CS3099 Junior Honours Project

Group A1

Nina Boothby
180017218

Rhona McCracken
180017395

Matthew McIlree
180004835

Robert Nguyen Van
180005905

Scott Thomson
180016547

University of St Andrews, April 15, 2021

Abstract

The sudden transition to remote ways of working and socialising necessitated by the COVID-19 pandemic has meant that Universities have had to consider new ways of fostering a sense of “community” via digital media. In this group project, we provide a compatible instance of a federated social media application that demonstrates features useful to University communities for both social and academic purposes. The interface is in the form of a website, which follows the basic paradigm of a classic social news aggregator, such as “Reddit” or “HackerNews”, augmented with additional features that are tailored to a more academic environment.

Declaration

Contents

Chapter 1

Introduction

1.1 Problem Overview

The software engineering team project gives a broad overview of software engineering, presenting its fundamental aspects as a collaborative professional activity including its concerns and approaches. This year we have worked to apply these concepts and practices as a team.

The project briefing for this academic year (2020-21) directed us to create "a platform for distributed online communities", similar to Reddit or Hacker News, as examples, but with a focus on a university environment. Additionally, these distributed online communities had to be *federated*. This means having multiple servers communicating, where each server can host different communities, and users can access all communities via all servers in a larger supergroup.

We are part of Supergroup A, which means we communicate with the other servers, also called "instances" in this group. Through the creation of an online platform that facilitates both social and educational interaction, we looked to encompass what it means to be a university community. We interpreted a "university focused" product as being an online platform that could be used by a university for its various social and academic subcomponents. In this sense we wanted to create an application which could be used by societies, lecturers or sports clubs.

Author: 180017218

1.2 Overview of Key Features

This report discusses our project in detail, both in terms of the features we implemented and the processes we followed. We list here, in order, the aspects discussed which we consider to be of the most importance and that we are most proud of.

Key Implementation Features

1. **Federation**: Full participation in two-way communication with other groups in the Supergroup via the agreed protocol.
2. **Whiteboards**: Interactive sketch feature for collaboration.
3. **Security**: Secure user authentication, password storage and validation of other servers using digital signatures.
4. **User roles and administration**: Hierarchical site-wide and community-specific user roles and associated permissions.
5. **Post Content**: Rich post content rendering including plain text, links, images, Markdown, MathJax, and polls.
6. **Post Curation**: Multiple ways of curating posts through subscription, voting, sorting and searching.
7. **Discussion Threads**: Two-level comment threading system with top-level comments and replies.
8. **User Profiles and Settings**: Distinct user identities with generated avatars, optional user biographies and account management.

Other Key Aspects

1. **CI/CD**: Sophisticated Continuous Integration and Deployment System. Our final site is fully deployed and available to access at `https://cs3099user-a1.host.cs.st-andrews.ac.uk`.
2. **Testing**: Large suites of automated unit tests.
3. **Group Organisation**: Strict development protocols and group working methodology.

Author: 180017218

1.3 Overview of Success Extent

We believe we have been highly successful in establishing both the fundamental aims of the project, as well as achieving a significant number of desirable but not crucial objectives that we are optimistic can be viewed as of “exceptional quality”.

The basic mechanics of creating, retrieving, editing and curating posts is well implemented and thoroughly tested, with user registration and administration providing enough security and exclusivity to be plausibly usable in a real world context.

Our interactive whiteboards, rich post content types, consistent and responsive user interface, along with our sophisticated permissions system contributes to our product going well beyond what is simply minimally workable, and towards a more finessed and fully-featured application.

Each feature involved some degree of research and independent assessment of alternative implementation options, pulling together functionalities from a large number of external libraries, including *flask-praetorian, flask-socket-io, flask-alchemy, react-sketch, react-boostrap* and many others. The various research tasks and learning requirements have been balanced well over different group members.

Finally, this report demonstrates original analysis of the various mechanisms related to the project that are not primarily code and implementation based. This includes overview of our highly structured planning and group-working methodology via Scrum, our entirely automated deployment and testing pipelines, as well as our comprehensive suites of automated tests for both the frontend and the backend.

Author: 180004835

Chapter 2

Project Details

2.1 Design

2.1.1 Requirements Analysis

Following the Agile/Scrum methodology our design process had to begin with the user requirements.

User Requirements

See Appendix A for our original user stories. The following priority requirements were created through processing these stories.

1. A member of a university module wants to keep in touch with other students and discuss lecture content in a study group

2. A student wants to hear announcements from their school about news or current research

3. A society wants to share announcements with its paid members and offer members a place for discussion

4. A parent of a student starting University would like to receive announcements and keep his details private

5. A lecturer is interested in using Academoo as a discussion platform for her class but wants to make sure only users enrolled in her module will see the content

System requirements

1. Member of a module study group:

   1.1 User can join a community for a module or start their own private study group community by choosing appropriate user roles

   1.2 Users can share posts in the study group community using different content types. Examples include polls to see what other students answered in past paper questions or markdown with mathjax to share mathematical content and code blocks.

   1.3 Users can create a room for a shared whiteboard and send the link via Academoo to all the users in their study group. This allows students to work together to draw out a problem and brainstorm ideas.

2. School-wide announcements

   2.1 A school or department in a University could create a community for their students to follow

   2.2 The school administration and staff could be elevated to higher user roles such as community admins and contributors, whilst restricting other users to only viewing posts and adding comments (General Member role)

   2.3 The school administrators can now make posts to the community and any students interested in those announcements can subscribe to the community and view all the posts

   2.4 The students of the school can comment on school announcements to express interest or share them with friends

   2.5 Members of the school could also upvote or downvote posts in the school and vote in polls to give the school feedback on the content they are providing

3. Society community

   3.1 A society may want to limit the members of its community by setting the default community role to prohibited (private community) and adding members as General Members or Contributors as they pay for membership, similarly to creating a mailing list

   3.2 Alternatively a society may allow non-paid members to view posts so allow everyone Guest privileges

   3.3 In a society members can either make posts or chat to one another through comments and replies
   3.4 A society could share announcements with its members and allow voting on AGMs or merchandise preferences using the poll post type
   3.5 Society members can subscribe to the society's community so they don't miss any posts

4. Private guest access to announcements

   4.1 A parent who wants to receive some information from the University could request to be added to any community they were interested in as a Guest user (allows user to view posts but not interact)
   4.2 For more information and helpful tips on the user roles available, the user could go to the Academoo help page and read the FAQ
   4.3 For a user worried about privacy, there is a private-account option in user settings, which prevents other users (except site-admins) from viewing any of your personal details except your Academoo username for identification

5. Discussion platform for a module

   5.1 The lecturer can set up the user roles for a community for their module with any other module lecturers as Admins and each class member as a Contributor (able to post) or General Member (can comment). The permissions system for the community is only customisable for community Admins (including the lecturer as owner) and site-admins (who must be authenticated with an initial secret key or added by other Admins).
   5.2 The lecturer can share content with the class to start a discussion, or prompt user feedback with a poll
   5.3 The interactive whiteboard feature could be used in tutorial sessions with students for the lecturer to explain a concept aided by a diagram or text annotation. If students missed a class or wanted a copy of the drawings for revision they could save the board as an image locally or the lecturer could share the image of the board to the module's community.
   5.4 The search bar feature is handy for students to lookup posts on certain pieces of course content if they want to look back at discussions for revision.

5.5 Students can ask questions and discuss with one another via posts or comments and replies.

Author: 180017395

2.1.2 High Level Architectural Design

To fulfil our user requirements, it was immediately clear that a web application would be necessary. We conceptualised the application as having two distinct parts, one that dealt with the presentation of the system (displaying posts, communities, user profiles) and another that managed the data and interaction with other groups. The client-server model was therefore a good fit for such a system.

Following meetings with the supergroup it was clear that a REST API was to be the favoured model of supergroup communication, for reasons of simplicity and flexibility. It therefore made sense for us to utilise the same REST API protocol internally decreasing the overall complexity of the architecture by only having to deal with one type of communication protocol.

We would have a “frontend” providing a user interface and making REST API calls to a “backend” that would interact with the database and external servers. The backend would act as the first point of contact for REST API requests from both the frontend and external servers.

See Figure 2.1 for a broad illustration of this overall architecture.

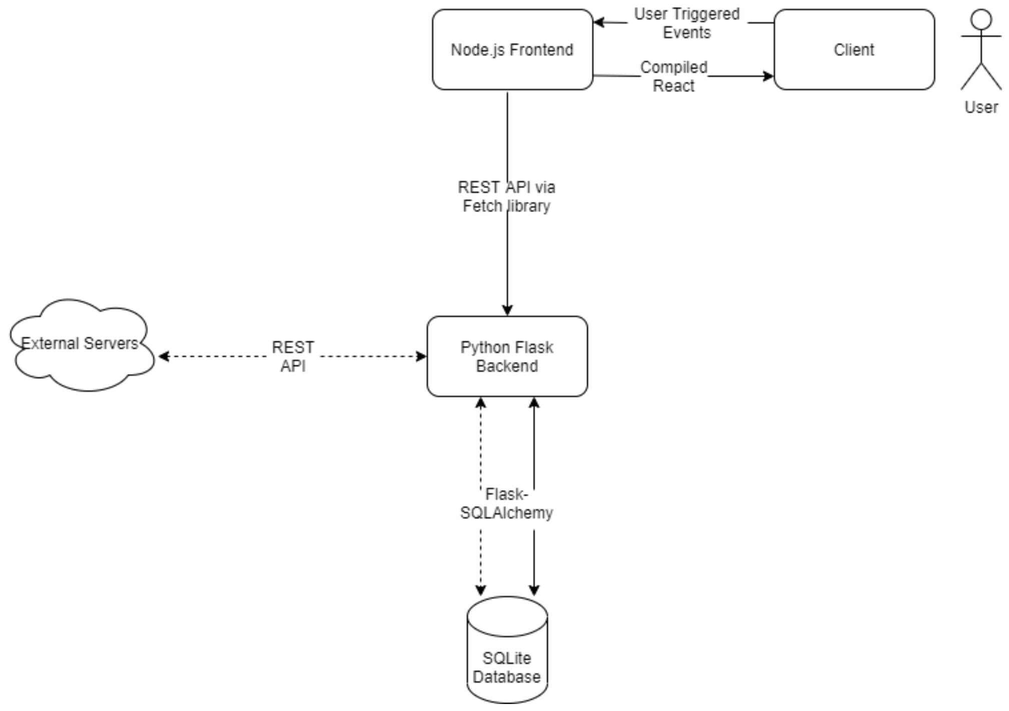


Figure 2.1: Data flow diagram.

Author: 180016547

2.1.3 Back End Architectural Design

Within the backend, we decided to centre our approach on the idea of "end-points". Each route would have a single, clear purpose which would be freely available for the frontend or other instances to trigger by request.

We determined that there were three broad kinds of endpoint:

- **Required routes** for the supergroup protocol: these would be reused where possible for our own purposes to minimise duplicated logic.
- **Authentication routes** which have the sole purpose of verifying a users credentials, supplying them with some form of "proof" that they are able to access other "protected" routes.
- **Internal API routes** that are not available to other servers, including routes for our permission/user-role system.

To keep the distinction between the above clear, we chose to separate them into different modules in our backend architecture.

The route system allows the backend to respond to a particular request on demand. To actually fulfil the functionality and provide the data being requested, we conceptualised another module "actions". This would consist of sub-procedures, one for every possible manipulation of data in our storage system.

Finally, we would have a module whose responsibility would be communicating with other instances. Within every route there would be an option to divert the flow of execution to this module when a request is specified as being "external". Using this design, we anticipated a low maintenance backend, where requests to our own instance and other instances are virtually identical so that the same procedure can be used for both.

The interactions between the components of our backend architecture are summarised in Figure 2.2.

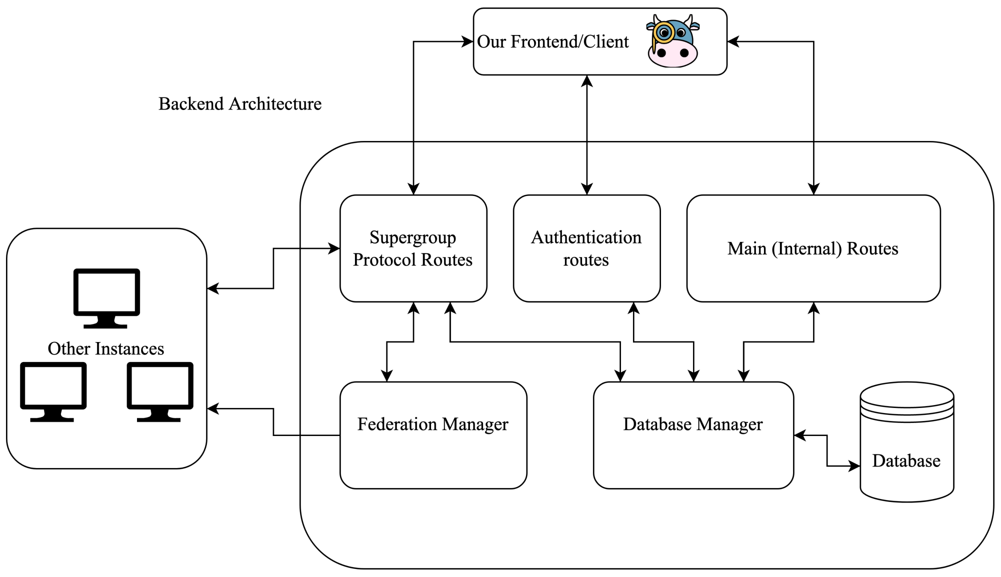


Figure 2.2: Backend architecture digram.

Author: 180005905

2.1.4 Database Design

To store the website data, we use a relational database. The database we decided to use was *SQLite* as it is easy to set up, and we determined the more advanced capability of other relational databases was not necessary for our website. We favoured a relational database over a NoSQL database for its capability to perform complex and efficient queries on data that hold relationships. This is useful for our data as we utilise searching, grouping and filtering of posts within communities.

The main drawback of using relational databases is that a schema must be defined before any data retrieval or insertion can be done. This makes it harder to change the database if design decisions or requirements change. It is in contrast to NoSQL databases, as a schema does not need to be defined beforehand and so changes can be easily made. However, we reasoned that this was not a large issue as the (relatively) small scale of our website meant that changes to the database schema could be performed by refactoring the database without a hit to overall performance.

To integrate the database into our Python-Flask backend, we used the *flask-sqlalchemy* module, which is a wrapper for the SQLAlchemy module and allows it to integrate more easily with our application. SQLAlchemy also provides object-relational mapping functionality. This provides benefits such as preventing SQL injection attacks and providing a more programmatic way to manage the database as opposed to using raw SQL strings.

See figure 2.3 for a diagrammatic representation of the relationships in our relational design.

The `User` table stores information about a user's personal account such as user id, host, whether an account is private and the accounts password hash. A post's author specifies a single user account, so there is a one-to-many relationship between `User` and `Post`. This means it is easy to get all posts authored by a single user.

The `Post` table stores information about a post within a community. To capture time of creation and modification timestamps, the default value of the created attribute is assigned to be the current unix timestamp and the modified attribute set to be the current unix timestamp whenever any attribute is updated. As there is no distinguishing features between a comment and a top level post except the presence of a title, the parent/child relationship is set to be a self-referencing one-to-many relationship from top-level post to comment post.

The supergroup protocol describes the content field of a post as an array of json objects. This makes it difficult to convert into a relational format, so each json object element in the content field of a post is stored as a row in the `ContentTypeField` table. While this increases the complexity of the relational schema by introducing raw json attributes, it allows efficient querying of post content fields.

