setup server to run at all times E:1 A:2
- Make a comment box in HTML E:2 A:3
- Store a comment in the database E:4 A:4.5
- Figure out how to pull store comments: E:2 A:2
- Start to implement react: E:2 A: 3
- Finalize database tables E:2 A: 2
- Write tests with Cypress and set to run nightly E:3 A:3.5
- Add a dark mode E:1 A:2
- Optimize for mobile platforms E:3 A:4
- Modify Edit comment box to popup E:1 A:1
- Set up API on Google Cloud E:1 A:1.5

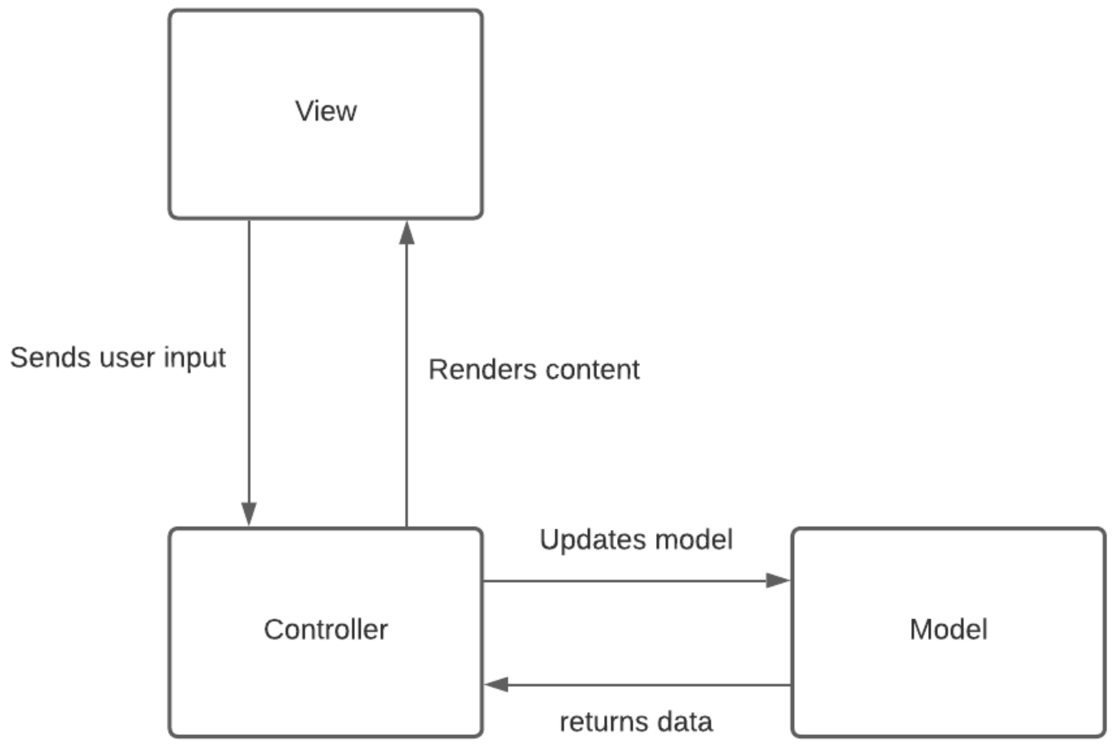
- Switching front end to ReactJS E:2 A:5

Group Backlog items:

- Setup group GitHub E:1 A:3
- Set up database locally E:3 A:1.5
- Set up routes and controller E:3 A:3
- Set up postman locally E:1 A:1
- Figure out how to store comment in the database E:4 A:4
- Add reply to comment functionality E:6 A:5
- Add comment editing functionality E:5 A:5
- Create presentation E:3 A:3

Section 3: Design Patterns

Our API makes use of the model view controller (MVC) pattern. As the name suggests, this pattern separates an application into three different parts, the model, view, and controller. The view is what the user interacts with. Input given by the user is taken by the view and passed to the controller through HTTP requests. The controller then handles the HTTP request and sends commands to the model which is responsible for database management. Data is then returned to the controller from the model and the controller updates the view. We used this pattern because it broke our API up into different parts, making it easier to address bugs and work on specific parts of the API without the clutter of the entire code base. Below is a diagram depicting this design pattern.



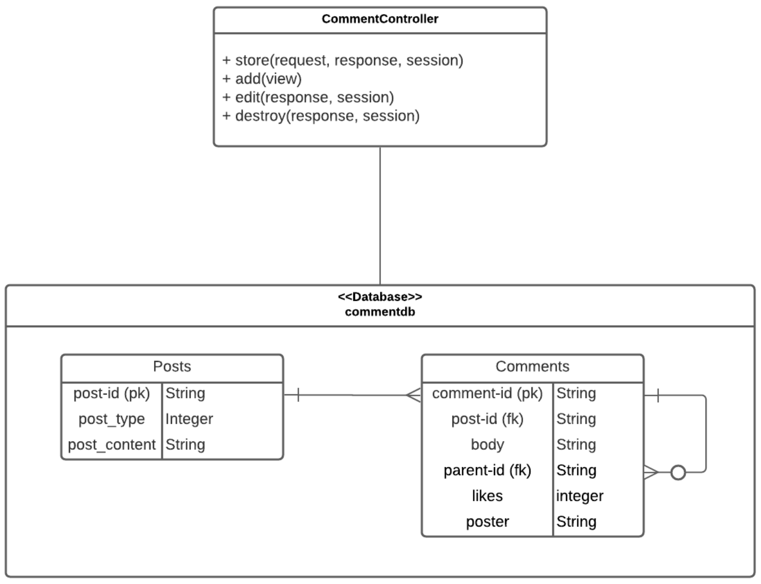
Section 4: Technical Writing

User Documentation: https://github.com/aalleexxss/ESOF432/blob/master/user-doc.md