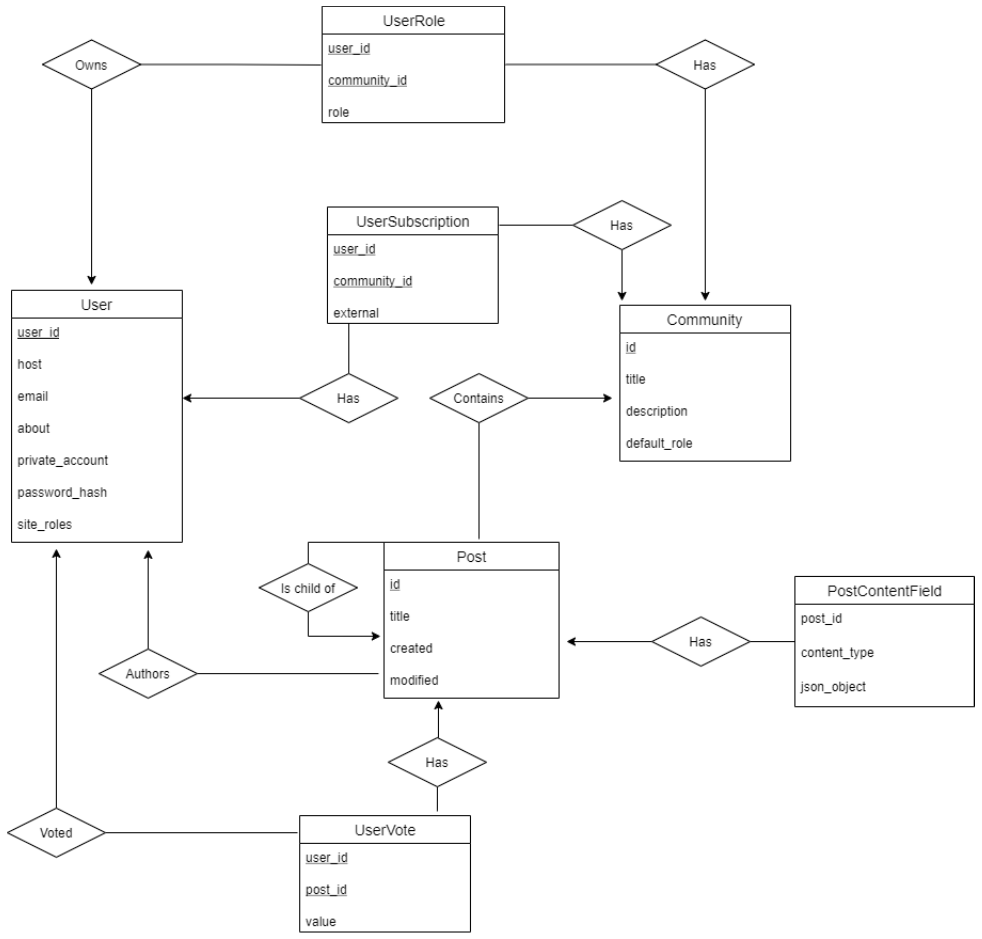


Figure 2.3: Entity relationship diagram.

To model upvoting and downvoting posts, the `UserVote` table is used. Here, the user making the vote and the post being voted on is stored. It is necessary to store the user making the vote as it allows the application to validate whether or not a user has previously voted on a post, preventing a user from upvoting or downvoting a post multiple times.

The `Community` table stores information about a community such as title, description and default role. As posts are associated with a single community, there is a one-to-many relationship between communities and posts. This makes it easy to find all posts associated with a community. Within a community, a user's role is what determines their level of access, whether it be a manually assigned role, or the default role for a community. This is modelled using the `UserRole` table. Every entry in this table holds the community involved, the user owning the role, and the role assigned. It has a many-to-one relationship between both `User` and `Community`. By using this method, it is easy to find all roles a user owns as well as all roles given out by default by a community.

To model user subscriptions where posts from all communities that a user has subscribed to can be displayed in one place, the `UserSubscription` table is used. Here, each row stores a specific combination between user and community that specifies that a user has subscribed to a specific community. It has a many-to-one relationship between both `User` and `Community` and stores the user, community and whether the user is on an external server or not.

Author: 180016547

2.1.5 User Interface Design

The initial design process for the User Interface design was based on a classic User Experience Design Flow.

First, wireframes were created (see Figure 2.4) to explore the possible layouts of features on the site, particularly considering the use of a navigation bar and side bar options.

Following this, user personas were created from the initial user stories to plan the flow through the site using our wireframes.

Once the general navigation and layout were designed and agreed upon, mock-up designs were created in full colour to draft out how the site may look and to decide on a colour scheme (see Figures 2.5 - 2.7).

The mock-ups allowed the team to discuss the User Interface design decisions from the start of the project and have a shared vision of what we were working towards in the final implementation. The success of our design process allowed us to create a useable, clear UI by the Minimum Viable Product submission, developed further as new features were added.

The final UI design is slightly different from the original plan - for example we deviated from having a side bar dropdown for communities on the left-hand side of the newsfeed page. However these changes were made to try and create a clearer, more intuitive interface and the general design decisions made at the start have been adhered to. One example of this is our strict colour scheme, following the five-colour design principle [Cartwright(2020)]. We chose three main colours, a highlight colour, and a dark colour to form our colour palette.

Using Bootstrap and overriding the default colours allowed us to easily set a default colour palette for consistency across the site. React Bootstrap

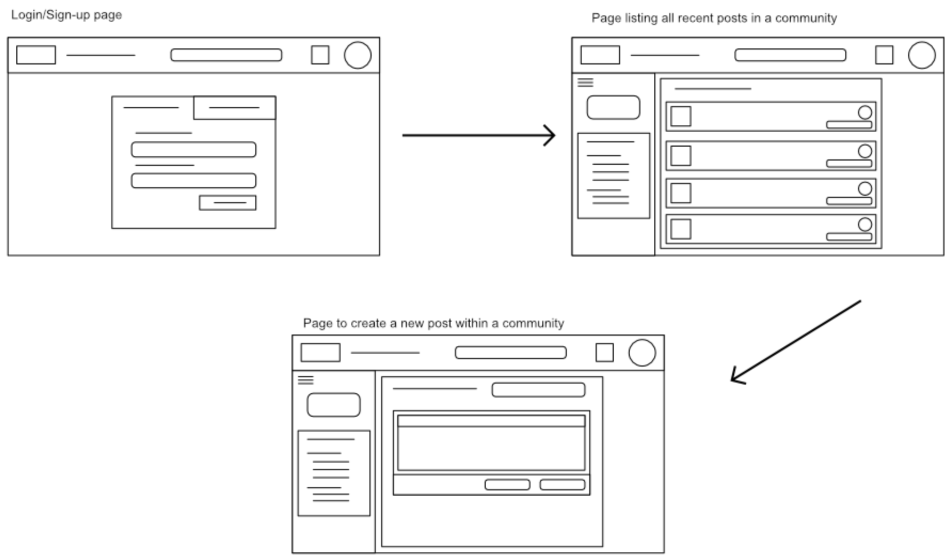

Login/Sign-up page
Page listing all recent posts in a community
Page to create a new post within a community

Figure 2.4: Initial Wireframes

components were fundamental in our User Interface and allowed us the freedom for customisation and styling when required but generally helped us to produce a consistent and intuitive UI. We also used React Bootstrap Icons [1] throughout the site, which helped to indicate what features were offered through standard recognisable icons.

Our layout aims to keep user attention focused on the main elements of the page, for example in the newsfeed having a central column take up most of the width of the page to display a feed of posts. The posts displayed in feeds also have a "preview" style applied to them to prevent very long posts or very large pictures from filling the page.

Post content is truncated slightly and can be viewed in full by clicking on the post: opening the comments view.

We attempted to create a responsive User Interface, meaning our layout adapts to different screen dimensions, however there are some improvements that we would have made to this for smaller screens given more time.

Finally, we chose the name "Academoo", and decided to adopt a Cow mascot, along with using terms involving puns on the word "moo". This helps

[1]https://icons.getbootstrap.com/

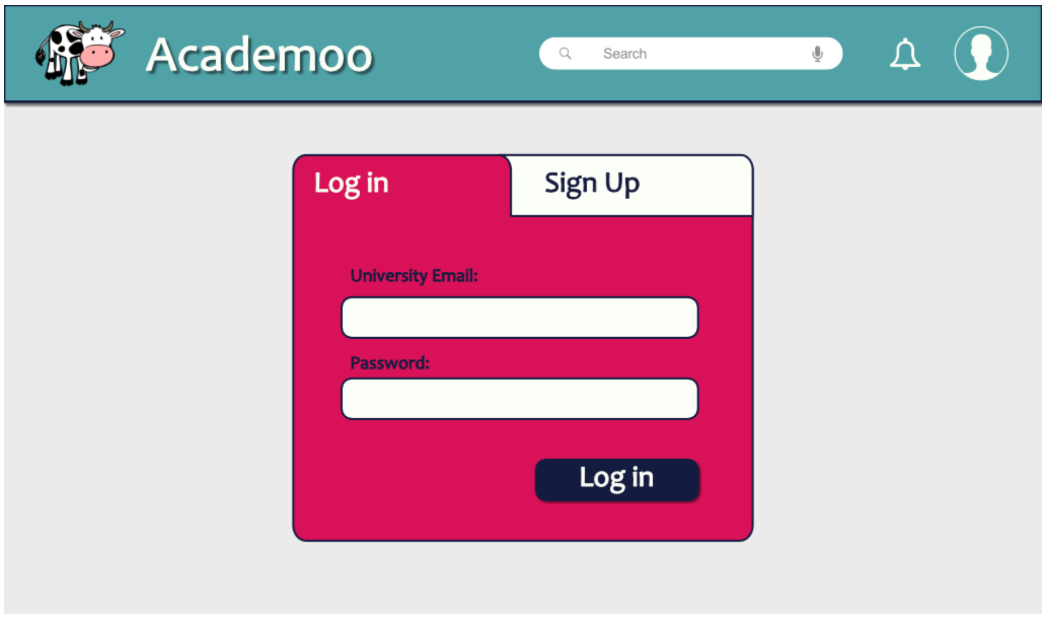


Figure 2.5: Login Mockup

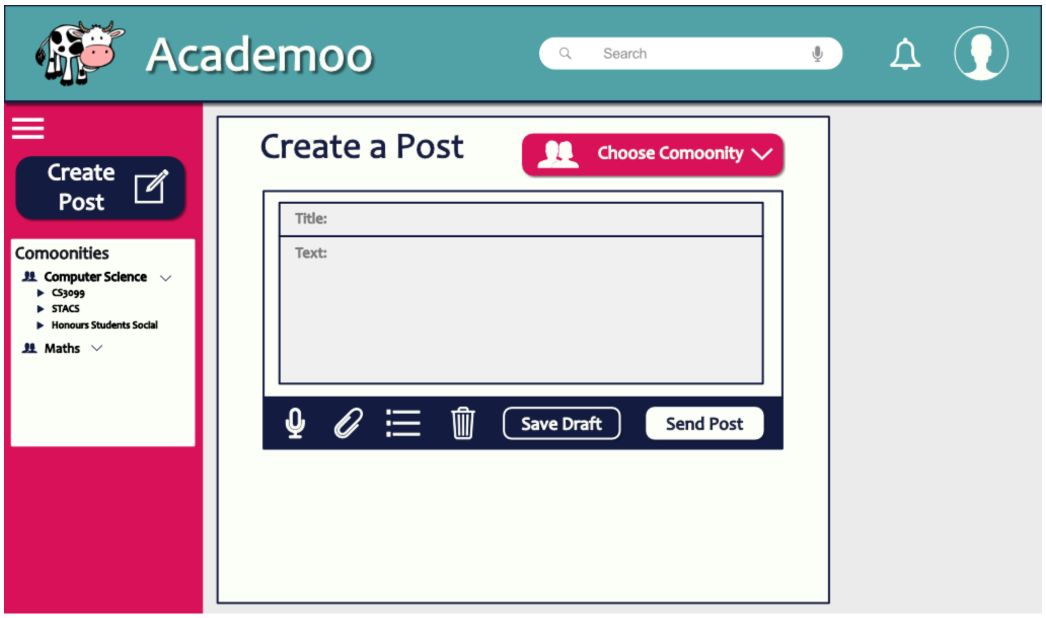


Figure 2.6: Post Creation Mockup

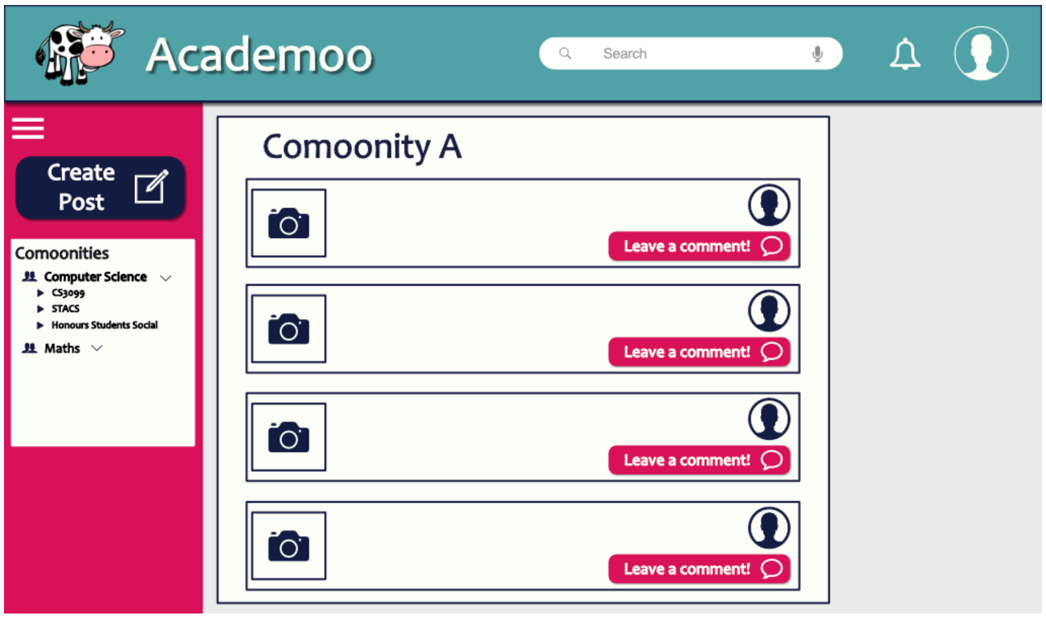


Figure 2.7: Community Feed Mockup

to add to the fun atmosphere of our site, emphasising the more light hearted side of academic life, and making our site distinctive.

Our final user interface (for comparison with our initial mockups) can be seen in figures (2.8 - 2.10).

Author: 180017395

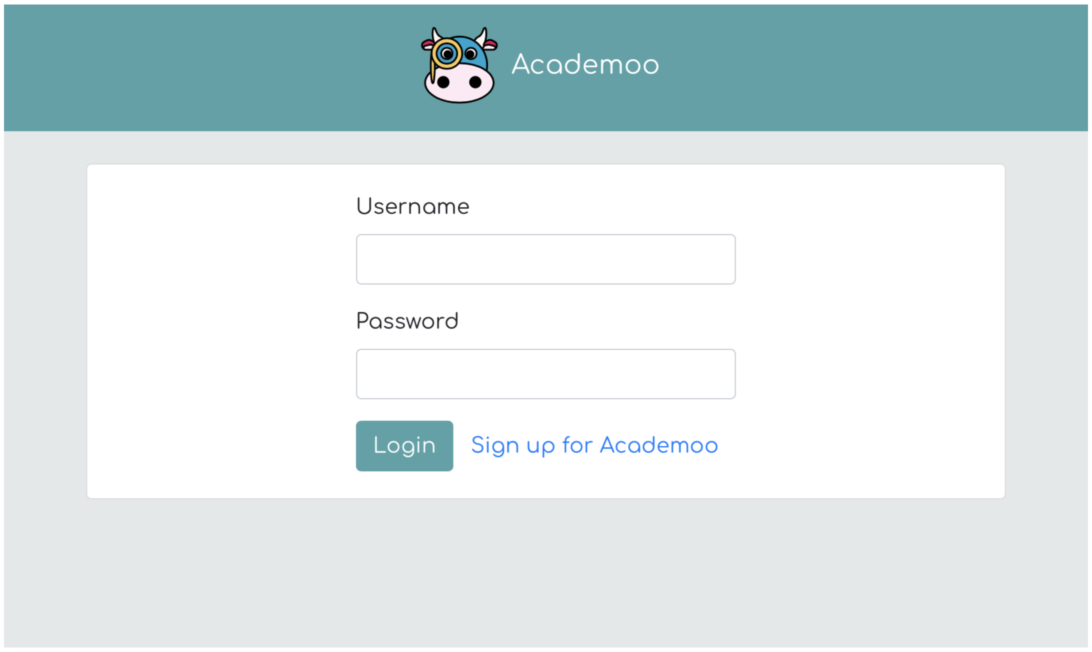
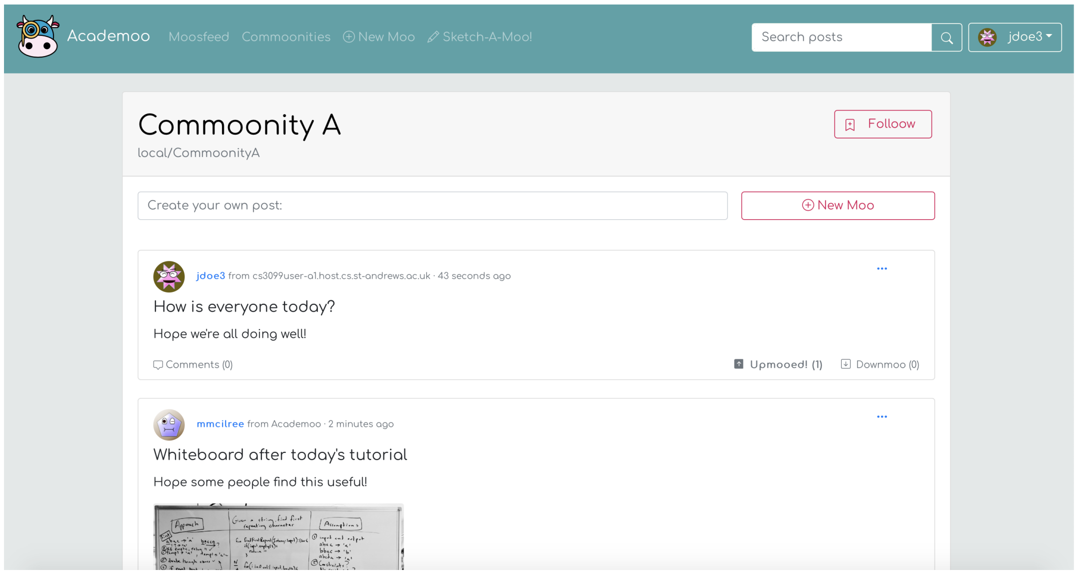


Figure 2.8: Final login interface.



Commoonity A

local/CommoonityA

Create your own post:

jdoe3 from cs3099user-a1.host.cs.st-andrews.ac.uk · 43 seconds ago

How is everyone today?

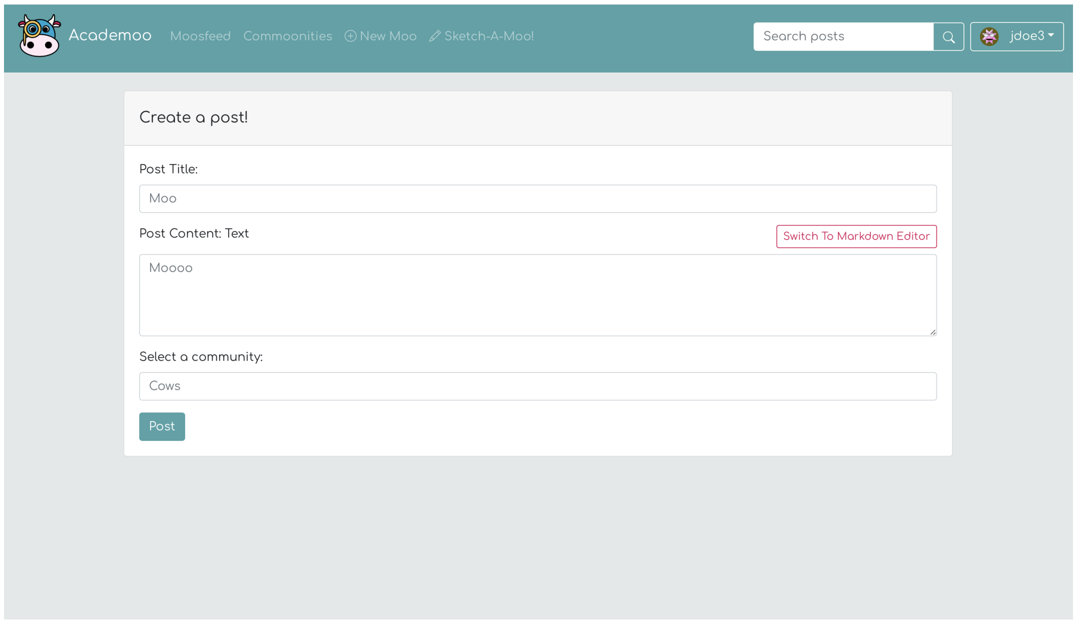
Hope we're all doing well!

mmcilree from Academoo · 2 minutes ago

Whiteboard after today's tutorial

Hope some people find this useful!

Figure 2.9: Final community feed interface.



Create a post!

Post Title: Moo

Post Content: Text

Moooo

Select a community: Cows

Figure 2.10: Final post creation interface.

2.1.6 Front End Architectural Design

Although we had chosen *React.js* as our primary tool for building our user interfaces, we noted that *React* itself does not enforce a particular architectural design. It is technically possible to write a React application in a single class component with a single render method, but this would make it very difficult to work as a team, would be frustrating to test, and would not promote code reusability.

We decided to organise our architecture by composing and nesting components. We aimed for a loosely coupled architecture, where each component knows very little about other components, and is agnostic as to where it appears on a page, providing it receives the correct "props".

Our requirements and interface design made it clear that multiple pages would be necessary, so following the composing/nesting principle we designed an architecture that is essentially a single page application giving the illusion of having multiple pages. A routing library such as *React-Router* could manage the browser routes, rendering the relevant top-level page components when the user navigates to them.

Our layered frontend architecture is illustrated in Figure 2.11.

Author: 180004835

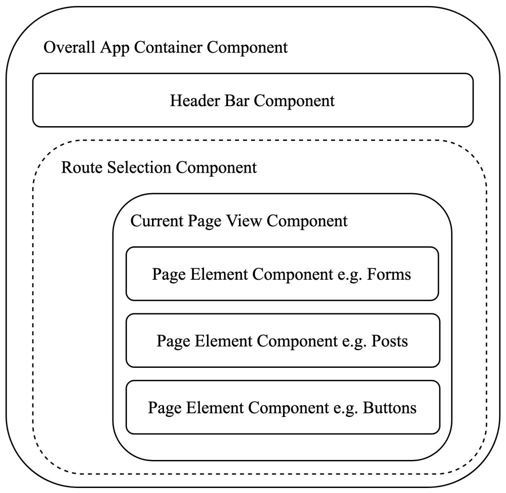

Overall App Container Component
Header Bar Component
Route Selection Component
Current Page View Component
Page Element Component e.g. Forms
Page Element Component e.g. Posts
Page Element Component e.g. Buttons

Figure 2.11: Layered frontend architecture model.

2.1.7 Key Effective Design Features

Components have been used throughout our implementation, and it is the substitutability of components which is so important. We believe our implementation uses this to make change easier, for future development. Componentisation allows for dynamic composition and re-composition, and as we have used components this allows us to have re-usable, flexible, independent functions. In the front end we have followed React's recommended style of passing props down to child components and lifting state up so that the components only contain logic that is directly related to the functionality that they perform. Within this we have used components which can be easily switched in and out, since all that is needed is to pass in the correct parameters. We can replace one instance with another, comparable instance without any noticeable functional change, increasing the reusability and flexibility of our code.

Similarly, with respect to the back-end, we can easily make changes by adding single routes and single actions. Each route and action is completely independent from one another, allowing for an unlimited range of new features to be added if necessary without any architectural changes.

Author: 180017218

2.2 Overall Implementation

2.2.1 General Implementation Principles

In general, as a group we adhered to general principles of good software design as far as possible. Some important principles were as follows.

Modularity

For example, our frontend is split into components, each of which has a single responsibility for user interface features. Components are nested where necessary, and data flows down the way from parent components to child components through React props.

As is recommended for React, we favour composition over inheritance[2], for our interface components.

In our backend we use Flask blueprints to encapsulate functionality, registering reusable functionality in the main `app.py`.

Reusability of Code

In many cases we avoid repetition by factoring elements used multiple times in various compositions into their own separate module. Examples of this include, the reusable `UserVote` component, `CommunitySubscribeButton` and general `Post` component, as well as the generic class representation of an `Instance` object for dealing with multiple other servers.

Object Orientation

Where possible, we have adopted an object-oriented approach, instantiating the base classes from various external libraries, and representing both our frontend elements and backend data structures as objects.

Author: 180004835

2.2.2 Technologies Used

Our web application is hosted on the school servers. We use *Nginx* as our intermediary proxy service to direct client requests to frontend and backend servers. The frontend is developed on the *React.js* framework and utilises a customised Bootstrap library for styling. The npm package *serve*[3] is used to

[2]https://reactjs.org/docs/composition-vs-inheritance.html
[3]https://www.npmjs.com/package/serve

deploy our frontend build into production.

The backend is a Rest API server built using the web framework Flask. This server is deployed using *gunicorn* which is a Web Server Gateway Interface (WSGI) application that is commonly used to deploy python web applications. Therefore, *gunicorn* acts as a bridge between the web server *Nginx* and the web framework *Flask*. Lastly, the backend server is connected to a database for data persistence. We are using *SQLite*, which is a relational database system with built-in Python support. Unlike other relational database management systems, *SQLite* does not run on a server, instead it is directly integrated with the server.

Author: 180005905

2.3 Implemented Features

2.3.1 Registering Users and User Authentication

Functionality to register users and some authentication was implemented in our Minimum Viable Product, where the user could create an account with Academoo and choose their own username and password. However, over the Christmas period the functionality, security and strength of the authentication was improved, after feedback from our supervisor. We acted on this feedback and as a result have much improved from the Minimum Viable Product.

Whenever accessing the site while not logged in, a `PrivateRoute` component ensures that users will be automatically redirected to the login page to begin accessing the site. However, if a user is logged in and so has an account, the default "`/`" route directs to the Welcome page.

New users will need to sign up for an account, and when doing this they must provide:

- An email address within the correct format (`text@text`)
- A unique username, this will be one which is currently not in use and therefore not stored in the database, as well as a username which is longer than 3 characters
- A strong password which must contain 1 capital letter, 1 number and 1 special character, as well as being 8 characters long.

In adding this stronger security, we protect our users against unauthorised access.

The error handling in the sign up form became the model for error handling throughout our forms on the site. Any errors in input, detected via a defined set of conditions, will be mapped directly to error messages inside Bootstrap `Alert` components, once the form has been submitted. This tells the user the exact error, which improves our accessibility as we inform our users of what has gone wrong, and specify what needs to be done to advance.

Once they have successfully signed up, users are taken to the login page to sign in with their new credentials. If they have entered the incorrect username or password they will be met with another error message encouraging you to try again. The sign-up and login pages also mutually link to one another.

Once successfully signed into Academoo, they are re-directed to the welcome page and from there can navigate the rest of the site.

At the beginning of semester 1, we considered adding email validation to check that the user was signing up with a real email, possibly restricted to a university domain. However given the complexities surrounding automated mail server administration, and the potential restrictiveness of requiring real email addresses, we decided not to pursue this feature.

Author: 180017218

2.3.2 Creating Posts with Content Types

Basic support for creating posts was implemented in the MVP, where users could create a new post in a given community with a title and text body. The create-post form was improved to include stronger validation to prevent users from posting in non-existent communities and enforce user permissions, with meaningful error messages.

Additionally, support was added for a wider range of post content types (in line with the supergroup protocol), for example posts with images and Markdown posts. Markdown support was added using the *react-markdown* library, to render the posts and create a preview component, allowing users to see the formatted post while writing.

This preview component renders the Markdown dynamically as a user types in the markdown editor and displays tables, images, code blocks and other advanced Markdown features. An information box was added to give users a general guide on the formatting options in markdown.

Since this is a social media for a University Community, support was added for MathJax to render mathematical formulas in community posts, so that students or staff in Mathematical subjects could discuss and share the content with appropriate formatting.

Posts are always displayed in the same way, whether they are in the news-feed, community feed or user profile posts list. The posts display with the title and content (in a preview form on the feed and in an expanded form in comments-viewer). Additional information was added to the posts such as the author username, community ID and host name, with links to the profile or community page for easy navigation. The time since a post was created is specified in appropriate units (seconds, then minutes then hours, days etc.) and there are buttons to view the comments on a post, upvote or downvote and an options menu in the top right corner.

This menu gives users the option to edit or delete a post if they have the required permissions - they must either be the author of that post or have community admin or site-wide moderator privileges. Deleting a post prompts a pop-up to ensure that the user is not making a mistake, on confirmation the post and all its comments are removed. This approach was chosen since the comments refer to the now deleted post and do not make sense as top-level posts in their own right. Editing a post prompts a more concise version of the post creator component to appear in a modal pop-up. A user can change the post title and/or contents but cannot change the content type: if the post is a markdown post then the markdown previewer will appear by default in the editor. Editing support is also available for comments without the ability to customise a title (since comments do not have titles).

Author: 180017395

2.3.3 Creating Comments

Commenting on posts was implemented in our MVP and allows for an infinite number of comments to be added on a single level. However, there has been many changes to comments since then, with comments able to be voted on, and comments allowing replies.

Comments use an identical underlying JSON structure to posts, which allow us to re-use our code for viewing posts for comments. Only users with the correct permissions can comment on posts. To post a comment, the user clicks the “Leave a Comment” button, which opens a pop-up (via the `Modal` component) and allows the user to enter the text they would like to comment. Once submitted, the `CommentViewer` component automatically refreshes, and the user can see the post with their new comment. Every comment contains the name of the user who wrote it and what site they posted it from. The user’s name is clickable, so others who would like to know more about this commenter can click the name and be taken to that persons profile page.

Comments now also support a second level of discussion threading via replies. Every comment can have an infinite number of replies, and the current number of replies, as well as the ability to reply to the given comment is displayed on the comment. Replies are children of comments, with the parent of a reply being the comment it is associated with. To reduce the complexity with commenting, we decided that replies cannot be replied to: similar to Facebook (see later section on comparison with similar applications). The supergroup protocol in theory supports infinite comment nesting, so our single level of replies is compliant with this. Comments and replies also support markdown.