Developer Documentation: https://github.com/aalleexxss/ESOF432/blob/master/dev-doc.md

Section 5: UML

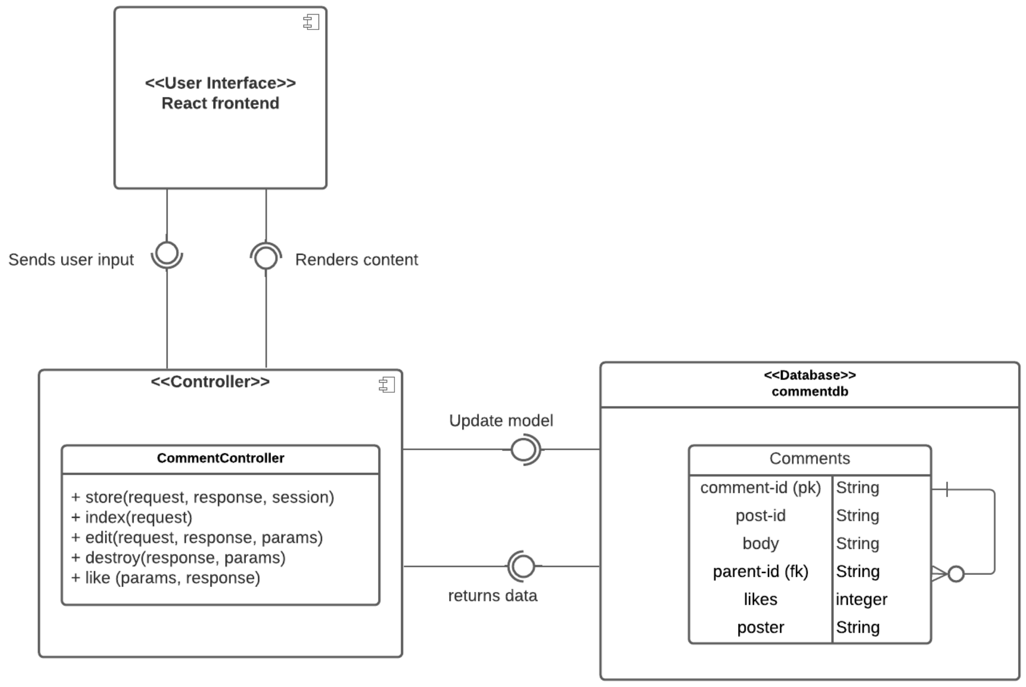
Initial design:



Design Decisions:

When designing our comment API, the biggest focus was on our database. Our databse is comprised of two tables, Posts and Comments. Each table has a generated primary key. We want our API to allow comments on images, videos, and text posts. This is why we have a post-type column in the Posts table. Because each post can have multiple comments, but each comment can only be on one post, there is a one to many relationship between posts and comments. As for the Comments, we wanted to allow comments on other comments. We agreed that "infinite subcomment" ie, a comment on a comment that's already a subcomment, can be hard to read and messy. Thus a comment can have zero or many subcomments, but subcomments can not have subcomments. Hence the one to zero or many relationship between comments. With our comment controller we wanted to be able to store comments to the database, add comments to a view, edit comments, and delete comments. The store method is responsible for taking a request, saving data to the database and sending a response. Add is responsible for putting a comment in a view. Edit and destroy each take a response and session as parameters and edit or delete a comment respectively.

Final design:



Our API went through quite a few changes through the development process. The biggest thing that changed was the implementation of a front end. Initially we were told no front end was needed for this project, however, that changed. We decided to use React to construct our front end. Another thing that changed about our design is our database. Initially, we had two separate tables, one that stores information about posts, and one that stores information about comments. Since this is primarily a backend API, specifically for sites that have posts that can be commented on, having a table that stores data about posts is redundant because the site using our API would already have that data. Therefore, we got rid of our table that holds post information. Our final design is more versatile and efficient than our initial design and more accurately depicts the model view controller pattern.

Section 6: Design trade-offs

The biggest trade-off we've dealt with in this project is the trade-off between simplicity and having a large set of features. For example, we considered having three different database tables for comments, posts, and users. This would allow a user to have our API take care of a lot of their data storage, however, this would also mean that a user that already stored said data would have to duplicate that data into our API. We decided to lean towards simplicity rather than complexity so that this API can be used in more situations.

Section 7: Software development life cycle model.

We are using agile for software development. Agile is an iterative software development methodology that focuses on continually meeting requirements, collaborating with team members, and reacting to changes in requirements. Agile development uses "scrum meetings" where each team member discusses what they are working on and what is holding them back from accomplishing their goal. Our group has scrum meetings every Monday, Wednesday, and Friday from 9:00AM to 9:50AM on WebEx. We also communicate in between scrum meetings through Discord.

Agile worked very well for our group. We were able to have three scrum meetings throughout the week. This was very beneficial because we were able to discuss what we've accomplished and what is holding us back. Additionally, discord was useful for giving group members quick updates on what we had been working on in between scrum meetings.