Comments allow users to interact with others users' posts and allow discussion. This aids our university site as a lecturer could post, and students could post comments and answer other comments, like it is done in Discourse, for example.

Author: 180017218

2.3.4 Communities

After registering users, and creating posts and comments, we felt the concept of a "community" was the next most fundamental feature for the Academoo site. Communities would act as a way of grouping posts, and organising common interest groups; be it for a module, society, or discussion topic.

The supergroup-determined structure of a community was simple: consisting of a title, community identifier, description, and list of associated posts. Each instance would support their own communities, while also allowing the communities on other instances to be viewed.

The earliest implementation of communities was simply a way of switch the source of the current list of posts in our single `PostViewer` component. As the scope of communities increased, with the ideas of subscription and administration coming into play, we decided to create a dedicated page "Explore Communities" which would showcase all the communities available across all instances.

In general the component interaction for the concept of a community was as follows:

- A community is created by a user via the `CommunityCreator` component mounted on the `/create-community` page.
- The community is visible on the `/communities` page.

- The community is given a dedicated route at `/communities/<communityIdentifier>`, and its posts are visible via a mounted `PostsViewer` component.

The *react-router* library we made use of was crucial for this structure. Giving each created community its own route (possible since uniqueness of community identifiers at the instance level was enforced) allowed us to make the site easily navigable, creating the illusion that each community exists on it's own static page, despite being simply a component with dynamically fetched data.

Author: 180004835

2.3.5 Post Curation

With communities being the most basic way of allowing users to curate and group posts, we expanded significantly on this feature by providing multiple ways for users to personalise and categorise posts, making the site much more useable and interesting when larger volumes of posts are circulating.

The first obvious step was the concept of community "subscription". This would allow users to select certain communities to add to a list of subscriptions, and then see all the posts from these communities most of interest in an aggregated `SubscribedFeed` (which we has styled as the "Moosfeed"). The Subscription functionality was based around a many-to-many relationship between Users and Communities, although we realised since community identifiers are not enforced as unique across all instances by the supergroup protocol, we would also have to label this relationship with the host of the community.

The subscription flow is then as follows:

- A user can subscribe to a community by clicking on a corresponding `CommunitySubscribeButton` (a reusable component that makes the necessary subscription request to the backend for a given associated community).
- The `SubscribedFeed`, mounted at the `/moosfeed` route first retrieves the list of subscribed communities for the current user, and then fetches the post data from each community to display in aggregate.
- The (sorted) post data is passed to the generalised `PostViewer` (the same as the one used in the community feed), for consistent display.

At the sorting stage in this process, we realised the possibility of providing additional curation abilities for posts by allowing user ratings of the post content. Rather than adopt the common social media convention of "likes" or "reactions", we decided to implement a more neutral voting system (styled as "Upmoo" and "Downmoo").

This proved to be a more subtly difficult feature to implement than we had anticipated. While a simple tally counter in the post data was enough to keep track of how many upvotes and downvotes a post has, care had to be taken to enforce the obvious necessity of disallowing users from voting more than once on the same post. We therefore had to create a relationship between users and posts votes, where each user has a single vote that can take the value of "upvote", "downvote", or "none".

When a user has not yet voted on some particular post, the tallies for the relevant vote type are simply incremented and the user's vote relationship is updated. But when the user votes on a post they already have a vote for, the following logic had to be adhered to:

- Upvoting an upvote: equivalent to removing your upvote. Vote becomes "none" and upvotes is decremented.
- Upvoting a downvote: equivalent to changing your upvote to a downvote. Vote becomes "downvote", upvotes is decremented and downvotes is incremented.
- Downvoting a downvote: (vice versa).
- Downvoting an upvote: (vice versa).

With this feature implemented, we were able to offer three ways of sorting posts on the Moosfeed: by most recent, by best voted (upvotes minus downvotes), or by most commented.

Author: 180004835

2.3.6 User Profiles and Settings

User profiles had a very basic implementation in the MVP in November. The user only had a username, default image and email displayed, over the Christmas break and into this semester, they have been given much more functionality and purpose. These additional features help personalise the user profile page, and give the user a distinct identity

To simplify our data storage system, Users profiles are supported by a third party service: Gravatar[4]. To add an image to a profile, a user must have a registered account with Gravatar, and that account must use the same email address as their Academoo account. In Gravatar they would select a profile picture, and the image associated with their Gravator account becomes the image associated with their Academoo account.

Users now also have a biography associated with them, much like in social media platforms such as Instagram and Twitter. This biography is set by the user and can be up to 140 characters, to allow for a “short and sweet” feel. We felt the biography was important, as it added personalisation to the users profile, and allows other users to understand and interact with one another: for example by identifying a user’s role within the university (“Third year maths student”, “Biology Lecturer”).

User profiles now also contain the list of posts from that given user. The inspiration for this was drawn from Reddit, where the user can see all the posts they have created on the site. If the user has not yet posted anything on Academoo yet, they are met with a message encouraging them to post, which directs them to the “New Moo” page allowing them to create a post on any existing community. If the user has posted, these posts are displayed in chronological order, and each in their own card to make it clear where each post starts and ends. In adding this feature, other users of Academoo can see the contributions of the given user and navigate to their post within a given community.

Over the Christmas break we also improved the user account settings of Academoo, whereby users could take actions such as changing their password, updating their bio, making their account private and deleting their account.

Changing password was a key feature for us to implement. It allows the user more control over their account, letting them change it to something easier to remember, for example. The user must enter their current password as authentication, and then they can enter a new password. The changed password will only go through if the current password hash matches the hash stored in the database. Once this has gone through, the user is notified with a green confirmation message telling them the password has now been changed.

The user can also modify their bio: the user starts out with not having a biography and can change/create one through the settings. Under ‘Update Your Bio” the user can enter 140 characters to let other users know about them. Once they are content with what they would like to add, they press the update button, and they are then navigated to their profile, where the

[4]https://en.gravatar.com/

updated bio is shown.

An important privacy feature we have also implemented is the "Private Account". We are conscious that some users might not want others to see their email, bio, profile picture or posts. So to make sure we are catering for all users, we added in the private account, which is a toggle which gives users the freedom to control their information. If not private, all users can see all information, if private, the other users can only see the username. We believe this is a key feature as it allows for privacy for our users, which we have considered to be important since we created our original user stories.

The final piece of functionality within settings is the "Delete Account" feature. We acknowledge that not all users will want to keep their account, and so if a user so desires to not be a part of Academoo anymore, they have the choice to delete their account. To delete, the user must enter their username, and password. The password hash is checked against the hash stored in the database to ensure they are the same. The user then clicks the submit button, and a pop-up window making sure that the user wants to delete their account appears. If the user is sure they can confirm this, and if the data entered is correct, they will be re-directed to the login page as they are no longer a registered user of Academoo.

All of the functionality and features implemented within user profile and settings add to the community feeling of Academoo. We believe these additional user account features implemented take our implementation well beyond the minimum standard of "registering and authenticating" users.

Author: 180017218

2.3.7 User Roles and Administration

The complex user roles and permission system was one of the biggest features we implemented in the project. This feature was added after feedback from our supervisor on the MVP: suggesting that a user permissions system would be a useful feature to add. User roles were also mentioned in our initial plan for creating private communities and a reporting system with moderator privileges. To implement the level of access restrictions required, community-specific roles were created in a hierarchy system and site-wide roles were added with over-arching permissions. An overview of the user roles framework can be found in our original design document in Appendix B.

In any community, a user can be assigned one of the following roles: Admin, Contributor, General Member, Guest or Prohibited. The permissions were constructed in a tiered system where a role automatically inherits all privileges of any role below it in the order of the roles listed above (e.g. a General Member has all the permissions of a Guest plus additional permissions). The only exception to this is the Prohibited role which none of the roles above inherit from, since this role prevents a user from interacting with any posts in a community.

Communities can have default roles set to give base permissions to all users who interact with them. The default role is initially set to "Contributor" for any new community but can be changed to any of the Community Roles. If a user makes a request, for example to view or post in the community, without being assigned a specific user role - the default role will be checked to see if they are allowed the permissions to perform the action.

On creating a community, the user is given automatic Admin privileges for it (to ensure that there is always at least one community Admin). This first Admin is the owner, who can immediately access the Community Manager page to assign roles to other users on any instance or set a default role for the community. The Community Manager page also displays a table with all the individual users that already have roles assigned to them. Community roles can be added, promoted or demoted for any user. Admins may not change their own role, to avoid a case where a community has no Admins, however another Admin may remove the Owner's Admin privileges.

Permissions to perform various community actions such as viewing posts/comments, creating posts, deleting posts etc. are listed in the design document (Appendix B). These permissions are enforced by protecting the API endpoints and returning HTTP 403 status errors with meaningful messages to be displayed to the user if they don't have permission to perform an action. The permissions system is implemented in the backend by querying a User Roles schema in the Database, as mentioned in the database design, for a user making a request, and also querying the community's default role if required to check their permissions. The implementation is added in the backend for security, since the fetch requests for each action should not return any data to the front-end if the user does not have permission. Server-side security is prioritised here since client-side validation may not be sufficiently secure to prevent a malicious user from extracting hidden data, despite not having permission to view it.

Initially, we planned to use a pre-existing flask library to construct the permissions system, to avoid the need for permission-checking logic in the endpoints for every action. However, after some research we found that none of the standard libraries offered the level of access control we required to construct a hierarchical permissions system for communities plus support for site-wide roles. Following this, we decided to create the permissions system from scratch, making it a much more time-consuming and complex task than initially anticipated. However, after several revisions and improvements the feature meets our design plan and fulfils all the requirements we had. The only complication with our implementation is how the permissions are tightly coupled with the endpoints for the actions themselves. Given more time we would have liked to explore different ways to construct our own permissions library and try and use decorators in python or some similar implementation to extract the permissions logic out of the underlying code for our other features.

Overall, the community roles are very powerful and give users a lot of control over how they interact with communities and control access lists. One use case for our system that was mentioned in our original user requirements at the beginning of the project was private communities. To implement a private community, an Owner can set the default role to Prohibited (meaning no user can choose to view the posts or interact with the community other than viewing its description). The Owner can add individual users to the private community by elevating their roles to anything above Prohibited, for example Contributor. In this way the Owner (and later other Admins) have complete control over who can interact with the community.

An extension of this feature would be to let users specify their own roles for community members and the associated permissions. This would be a very powerful feature, offering users a lot of flexibility - however it would require a careful refactoring of our tiered permissions system to allow customisation, so we considered it too much work for the scope of this project.

In addition to community-specific roles, our implementation also offers site-wide roles: Administrators and Moderators. The Administrator role can be first assigned to a user if they have access to a private 20-digit key (similar to an initial login key). For the purpose of our testing and demonstration this is a hard-coded key, however if this product were to be distributed to Universities it would be simple to add a key-generating process to give each customer a unique key so they could set up their initial Admins. As an Administrator or Moderator (or both), a user has access to a control panel with various settings for each role. Administrators have the option of adding other users as Administrators or Moderators or removing their permissions entirely. Just as with community admins, site Admins cannot change their own roles. Moderators have the option to disable any user's account, removing their access to the site by preventing them from logging in (this gives a meaningful error message on attempt). Moderators can also re-activate a user's disabled account.

To implement site-wide roles, the authentication framework: Flask Praetorian, also used for our log-in token authentication process, was used to set user roles. This implementation allows a list of roles to be specified for a user and provides Python Decorator functions to be used with API endpoints to prevent access for users without appropriate permissions. Unfortunately for the use of our community permissions system, in many endpoints these decorators could not be used, since all of the users' community-specific roles also had to be queried, hence access to the endpoint was necessary. To support disabling and re-activating accounts, a “Basic” site-wide role was added as a dummy role required for log-in. This role was added by default for every user in the database and could be removed to disable a user account.

The site-wide roles also provided users with some access permissions in communities, including overwriting some Community Admins permissions in certain cases. For example Moderators can edit or delete posts made by any user in any community.

Further extensions could grant additional privileges to site-wide roles such as verifying communities, removing communities and removing user profile information, but given the time constraints we did not pursue these.

Author: 180017395

2.3.8 Security

The security protocol is described in the supergroup protocol specification[5]. It is a thorough implementation that closely resembles a simplified version of the HTTP security protocol[6]. It relies on message digests and digital signatures to ensure authenticity and integrity between the instances within the federated network. The underlying asymmetric cryptography algorithm is the RSA public-key cryptosystem. The messages are hashed using SHA-512.

Author: 180005905

2.3.9 Whiteboards

The decision to implement a whiteboard arose from research period where each member of the group considered how we might implement a different “exceptional” or “showstopper” feature. These included live-chat, video-calling, automation and others.

The main justification for choosing whiteboards was that in a University community there could be many uses for drawing out ideas, diagrams, annotations and sharing those creations with others. For example, we imagined a

[5] https://fmckeogh.github.io/cs3099a-specification/#tag/Security

[6] https://tools.ietf.org/id/draft-cavage-http-signatures-12.html

community for a module where a lecturer could draw a diagram to demonstrate a concept on a shared whiteboard with the rest of the class. Class members could see dynamic updates as the lecturer added to the drawing and the drawing could be saved and shared for anyone who missed the live demonstration.

A drawback of this feature was that it may be similar to existing whiteboard tools such as Microsoft Whiteboard in Teams or OneNote. We have tried to create a feature with a unique purpose, to give users access to a range of features to draw diagrams or annotate text and to share their creations with others through live updates as well as static posts. We have built a social media site where users on phones or tablets can draw what they're thinking and share it with friends or class-mates in study groups all in one seamless application. This feature was designed for a University community and with the combination of drawing tools and sharing capabilities it has more unique potential in comparison to the static OneNote document or Whiteboard shared via a Teams call.

The basic whiteboard was implemented with a library called *React-Sketch*, which provided built-in support for a canvas and a range of drawing tools. Features offered to users include drawing tools: freeform, straight line, rectangle and circles; pan, select and erase tools; colour picking and line thickness options; and a text tool. Furthermore, whiteboard sketches can be cleared, saved as a PNG to the user's device or shared on Academoo in a Markdown post.

We then built on this implementation using *Socket.io* to create rooms with unique room codes that allow users to share live whiteboard updates with other users in the same room. These rooms are temporary and can be created by any user when they create a new whiteboard. The link or code to the room can be shared with others via Academoo posts to allow them to join the room and watch or contribute their own drawings on the shared canvas.

A Flask library for *Socket.io* for Python was used to set up a WebSocket server in the backend that responds to client requests to join or leave a room or share a whiteboard update via a message. The frontend accordingly uses the JavaScript *Socket.io* client library to make requests to the server and send updates. The whiteboard updates are made by sending the current JSON state of the whiteboard in a message to the room (the server receives this message from the sender client and broadcasts the update to all other clients in the room). The current implementation of the whiteboard allows a user to create a room, join a room and leave a room and share dynamic updates to others in the room (any user in the room can draw and it will propagate to others).

The rooms are temporary, since the codes can be re-used after a room is closed, so we did not want to store details about rooms in the main SQL database - however we would have liked to store permissions information. For example, we planned to have a concept of an "owner" for a room and give them associated permissions to change settings that allow or prevent other users from contributing. Although we did not have time to add these final implementation details, we have created an effective shared whiteboard implementation that shows a strong proof of concept for our original planned feature.

Author: 180017395

2.3.10 Search

As another extended feature, we added a search box to query posts on Academoo. The search functionality is an effective way for users to find content. The metric behind our search algorithm is Levenshtein Distance [Black(2008)]. Given two strings, this metric returns the minimum number of single character modifications to make both strings identical. However, this methodology alone would not be effective when the search query is a short string and the posts are long paragraphs. Therefore, we perform substring matching to find a substring with the smallest Levenshtein Distance to the search query. This implementation enables us to examine the relevance of all the posts.

Author: 180005905

2.3.11 Polls

For the specific additional post content type of polls, our design and functionality was greatly inspired by Twitter Polls. This meant that the results of the poll can only be viewed after the user has voted. Moreover, the user can only vote once and their vote will be anonymous. This design decision makes our polls effective for many applications such as collecting unbiased feedback for a module.

Author: 180005905

2.3.12 Federation

As previously stated, the application utilises federation between groups in our supergroup by allowing any group to access any other groups communities and all content within them. This is done by having all groups in the supergroup follow a protocol for communicating between servers. By doing this, all groups know what kind of data they expect to receive from other servers. The protocol is built around sending and receiving HTTP requests to and from REST endpoints to ask an instance to perform a certain action.

The supergroup protocol specifies that data be sent and received in JSON format. This is due to its simplicity and flexibility as well as its extensive support in most relevant programming languages. The protocol also considers the possibility of invalid requests or requests without the necessary permissions being made. In such a case, a JSON file with an error message is sent. We utilise this aspect to perform various validation on incoming and outgoing requests.

For our instance to communicate with external instances, they are represented as an object of the `Instance` class. Each object stores relevant information such as server name and host URL. Class methods are used to communicate with external instances by calling the endpoints that follow the supergroup protocol. In addition to this, we have created another class called `Manager` to manage all `Instance` objects, which facilitates communication between backend code and other instances. The `Manager` class is also responsible for caching posts retrieved to prevent an excessive number of requests to other instances. We have decided to implement LRU Cache so we can identify when the cache is out of sync when the "latest update" timestamp is changed, as specified in the supergroup protocol. When the backend wishes to communicate with another instance, an `?external=` parameter is specified in the URL. This specifies that the request is asking for data from an external instance. The backend then uses this parameter to communicate with the `Manager` object instead of our local database.

As we are taking data from an external location, the quality of data cannot be guaranteed. Therefore, various forms of validation are implemented to ensure that invalid calls do not cause the application to error. Each request where a JSON document is used, a schema is specified in the supergroup protocol. Validation is performed to ensure each JSON document that is received matches the JSON schema specified. If a request fails this validation, an error message response is returned to the requesting instance and the request disregarded. This is done using the *jsonschema* Python library which allows a JSON schema to be specified in dictionary format.

Author: 180016547

Chapter 3

Development Model

3.1 Agile and Scrum

Agile Methodologies is a project management process which involves discovering requirements and developing solutions through the collaborative effort of the group and considering the end user. By breaking the project up into several phases, the group is able to discover what needs to be done, and task allocation is much easier within the group. One of the key goals for using agile methodologies is to create better software is responding to change: if there is a blocker, the task needs to change with this blocker. Agile is considered a better practice to follow than the traditional "waterfall" approach, as with Agile one can start coding as soon as you have an idea of what the customer wants, whereas with the waterfall, the software development process requires many documents and analysis before developers can begin; the waterfall process takes far longer than Agile. Agile stresses collaboration over documentation and the ability to change, rather remaining within a rigid process.

The agile framework we decided to use is called Scrum. This is centred around the idea of 'sprints' and meetings, which our team has closely followed. Scrum involves planning, where sprint priorities are identified and commitment, where the team reviews the plans and decides what can be done in the given time frame. A large part of scrum involves daily standup meetings where the team can inform each other, of progress and blockers. A "scrum master" is delegated to help the team in the Scrum process. The Scrum Master delegates tasks and runs the stand-ups to ensure they run smoothly and correctly.

As well as having a Scrum Master in the meetings, a Product Owner can also be there to represent the client and answer any questions the developers may have. The product owner gives the insights, ideas and feedback to the product vision. They work with the development team to make their vision a reality. The product owner can break down the product into their vision using user stories that show in detail who the target audience is, and what the problem is for them and how the software being built will solve their problem. The user stories are then shown to the developers, to ensure they all share the same understanding of what is being asked of them.

Within agile development there are common practices used such as:

- **Pair programming**. This is where two developers work together to achieve higher quality code and it also helps more advanced developers teach those less advanced.

- **Test Driven Development and Automation**. We realise these principles through using our constant testing of current features to find bugs and fix them straight away, instead of testing at the end, and discovering those problems afterwards. By taking inspiration from test driven development we allow for the generation of new tasks based on bugs and problems found in the working code.

As a group, we have followed Agile and Scrum very closely. Throughout the year we have had sprints, Scrum masters, product owners, pair programming and some test driven development. As a group we feel that using Agile and Scrum has thoroughly helped us in our planning, development and execution of our project. In the next sections we describe precisely how we implemented these principles in our group-working.

Author: 180017218

3.1.1 Planning Methodology

In order to follow Scrum, we have been having fortnightly sprint kickoffs since the beginning of the project. In this we start a two-week sprint, decide what needs done, and assign tasks. The sprint kick-offs are run by the Scrum Master (which in the beginning we assigned randomly to give everyone experience, but as the project progressed became by volunteering, since some members felt more comfortable in the role). The Scrum Master then creates the new sprint, decides with the group what we are going to prioritise and work on, and assigns tasks to the relevant people. These kickoffs are attended by all group members and usually last around an hour as we discuss what needs done and delegate. We used *Jira* for tasks, as will be discussed below.

Every weekday we have a daily stand-up, where we discuss what we did the day before (which may be nothing due to work for other modules), any problems we have encountered (blockers), and what we're planning to do that day. These daily stand-ups have allowed our group to understand where everyone is, and they also allow for group discussion when someone is encountering a blocker and needs help. These are not compulsory to attend for our group, however the majority of our members have attended nearly all of the meetings.

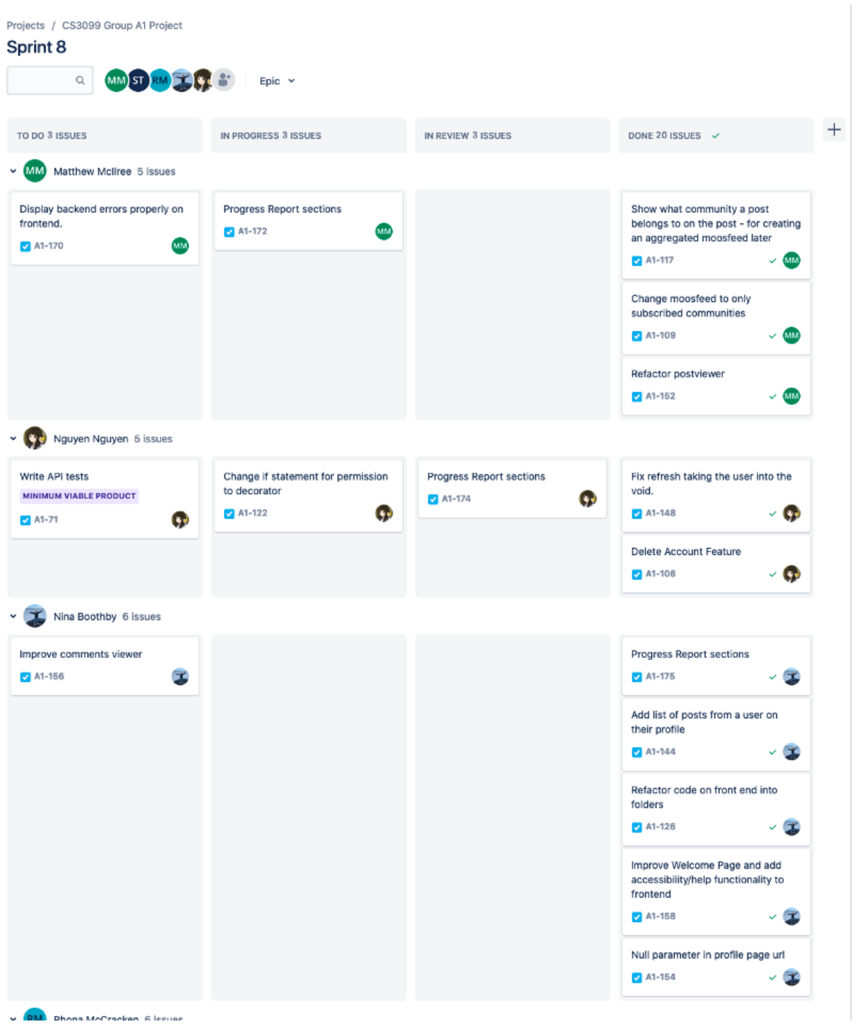
Author: 180017218

3.1.2 Group Workflow

To describe the group workflow throughout the year, it is best to consider the context of a typical sprint.

At 12pm on a Monday we meet on Teams (due to obvious remote working constraints) as a group to begin the new sprint. Once everyone has joined the call we begin. First, we begin with delegating a new scrum master. Member A volunteers to be scrum master. From here, member A shares their screen and shows the current *Jira* board which contains the sprint about to be finished. Any members who have not put their tasks in the correct section (completed/in review/in progress) are given the chance to move their tasks before the new sprint begins. Once everyone is happy the tasks are in the correct place, the sprint can be finished and the new one can begin. Any tasks not completed in the last sprint are moved over to the new one. The scrum master then looks into the backlog at tasks which are needing to be done. As a group we discuss which ones we believe we should implement and also look to our semester plan for what we had hoped to achieve in the current sprint. Member A then moves the task into the new sprint and assigns it to the relevant person. We continue this process until the scrum master is happy we have enough tasks for everyone which are doable in the given time. Once everyone is happy, the sprint then begins and all group members know what they need to work on.

At 12pm on Tuesday we meet again on Teams for our daily stand-up where we all discuss what we have done. Usually the day after the sprint kickoff not much progress has been made due to commitments in other modules. Any outstanding merge requests which have conflicts can be sorted in these meetings by the group to ensure we do not break the project. Any help anyone needs can also be addressed in the daily stand-ups as other group members may know how to fix. This process repeats every weekday; we do not meet on weekends for obvious reasons. Nearing the end of the bi-weekly sprint, Member A will check up on everyone who has outstanding tasks to make sure they are doing okay, and seeing if there is any problems with the tasks. As members work through tasks they can move them into the correct section. For example, Member B has started a task so it goes from "To Do' → "In Progress" etc. By having the *Jira* board, it is clear where everyone is at and allows for understanding of others' tasks, see Figure 3.1 for an example.


Projects / CS3099 Group A1 Project

Sprint 8

Epic

TO DO 3 ISSUES	IN PROGRESS 3 ISSUES	IN REVIEW 3 ISSUES	DONE 20 ISSUES ✓
Matthew McIlree 5 issues			
Display backend errors properly on frontend. A1-170 MM	Progress Report sections A1-172 MM		Show what community a post belongs to on the post - for creating an aggregated moosfeed later A1-117 ✓ MM
			Change moosfeed to only subscribed communities A1-109 ✓ MM
			Refactor postviewer A1-152 ✓ MM
Nguyen Nguyen 5 issues			
Write API tests MINIMUM VIABLE PRODUCT A1-71	Change if statement for permission to decorator A1-122	Progress Report sections A1-174	Fix refresh taking the user into the void. A1-148 ✓
			Delete Account Feature A1-106 ✓
Nina Boothby 6 issues			
Improve comments viewer A1-156			Progress Report sections A1-175 ✓
			Add list of posts from a user on their profile A1-144 ✓
			Refactor code on front end into folders A1-126 ✓
			Improve Welcome Page and add accessibility/help functionality to frontend A1-158 ✓
			Null parameter in profile page url A1-154 ✓
Rhona McCracken 6 issues			

Figure 3.1: An example of a typical Jira board for task management.

In line with Test Driven Development, if during coding, we encounter any problems, we alert the whole group to this issue using our group chat. Sometimes, other members are aware of these errors and have actually fixed them in their recent commit. However, the majority of the time this is not the case and that problem will then be added as a current task to a member, to ensure it gets fixed quickly, or if it is a low priority issue, added to our task backlog.

Some weeks we have scheduled pair programming. This is usually used if a member is struggling and needs help, or if there is a large feature being implemented and it needs two people to ensure it works correctly. Sometimes pair programming happens over Teams where one member will share their screen and code and the other can see and help. Pair programming has worked well for our team, however due to the pandemic we have rarely managed to meet in person. Pair programming would be much easier in a computer lab setting, and we would also have used it more if we could all have been together more often.

Author: 180017218

3.1.3 Tools Used

As mentioned above we have used many different tools to allow us to use Agile and Scrum. Each tool has a completely different function which has allowed us to succeed as a group.

1. **Jira** - Online software used to plan, track and manage agile software development projects which has been developed by Atlassian. This has been one of our most used tools and we all agree it has worked great for us.

2. **Confluence**: A web-based collaboration tool used to help teams collaborate and share knowledge. Confluence allows us all to have our team documents in one place and collaborate with each other. Confluence has been extremely helpful for us this year as we cannot write things on paper or use whiteboards: everything has to be online.

3. **Microsoft Teams**: We have used Teams for our sprint kickoffs, daily stand-ups, supervisor meetings and pair programming. The video calling feature on Teams has been extremely helpful and allowed to 'meet', even though we have never met face-to-face.

4. **Facebook Messenger**: Messenger has been our main mode of informal communication. When not in meetings we discuss ideas/problems/merge requests in our group chat. This allows us all to get back to each other quickly if we cannot hold a meeting.

Author: 180017218

3.2 Interaction with the Supergroup

The organisation of the Supergroup has had some constants and some evolving facets as the year has progressed. As stated in previous reports, we have:

- Chosen representatives from each group.
- A weekly meeting of representatives from each group on Microsoft teams.
- A Git repository containing documentation for our REST API: changes to this were initially manual, but became more formal through pull requests.
- Informal testing meetings between groups.
- Organised testing sessions with multiple groups towards the end of the project work period.

In general, we feel group A1 has taken an active role in the supergroup organisations. Our representative organised many of the initial meetings and set up the weekly slots on teams, including setting up polls to find out when other members were free, and chairing many of the meetings.

The finalised protocol is hosted at: https://fmckeogh.github.io/cs3099a-specification/.

Author: 180004835

3.3 Continuous Integration and Deployment

3.3.1 CI/CD Principles

In order to ensure that our mode of software development was streamlined and sustainable, as well as to practice effective methodologies of modern software engineering, we worked towards having a sophisticated Continuous Integration and deployment system. The ultimate aim is to preserve the integrity of the master (production) branch, and to always have a working system running in deployment.

The flow for the system was as follows:

- A developer creates a new branch from the master branch in our version control system named `devName/featureName`. Branching allows group members to try out ideas in code and make quick fixes without interfering with the work of others. By agreement, all new work must begin in its own branch.

- When the work in the branch is completed (a new feature set or fix is fully implemented) the branch is submitted for "merge". We specially requested from the Systems Administrators that a strict reviewing system was put in place for this process, so that it is impossible for a branch to be merged without approval from at least one other group member. At this review stage, suggestions and questions may be made by other group members.

- At each commit, the software is built and a relevant suite of tests (either client or server tests) is run, providing realtime feedback to the developer as to whether their code is making breaking changes or not. Once approved, we requested an additional protection be placed on the master branch, which is that the latest commit should have no pipeline failures before merging takes place.

- After the merge (including fixing any merge conflicts if necessary), the build and test stages are run again, along with a final third stage "deploy" which automatically incorporates the changes in a deployed version of our instance running in production mode on the school's host servers.

Author: 180004835

3.3.2 CI/CD Implementation

The pipeline was accomplished using Gitlab runners provisioned by the school. We have separate testing suites for the backend and the frontend. Selective testing was introduced to optimise the pipeline. This meant that only relevant tests are executed based on recent modifications. Additionally, prerequisites and dependencies were cached and are only updated when the requirements change. The application is automatically deployed on school's host servers (`https://cs3099user-a1.host.cs.st-andrews.ac.uk`) and handled by *tmux* sessions. This practice of continuous integration and deployment has been pivotal to the efficiency and effectiveness of our team.

Author: 180005905

3.4 Testing Summary

3.4.1 Testing Philosophy

In software development, testing is incredibly important, so we built server- and client-side testing into our CI/CD pipeline from the start of the project. We chose to use Automated Testing in our project to ensure that any breaking changes would be caught in the pipeline, identified by our tests before causing any problems in our deployed site. Although our testing was not always rigorous throughout development, we tried to add enough front-end and back-end tests as we progressed to ensure our application had no major bugs or deployment issues. We followed Test Driven Development as far as writing many small unit tests, writing tests concurrently with features, and running all tests on each iteration of the product, but we were not as strict as to insist on writing tests before implementing, as would be done in pure TDD.

In our final implementation we are satisfied that our tests are comprehensive enough to demonstrate that the features we have added work as expected and that any issues are caught and reported with user-friendly error messages. There is always room to improve testing but through the use of parameterised tests in the front end (testing routing and mocking API calls to ensure components render properly); and API tests and database tests in the back-end - we are satisfied that our testing has good coverage for our large application.

Author: 180017395

3.4.2 Front-end Testing

In practice, for our front end testing suites, we have used *Jest*, with the *enzyme* library for testing react components. To avoid unnecessary coupling of the frontend and backend tests, we made significant use of the "mocking" feature: essentially providing dummy data for our front end tests in place of actual API calls. For each "page" component: i.e. the top level component for each route, we have both LoggedIn and LoggedOut tests; the former checking that each component in question renders without crashing, and latter checking that every private route redirects to the login screen when the user is not authenticated.

We also perform more in depth testing on individual components, taking special care with components that must validate input from the user, such as the Login/SignUp forms and the Post/Community Creation forms. To ensure a wide variety of possible inputs is accounted for, we made use of the *jest-each* library which allows for customisable parameterised tests. The complete output from all 103 of our frontend tests can be found at Appendix D.

Author: 180004835

3.4.3 API Testing

The API tests were powered by testing tool *pytest*. The backend server is very extensive, therefore we had to divide the tests into multiple sections. The protocol tests were divided into internal protocol tests and supergroup protocol tests. Additionally, we had tests for our authentication system and federation between instances. This was achieved by mocking requests to other instances. To gauge the effectiveness of our tests, we measured the code coverage of our backend tests. We were able to achieve a comprehensive 90% test coverage. Appendix E shows output from our backend tests.

Author: 180005905

3.4.4 Database Testing

Unit testing was especially important for database query testing, as it allowed for a way to test features independently without the order of tests being significant. This is achieved using the *unittest* Python library which provides functionality to create and run the tests. Test data and order independence is achieved using setup and tear-down functions before and after each unit test. During set up, a new Flask app and database are created which are independent of the main application. During tear down, the new flask app and database are deleted.

Author: 180016547

3.4.5 Usability Testing

Due to time constraints we were unable to complete user-experience testing for the project. If we had been able to do this our testing methodology would have been to write clear step-by-step instructions for users to trial important actions and features in our site, such as creating an account, making a community and posting.

This may have followed a typical user flow, as outlined in our initial user experience design, created from our user stories. Following this, we would have constructed a clear, unbiased questionnaire to give participants to gain feedback on the usability, accessibility and effectiveness of the main features of our application. This testing process would have given us a solid basis for evaluating how successfully we met the user requirements and what needed to be improved.

Unfortunately, due to time restrictions and concerns around the ethics of this type of testing, we were unable to write this survey to collect feedback, however as a group we have completed rigorous manual testing. Of course, our group members are biased since we know how the application should work, but we are satisfied that combined with our functional testing and seeing members of the supergroup interacting with our deployed site, this is enough to demonstrate that we have successfully created useable features.

Author: 180017395

Chapter 4

Evaluation and Critical Appraisal

4.1 Comparison with Project Objectives

We believe we have met all of our crucial project objectives, as well as some "showstopper" nice-to-have features. The evaluation of our successful completion of the initial requirements is based on our functional testing and manual testing of the program.

In our initial report, we wrote a broad overview of what we would like to have achieved by the end of this project. In this section, we quote from this report and explain how we have fulfilled the objectives.

Basic Objectives

> "We envisage that our system would support an online community that represents a broad subcategory of experience within a particular university."

— Our final system is entirely open ended, allowing for the deployed instance to contain a set of related or unrelated communities, as the user base sees fit. To simplify terminology, we now refer to the overall community platform as our "instance", and the "subcommunities" as simply "communities".

> "Within each subcommunity, a user will be able to view a list of posts, and also create posts, which may be social or educational depending on the subcommunity's purpose."

— This has been fully implemented. The purpose of the community can be made clear through its title, description, and the assigned roles.

> "Within each post, users may also post comments which will be public to everyone viewing the post. Comments will be nested in hierarchical tree structure."

— Post commenting is fully implemented, although we placed restrictions on the hierarchical tree structure for simplicity, allowing just one level (comments with replies).

In summary, by November, we had a working project which allowed users to register to use the federated university site, we had functionality to post new articles on a given community, we had functionality to post comments on posts. This was all aided with our web-based user interface which was clear to navigate. We had also implemented federation by this point, with our instance able to propagate content between instances.

Medium Priority Objectives

> "Moderation rules set for a community/subcommunity by creators/appointed admin users."

— The user administration role levels, combined with the ability of community admins to edit other users' posts means community creators/admins can enforce whatever moderation policy they see fit for their community.

> "A user can change their account settings to select whether their profile info (name, email, etc.) are public to users in their community and/or public to users in other communities. Users can keep all details private except username if desired."

— This is fully implemented in our "private account" feature.

> "A user could join a community as a guest who can view posts from one community but not post or comment. (Similar to subscribing to a mailing list)."

— This is possible through our "guest" role in our community permissions system.

> "A user who can create a group within a subcommunity and set restrictions on who can join the group, user could set permissions so only they could add group members (e.g. for a module class, society group for paid members). A user who creates a group could add other users as admins who have same permissions to add new users to the group. User could set group permissions so some users can view the posts in the group and some members can edit posts (e.g. only committee members can post). General subcommunity space still available for public posts/comments on a topic (e.g. general society members can post here but not in group)."

— Again, all of this is possible through setting roles in our permissions system.

> "A user can see a summary of recent/relevant filtered posts on home page of any subcommunity/group."

— While this is not available on a per-community basis, we have an equivalent feature to this through the "Moosfeed" and community subscriptions.

> "For security users registering must use a university institution email account (users should join community listed for their university so that email can be authenticated). Any non-university users may be added as guests/anonymously to view public community-level posts only."

— This was one of the only high priority features we did not end up introducing into our product. We decided against implementing this since even though our product is university-centred, we wanted it to be accessible to everyone, and so limiting who can use it at this level was deemed to be unnecessary by the group.

Other "Nice-To-Have" Objectives:

> "By the end of this project we would like to have achieved more advanced functionality in addition to these minimum capabilities. Although there are a great many possibilities and examples available in the domain of social media systems, we would like to prioritise those that are most relevant to a university experience. "Polls" that anyone with access to a subcommunity can see and contribute to would be a desirable extension."

— As stated, one of our extended post types is the poll type, which could be used for a quick test or for gauging feedback.

"Preview of links to pictures/videos in posts."

— For images/links, this is available in normal text posts, as well as through markdown. Video previews are not supported (although it is simple to link to an external video streaming service).

"Bot to link posts for common questions."

— This was a feature we considered but ultimately decided to deprioritise.

"Whiteboard feature for class teaching – free form drawing."

— Our set of features that we consider to be of "exceptional quality" includes our whiteboard. The whiteboard allows people to create images and text and share them to a given community or download them for personal use. The whiteboard can also be used as a share screen functionality, several users can log on to a whiteboard and make changes which all users can see live and make changes and add as needed. This feature adds to the university environment as it would allow lecturers to draw content on the screen and for students to follow along, or for societies to brainstorm ideas and everyone can contribute. It consolidates Academoo as a place for collaboration and education.

Ultimately, our manual testing as well as our automated tests are what give us confidence our objectives have been achieved. We have all the functionality we aimed to implement, and we can verify the correctness of this functionality both by trying it for ourselves and by running our automated verification process.

Author: 180017218

4.2 Comparison with Related Work

We deliberately intended throughout our process that our finalised product would not be a "clone" of any existing solution. Nevertheless, the idea of a social news aggregator is one that is extremely common across many web and desktop applications, and so we have adopted many conventions and taken inspiration from existing work.

4.2.1 Reddit

Reddit is most certainly the largest influence on the way our platform is constructed. We mirror the idea of "subreddits" with our "communities" and also adopt an upvoting and downvoting system similar to Reddit's. The ability of voting systems to discern quality content is a subject of study, and our very simple algorithm (upvotes minus downvotes) is far less sophisticated than Reddit's. Even with a more sophisticated algorithm, according to a 2015 study, "there's a sort of "two-stage process" of popularity on Reddit and Hacker News. Many articles fail to receive any reasonable amount of attention after being submitted and fade into obscurity, regardless of quality. However, amongst the set of articles that receive a reasonable amount of attention, relative popularity is a strong indication of relative quality" [Stoddard(2015)]

For our part, we do not take any measures to counteract the fact that posts that have been around for longer will naturally have garnered more upvotes, but may not necessarily be of higher quality than brand-new posts with a low number of votes. This is an issue we could tackle with more time, as well as testing using a much larger user-base.

For simplicity of interface, we also excluded some of Reddit's less intuitive features. For example, our comment system follows the two-layered "comments" and "replies" approach, as opposed to the infinite nesting of reddit, which can be intimidating for a new user. As mentioned on the SpyreStudios design magazine "Heavily trafficked websites that garner hundreds of comments will certainly benefit from threaded replies. But if you run a simple blog it may be easier to just leave individual comments as they are."[Rocheleau(2015)] We opted for a happy medium, as suggested on a "CodingHorror" blog: "What matters is that we allow one level of replies and that's it." [Atwood(2012)]

4.2.2 Facebook

Our user profile system, with biographies, privacy settings and avatars draws some inspiration from Facebook. Since our academic social news website emphasises all of the different kind of relationships users may have, through shared participation in communities (e.g. student/lecturer, society organiser/lecturer, friends/colleagues), we opted not to emphasise the social network aspect by explicitly linking users, for example by creating a friends or followers feature.

4.2.3 Microsoft Teams

Our idea of "communities" is also somewhat similar to the "Teams" of Microsoft Teams. MS Teams has certainly proved useful for online teaching and working in a remote university environment, although we made the conscious decision to aim Academoo at being more of a middle ground between working and socialising in University context. Our shared whiteboard feature (for which there is no parallel in Reddit or Facebook), is somewhat similar to a new feature of MS Teams meant to aid online teaching.

Author: 180004835

4.3 Comparison with Plans

Throughout the project, we have endeavoured to follow our plans as closely as we could, but with various circumstances: feedback, time constraints, new knowledge, personal, meant we had to make changes to our original plans as we developed Academoo throughout the year.

As detailed previously, by the middle of semester 1 we had set ourselves a list of functional requirements that we wanted to implement set by priority (high/medium/low) as an indication of what order to implement them in and we are pleased that we implemented the vast majority of these requirements throughout the year.

Our initial priorities in October were to have the basic functionality for the MVP implemented. This included, user registration, user profiles, server authentication for log-in, community creation, posting on communities, commenting in communities, see communities hosted on other servers, comment on other servers instances. The majority of these 'high' priority requirements were implemented for the MVP as initially wanted, however due to time constraints a few were not. We therefore adapted our plans to include voluntary work over the Christmas period. Features included in this plan were: richer details in user profiles (more than just username) and images/links to be viewed on posts correctly, and a crucial refocussing on the user roles system, which despite emphasising in our initial report, we had neglected for the second report. These adaptations put us back on track to achieving our features for the final product in April.

After Christmas and into Semester 2 we then worked on extending the functionality of our product from the MVP, cultivating the distinctiveness of our site. Our medium and some low requirements discussed in our report were implemented in this timeframe. We set up moderation rules for communities, and implemented the access control that we thoroughly planned out in detail, following our restructuring of priorities. We also implemented the public/private profile, with a user able to select what type of privacy they would like to have, in keeping with our initial plans for this feature (only a user's username shown if private). From our medium priority requirements we also implemented the ability to edit posts and delete them, and the summary of posts on a page to which a user has subscribed.

At this point, we again had to consider a refocussing of priorities, after some discussion with our supervisor. We had to decide whether it would be more beneficial to continue adding greater diversity of "basic" features, or to pursue in depth one of the 'Nice-To-Haves'. The main priority requirement we selected, our showstopper, is the whiteboard feature for class teaching using free form drawing. As described previously, we see this a key feature for a university community where the lecturer could write equations/brainstorm/images/text to the class for more thorough understanding of a given topic.

Some of the features from semester 2 plan we wanted to implement but didn't include: direct messaging, wikibots, video calling and customisable forms. These features were big stretches, and their omission is only a minor deviation from our plans. When discussing them, we were honest in our assessment that we would only be able to implement one or two at most, and on research we decided that whiteboards and polls would be the best to implement for our group.

This information and analysis shows that we have followed our plans closely where feasible, but remained adaptable throughout the year as we developed Academoo. We prioritised some features ahead of others in development to make sure that we secured the basics before implementing other features. The definitive summary of when tasks were carried out is given in our weekly progress reports, included in Appendix C.

Author: 180017218

Chapter 5

Conclusions

5.1 Key Achievements

We consider the following features and processes to be our key achievements of exceptional quality:

- Robust and methodical CI/CD system.
- Comprehensive automated tests.
- Highly organised group working methodology.
- Clear and extendible user interface.
- Carefully considered extended features for an academic environment (whiteboards, polls, user access control).
- Full participation in required features of the Supergroup protocol.

5.2 Learning Outcomes

Ultimately this project has been an excellent learning experience in the field of software development. We have all gained experience in organising ourselves as an independent software engineering group, as well as organising interaction between groups. We are all now more adept with issue tracking, version control, reviewing code, and co-ordinating multiple people working on the same set of files.

At the outset of the project, we all listed our experience levels in order to determine which technologies we should use, and every member of the group has taken time to independently learn the required core technologies they were not previously familiar with: (*React, Flask, SQLAlchemy* etc.).

Author: 180004835

5.3 Future Development and Improvements

With more time, Academoo could be further augmented with more of the standard features that might be expected from a social media site. We would start by introducing tags and filters as a further way to subgroup posts, since we already had the infrastructure for this available in the backend, but decided to deprioritise it in the frontend due to time constraints, and to allow for more testing of existing features.

With the "live" aspect of the whiteboard feature implemented as a demonstration of the sockets capability, we could also move into further "live" features, for example live, direct messages, live notifications, and video livestreaming.

As evidenced by Microsoft Teams, clearly video calling is a crucial aspect of remote University life, and we would look for ways to integrate this seamlessly into the social network paradigm. If University-taught modules each had their Academoo own community, making use of our whiteboard and polls features to enhance the learning experience, then we could incorporate a live video feed of the lecturer into the community itself to complete the experience.

Finally, there are some aspects of our existing implementation that we would look to improve even further. It is always possible to perform even more in-depth testing to try to catch more subtle bugs in the program. We might also consider consolidating the fetching framework for our API, so that fetches are made centrally, and then components would only deal with the fetched data rather than making the fetches ourselves. For scalability reasons, we might also consider migrating our database system to a NoSQL setup at some future point.

Author: 180004835

5.4 Conclusion

Overall, we are pleased with our final product, satisfied with how we have worked as a Team, and have enjoyed putting together this very substantial and complex piece of software. We are very proud of our design, our diverse range of significantly extended features, as well as our very successful group working and CI/CD methodologies.

While many aspects have been challenging, the experience afforded to us through overcoming these challenges has been invaluable.

Author: 180004835

Bibliography

[Atwood(2012)] Atwood, J., 2012: Web discussions: Flat by design. URL https://blog.codinghorror.com/web-discussions-flat-by-design/.

[Black(2008)] Black, P. E., 2008: Levenshtein distance. U.S. National Institute of Standards and Technology, URL https://xlinux.nist.gov/dads/HTML/Levenshtein.html.

[Cartwright(2020)] Cartwright, B., 2020: The designer's guide to color theory, color wheels, and color schemes. URL https://blog.hubspot.com/marketing/color-theory-design.

[Rocheleau(2015)] Rocheleau, J., 2015: How to design readable comment threads. URL https://spyrestudios.com/howto-design-comment-threads/.

[Stoddard(2015)] Stoddard, G., 2015: Popularity and quality in social news aggregators: A study of reddit and hacker news. *Proceedings of the 24th International Conference on World Wide Web*, Association for Computing Machinery, New York, NY, USA, 815–818, WWW '15 Companion, doi:10.1145/2740908.2742470, URL https://doi.org/10.1145/2740908.2742470.

Appendices

Appendix A

User Stories

- **Lea** is a 2nd year Biology student enrolled in BL2300, BL2301, BL2305 and CH2601 at the University of St Andrews. She would like to discuss aspects of these modules with other students who are enrolled and also hear general relevant announcements from the schools of Biology and Chemistry. Additionally, Lea is an active member of the University Biotech society and wants to keep in touch with other members, as well as keeping up to date with announcements from the organisers.
- **Urho** is the father of a new 1st year student at the University of St Andrews. He would like to keep up to date with university announcements relevant to parents (e.g. accommodation contracts, coronavirus updates), but would prefer not to enter any personal details on the internet as he is concerned about his privacy.
- **Quintin** is the events manager of the University of St Andrews Fine Food and Dining Society. He would like an easy-to-use public platform to display general information for non-members, but also a more exclusive platform to advertise his events to members only (avoiding gate-crashers).
- **Benita** is a professor of International Relations at the University of St Andrews. She would like a platform share some of her unpublished written material with her IR3039 module class. However, she is concerned about her original research being plagiarised, and so would like assurance that the material will not be available to anyone not enrolled in the module.
- **Gordon** is a Systems Administrator in the St Andrews School of Computer Science. He would like an informal forum to quickly answer queries

related to the school's computer systems. But since he receives a large number of queries, he would also be interested in creating a bot that posts links to the Systems Wiki based on keywords in the questions.

- **Iona** is a 4th year history student and skilled unicyclist at the University of St Andrews who is interested in starting a Circus Skills society. She would like a public platform to gauge general interest in such a society. In the future she ideally hopes to run a platform where she can share general information, post links to resources, and perhaps even live stream video tutorials for juggling.

Appendix B

User Roles Design

Each server must have admins to set moderation rules and monitor community creation. There are two types of instance-wide users: admin and regular user.

Instance-Wide Admin Permissions:

- Admin page to set moderation rules (e.g. content allowed on site, permissions for specific users, banning users etc.)
- Approve or shut down communities
- Verify communities? (i.e a STACS community may be verified as a recognised society but a study group community may be private).
- Block content from another instance
- Delete inappropriate posts or comments
- Override community admins' choices for assigned user groups
- Remove community admins permissions
- All regular user permissions

Regular users can login, update their profile, make communities (need to apply for approval for a verified community), make posts/comments in communities where they have permission to do so (see community-wide permissions) and delete their own posts and comments.

Community-Wide Permissions:

4 user groups that can be assigned by the owner (community creator) or appointed admins of communities.

User groups:

- Community Creator/Admin
- Contributor – can make posts and comment on posts in community
- General member – can comment on posts but cannot create top-level posts in the community
- Guest – can only view posts and comments in community, cannot post or comment

Actions within a community:

Action	**Guest**	**General Member**	**Contributor**	**Creator/Admin**
View posts	Yes	Yes	Yes	Yes
View comments	Yes	Yes	Yes	Yes
Add (top-level) posts	No	No	Yes	Yes
Add comments	No	Yes	Yes	Yes
Delete posts	No	No	Yes (only posts this user added)	Yes (by any user)
Delete comments	No	Yes (only comments this user added)	Yes (only comments this user added)	Yes (by any user)
Edit posts/comments	No	Yes (only comments this user added)	Yes (only posts/comments this user added)	Yes (only posts/comments this user added)

Appendix C

Weekly Progress Reports

Week 2 Progress Report:

Everyone

- Met and was introduced to each other.
- Discussed specification.
- Discussed availability for meetings.
- Agreed to use Jira/Confluence.
- Agreed on Sprint structure.
- Agreed to meet next Monday for full planning meeting.

Week 3 Progress Report:

Everyone

- Attended sprint planning meeting.
- Attended daily stand-ups.
- Met supervisor.
- Wrote up experience.
- Wrote up general brainstorming.

180017218

- Elected Scrum-Master.
- Set up Jira/Confluence.
- Allocated tasks on Jira.

180017395

- Elected product owner.
- Wrote up federation plan

180004835:

- Elected Supergroup Representative.
- Arranged first meeting with supervisor
- Met with supergroup.

Week 4 Progress Report:

Everyone:

- Attended general planning meeting.
- Attended daily stand-ups.
- Wrote basic requirement priorities.
- Set up GitLab with SOCKS proxy.
- Discussed relative benefits of possible languages and technologies.

180017218:

- Wrote Scrum section of progress report.
- Summarised semester plan.

180017395:

- Summarised functional / non-functional requirements.
- Wrote requirements section of progress report.

180004835:

- Summarised protocol requirements.
- Wrote descriptive user stories.
- Formatted progress report.

180016547:

- Wrote broad overview section of progress report.

180005905:

- Wrote languages and technologies section of progress report.

Week 5 Progress Report:

Everyone:

- Proofread progress report and submitted.
- Attended sprint planning meeting.
- Attended daily stand-ups.

180017218:

- Elected Product Owner.
- Followed React Tutorials.

180017395:

- Elected Scrum Master.
- Followed React Tutorials.

180004835:

- Attended supergroup meeting.
- Followed React Tutorials.

180016547:

- Followed Postgresql and MongoDB tutorials.
- Experimented with databases.

180005905:

- Wrote example protocol Swagger.

Week 6 Progress Report:

Everyone:

- Attended general planning meeting.
- Attended daily stand-ups.
- Agreed on name.
- Agreed on commit message conventions

180017218:

- Created basic react app structure.

180017395:

- Created mock-up designs for web interface.
- Create wireframe for page navigation.

180004835:

- Attended Supergroup meeting.
- Created basic react app structure.
- Brainstormed name ideas.
- Drew logo.
- Set up front end CI.

180016547:

- Created Pros and Cons Database comparison.
- Created skeleton database.
- Implemented initial database queries.

180005905:

- Set up back-end CI.
- Cleaned up GitLab structure.
- Created flask app structure.
- Implemented test API endpoints.

Week 7 Progress Report:

Everyone:

- Attended sprint planning meeting.
- Attended daily stand-ups.

180017218:

- Implemented stub login page.

180017395:

- Implemented Post display from JSON
- Added tests.

180004835:

- Elected Scrum master.
- Attended supergroup meeting.
- Implemented sample navigation bar and welcome message.
- Added tests.

180016547:

- Implemented more database queries.

180005905:

- Elected Product Owner.

Week 8 Progress Report:

Everyone:

- Attended general planning meeting.
- Attended daily stand-ups.
- Agreed on naming conventions.

180017218:

- Implemented login page.
- Implemented signup page.

180017395:

- Refactored post display into separate components.
- Implemented page navigation via React Routing.
- Added functionality to navigation bar.
- Improved CSS Themes.
- Added tests.

180004835:

- Attended supergroup meeting.
- Implemented correct navigation bar.
- Implemented create post form.
- Implemented fetch requests for post viewer.
- Added tests.

180016547:

- Added user authentication to database.
- Fixed database bugs.
- Added auto-generated post ids.

180005905:

- Implemented internal API endpoints
- Implemented supergroup protocol.
- Deployed backend on host servers.
- Deployed front end on host servers.
- Added authentication for API.
- Optimised CI/CD pipelines.

Week 9 Progress Report:

Everyone:

- Attended general planning meeting.
- Attended daily stand-ups.
- Agreed on naming conventions.

180017218:

- Completed log-in sign-up form implementation.
- Added error messages to login/sign up forms.
- Added username to login form.

180017395:

- Fixed routing tests.
- Added further tests.

180004835:

- Attended supergroup meeting.
- Added routing from login form.
- Implemented comment creation
- Improve post appearance.
- Added user profile information.

180016547:

- Elected Product Owner.
- Added user authentication to database.
- Fixed database bugs.
- Added auto-generated post ids.

180005905:

- Elected Scrum Master.
- Implemented community creation
- Implemented community switching.
- Fix backend bugs.
- Improved comments rendering.
- Improve navbar appearance for logged out users.

Week beginning 4/01/21:

180004835:

- Fixed post redirect
- Added sign up form validation
- Added explore page
- Added "Typeahead" component for user search

180017395:

- Fixed login redirect

Week beginning 11/01/21:

180004835:

- Added Community creator validation

Week beginning 18/01/21:

180004835:

- Extra frontend tests

180017395/180004835 (pair programming):

- Added basic backend framework for user roles

180017395:

- Added Manage Community Page
- Added API endpoints for user roles

180017218:

- Added Bio to profile page

Sprint 7

Week beginning 25/01/21:

180005905:

- Fixed token expiration
- Added change password feature

180017218:

- Added basic profile picture

180017218/180017395 (pair programming):

- Added public profile page

180016547:

- Fixed database schema for user roles
- Updated database actions to match supergroup protocol

180004835:

- Improved Posts Layout
- Fix backend errors

180017395:

- Added Default user roles
- Fix backend errors

Week beginning 1/02/21:

180005905:

- Deployed to Pseudo-Accounts
- Fixed ports issue

180017395:

- Added privacy settings
- Implemented edit/deleting posts

180004835:

- Added basic community subscription
- Refactored frontend code

180016547:

- Fix route registering issues
- Fix filtered post ordering
- Fix HTTP request headers

180017218:

- Fixed testing issues
- Added clickable usernames to posts

Sprint 8 180005905:

- Fixed page refresh deployment issue

180017395:

- Added instance-wide administrators
- Implemented Markdown posts
- Researched Whiteboards feature

180004835:

- Fixed testing issues
- Fixed Subscription issue
- Researched Video Streaming feature

180017218:

- Refactored front end into folders
- Added Help pages

180016547:

- Added backend JSON validation

Week beginning 22/2/2021

180005905:

- Wrote report sections

180017395:

- Wrote report sections

180004835:

- Wrote report sections
- Compiled report
- Fixed sidebar

180017218:

- Wrote report sections
- Added posts to profile
- Improved welcome page content

180016547:

- Wrote report sections
- Added header validation and support in federation

Week beginning 1/3/2021

180005905: 180017395: 180004835:

- Added a check for none user
- Update instance list to known working instances
- Fixed federation problems

180017218:

- Refactored the delete account feature 180016547:
- Fixed headers
- Fixed filtered posts for children
- Updated out of date JSON schema

Week beginning 8/3/2021

180005905: 180017395:

- Started creating basic sketchpad

180004835:

- Created fetch mock test file
- Began creating basic sketchpad
- Added tests for frontend

180017218:

- Added in password check to delete account
- Fixed error messages

180016547:

- Added upvote/downvote to DB
- DB tests

Week beginning 15/3/2021

180005905: 180017395:

- Fixed errors display on front end for viewing posts
- Added remove selection tool for whiteboards
- Added mathjax to markdown
- Added username instead of 'yoo' in header bar
- Private account no longer displays users posts

180004835:

- Fixed backend who voted what
- Added front end upvote and downvotes

180017218:

- Added community posted to profile posts
- Added backend functionality to check password for delete account

180016547:

- Expanded JSON validation to include error messages

Week beginning 22/3/2021

180005905:

- Added endpoint for public keys
- Added digital signature functionality
- Signature validation for supergroup routes
- Cleaned code and attached error messages
- Added backend route to toggle security
- Backend fix cannot reach host

180017395:

- Wrote front end tests
- Added moderators can disable accounts
- Added sitewide permissions for editing and deleting posts, changing user roles and bug fixes

180004835:

- Added vote sorting for upvote /downvote
- Further Postcreator tests and more authfetch mocks
- Fixed post deletion for admins
- Made loading screen nicer

180017218: 180016547:

- Added extra DB condition to check for default role
- Fixed community validation

Week beginning 29/3/2021

180005905:

- Wrote API tests
- Fixed backend timeouts

180017395:

- Added backend socket support for rooms feature
- Added buttons for sharing updates on whiteboard
- Send invite buttons on Whiteboard

180004835:

- Fixed security problems
- Added external subscriptions
- Added backend socket support for rooms feature

180017218:

- Added reply box and accordion functionality
- Fixed password regex
- Commented half of front end
- Cleaned up front end code

180016547:

- Added comments to backend

Week beginning 5/4/2021

180005905:

- Wrote final report sections
- Fixed post deletion problems
- Added autogenerated profile pictures
- Added polls
- Added search bar

180017395:

- Wrote final report sections
- Added comments for half of front end
- Added notifications in whiteboard for users joining/leaving
- Changed welcome page, background colour and added search button

180004835:

- Fixed replies for a comment on toggle, than trying to get all at once
- Wrote final report sections
- Fixed post time being off by an hour

180017218:

- Wrote final report sections
- Added comments nesting to deployed
- Added frontend tests

180016547:

- Wrote final report sections
- Added more DB tests

Week beginning 12/4/2021

180005905:

- Added extra sections to report after feedback

180017395:

- Merged live sketch to deployed
- Added extra sections to report after feedback

180004835:

- Merged live sketch to deployed
- Added extra sections to report after feedback
- Fixed signature problem

180017218:

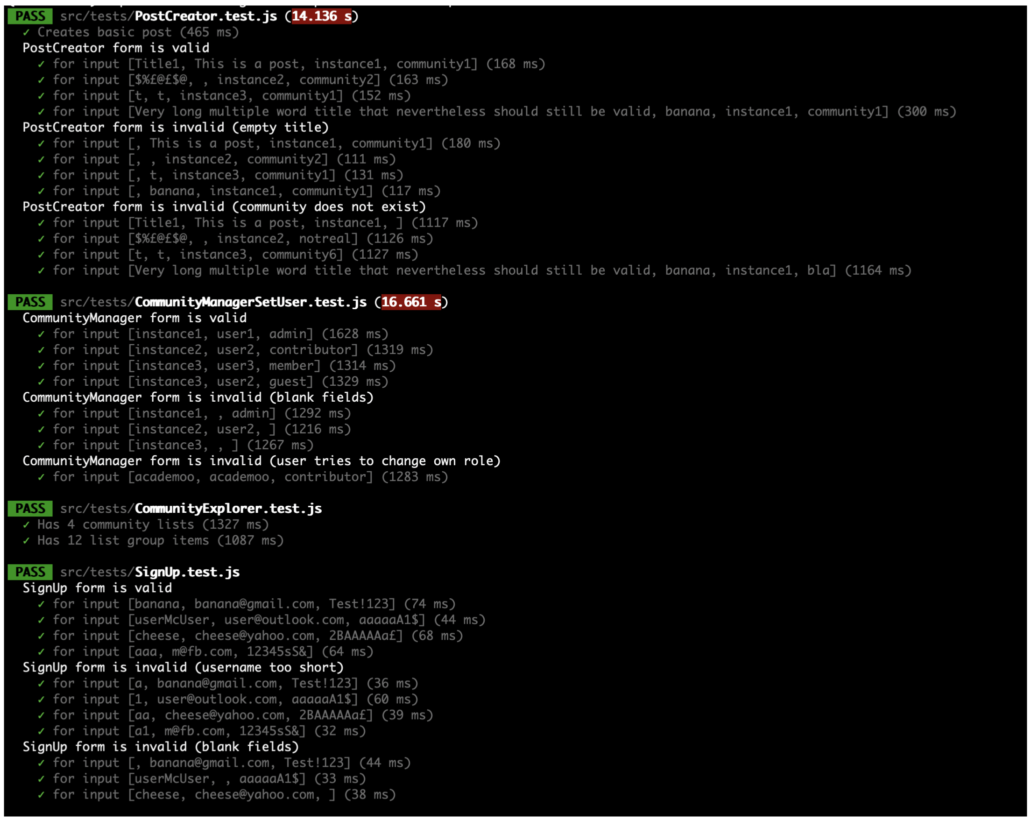
- Added extra sections to report after feedback

180016547:

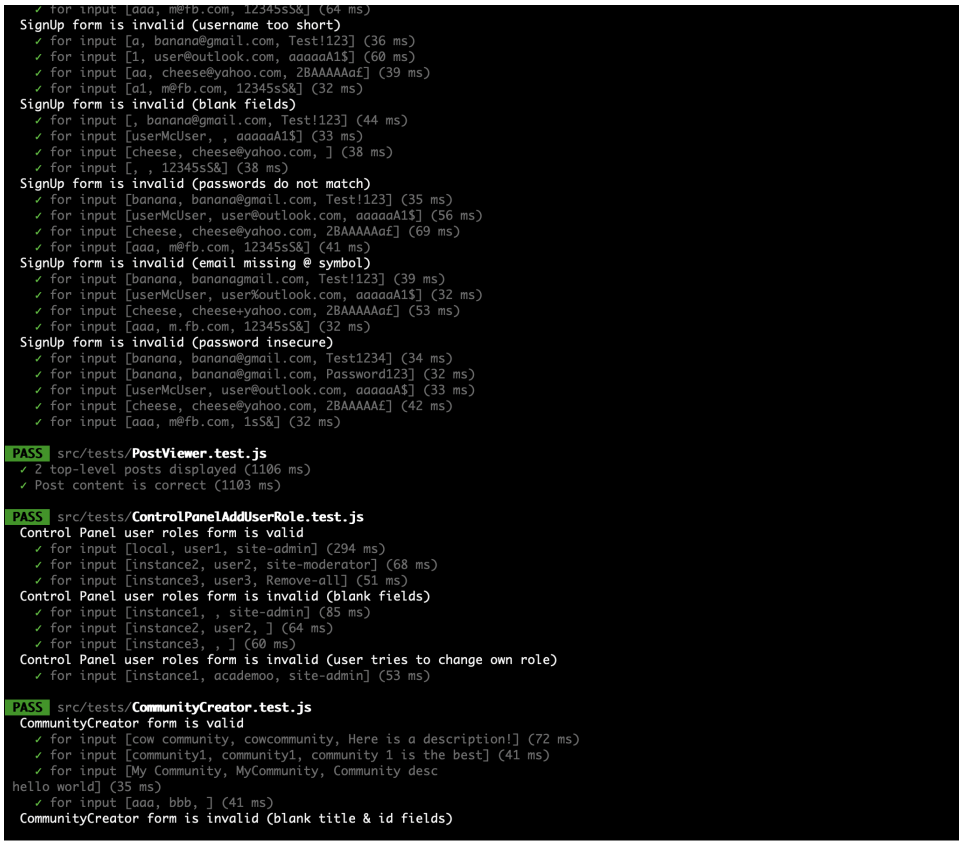
- Added extra sections to report after feedback

Appendix D

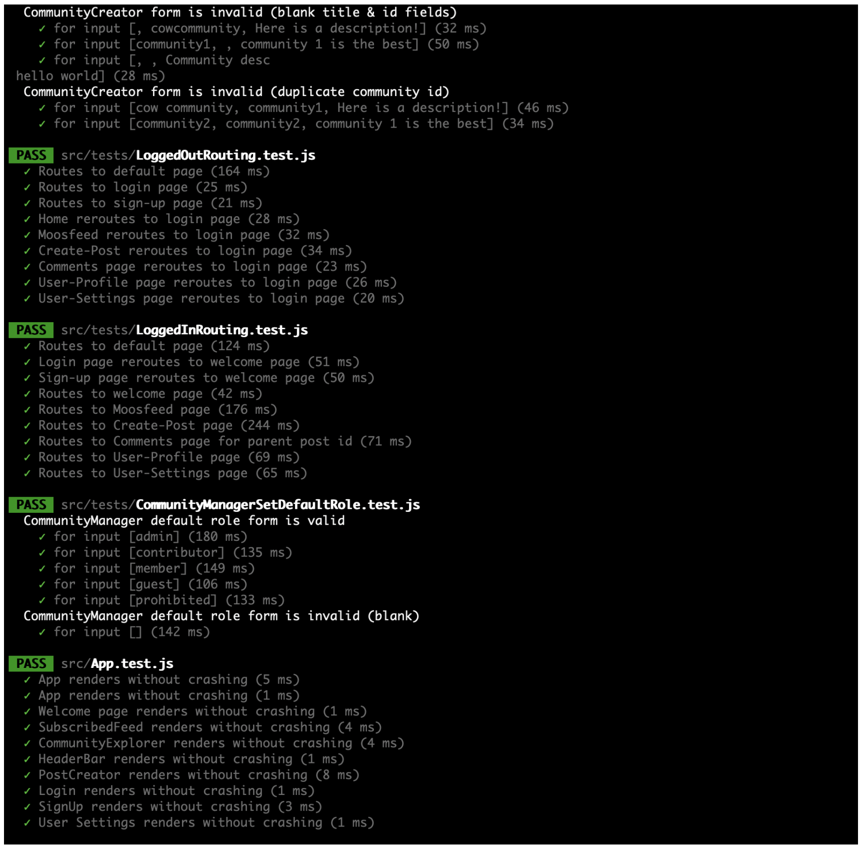
Frontend Test Summary



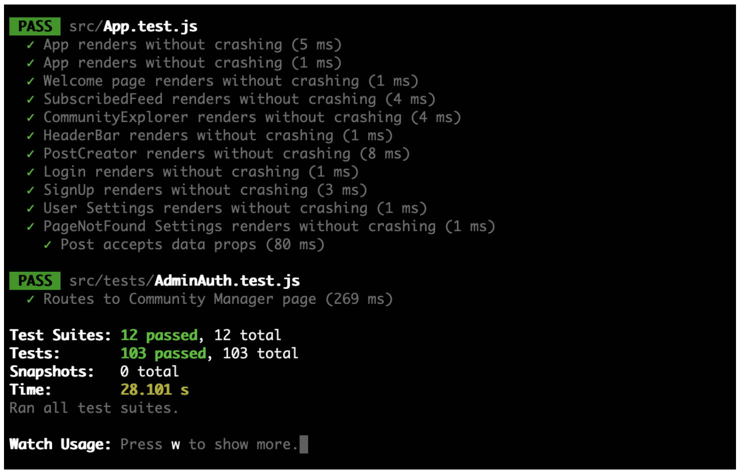
```
PASS  src/tests/PostCreator.test.js (14.136 s)
  ✓ Creates basic post (465 ms)
  PostCreator form is valid
    ✓ for input [Title1, This is a post, instance1, community1] (168 ms)
    ✓ for input [$%£@£$@, , instance2, community2] (163 ms)
    ✓ for input [t, t, instance3, community1] (152 ms)
    ✓ for input [Very long multiple word title that nevertheless should still be valid, banana, instance1, community1] (300 ms)
  PostCreator form is invalid (empty title)
    ✓ for input [, This is a post, instance1, community1] (180 ms)
    ✓ for input [, , instance2, community2] (111 ms)
    ✓ for input [, t, instance3, community1] (131 ms)
    ✓ for input [, banana, instance1, community1] (117 ms)
  PostCreator form is invalid (community does not exist)
    ✓ for input [Title1, This is a post, instance1, ] (1117 ms)
    ✓ for input [$%£@£$@, , instance2, notreal] (1126 ms)
    ✓ for input [t, t, instance3, community6] (1127 ms)
    ✓ for input [Very long multiple word title that nevertheless should still be valid, banana, instance1, bla] (1164 ms)

PASS  src/tests/CommunityManagerSetUser.test.js (16.661 s)
  CommunityManager form is valid
    ✓ for input [instance1, user1, admin] (1628 ms)
    ✓ for input [instance2, user2, contributor] (1319 ms)
    ✓ for input [instance3, user3, member] (1314 ms)
    ✓ for input [instance3, user2, guest] (1329 ms)
  CommunityManager form is invalid (blank fields)
    ✓ for input [instance1, , admin] (1292 ms)
    ✓ for input [instance2, user2, ] (1216 ms)
    ✓ for input [instance3, , ] (1267 ms)
  CommunityManager form is invalid (user tries to change own role)
    ✓ for input [academoo, academoo, contributor] (1283 ms)

PASS  src/tests/CommunityExplorer.test.js
  ✓ Has 4 community lists (1327 ms)
  ✓ Has 12 list group items (1087 ms)

PASS  src/tests/SignUp.test.js
  SignUp form is valid
    ✓ for input [banana, banana@gmail.com, Test!123] (74 ms)
    ✓ for input [userMcUser, user@outlook.com, aaaaaA1$] (44 ms)
    ✓ for input [cheese, cheese@yahoo.com, 2BAAAAAa£] (68 ms)
    ✓ for input [aaa, m@fb.com, 12345sS&] (64 ms)
  SignUp form is invalid (username too short)
    ✓ for input [a, banana@gmail.com, Test!123] (36 ms)
    ✓ for input [1, user@outlook.com, aaaaaA1$] (60 ms)
    ✓ for input [aa, cheese@yahoo.com, 2BAAAAAa£] (39 ms)
    ✓ for input [a1, m@fb.com, 12345sS&] (32 ms)
  SignUp form is invalid (blank fields)
    ✓ for input [, banana@gmail.com, Test!123] (44 ms)
    ✓ for input [userMcUser, , aaaaaA1$] (33 ms)
    ✓ for input [cheese, cheese@yahoo.com, ] (38 ms)
```



```
    ✓ for input [aaa, m@fb.com, 12345sS&] (64 ms)
  SignUp form is invalid (username too short)
    ✓ for input [a, banana@gmail.com, Test!123] (36 ms)
    ✓ for input [1, user@outlook.com, aaaaaA1$] (60 ms)
    ✓ for input [aa, cheese@yahoo.com, 2BAAAAAa£] (39 ms)
    ✓ for input [a1, m@fb.com, 12345sS&] (32 ms)
  SignUp form is invalid (blank fields)
    ✓ for input [, banana@gmail.com, Test!123] (44 ms)
    ✓ for input [userMcUser, , aaaaaA1$] (33 ms)
    ✓ for input [cheese, cheese@yahoo.com, ] (38 ms)
    ✓ for input [, , 12345sS&] (38 ms)
  SignUp form is invalid (passwords do not match)
    ✓ for input [banana, banana@gmail.com, Test!123] (35 ms)
    ✓ for input [userMcUser, user@outlook.com, aaaaaA1$] (56 ms)
    ✓ for input [cheese, cheese@yahoo.com, 2BAAAAAa£] (69 ms)
    ✓ for input [aaa, m@fb.com, 12345sS&] (41 ms)
  SignUp form is invalid (email missing @ symbol)
    ✓ for input [banana, bananagmail.com, Test!123] (39 ms)
    ✓ for input [userMcUser, user%outlook.com, aaaaaA1$] (32 ms)
    ✓ for input [cheese, cheese+yahoo.com, 2BAAAAAa£] (53 ms)
    ✓ for input [aaa, m.fb.com, 12345sS&] (32 ms)
  SignUp form is invalid (password insecure)
    ✓ for input [banana, banana@gmail.com, Test1234] (34 ms)
    ✓ for input [banana, banana@gmail.com, Password123] (32 ms)
    ✓ for input [userMcUser, user@outlook.com, aaaaaA$] (33 ms)
    ✓ for input [cheese, cheese@yahoo.com, 2BAAAAA£] (42 ms)
    ✓ for input [aaa, m@fb.com, 1sS&] (32 ms)

PASS  src/tests/PostViewer.test.js
  ✓ 2 top-level posts displayed (1106 ms)
  ✓ Post content is correct (1103 ms)

PASS  src/tests/ControlPanelAddUserRole.test.js
  Control Panel user roles form is valid
    ✓ for input [local, user1, site-admin] (294 ms)
    ✓ for input [instance2, user2, site-moderator] (68 ms)
    ✓ for input [instance3, user3, Remove-all] (51 ms)
  Control Panel user roles form is invalid (blank fields)
    ✓ for input [instance1, , site-admin] (85 ms)
    ✓ for input [instance2, user2, ] (64 ms)
    ✓ for input [instance3, , ] (60 ms)
  Control Panel user roles form is invalid (user tries to change own role)
    ✓ for input [instance1, academoo, site-admin] (53 ms)

PASS  src/tests/CommunityCreator.test.js
  CommunityCreator form is valid
    ✓ for input [cow community, cowcommunity, Here is a description!] (72 ms)
    ✓ for input [community1, community1, community 1 is the best] (41 ms)
    ✓ for input [My Community, MyCommunity, Community desc
hello world] (35 ms)
    ✓ for input [aaa, bbb, ] (41 ms)
  CommunityCreator form is invalid (blank title & id fields)
```



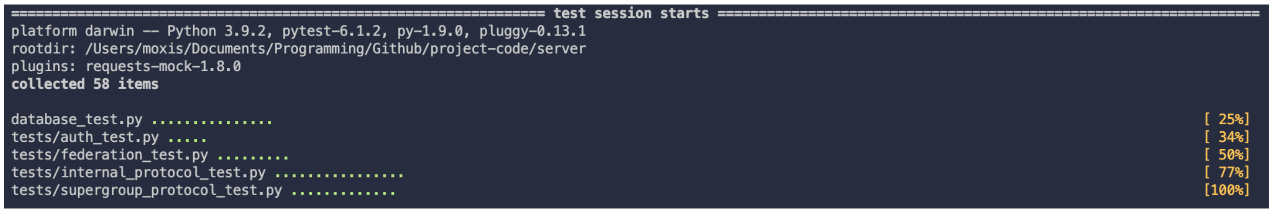
```
  CommunityCreator form is invalid (blank title & id fields)
    ✓ for input [, cowcommunity, Here is a description!] (32 ms)
    ✓ for input [community1, , community 1 is the best] (50 ms)
    ✓ for input [, , Community desc
hello world] (28 ms)
  CommunityCreator form is invalid (duplicate community id)
    ✓ for input [cow community, community1, Here is a description!] (46 ms)
    ✓ for input [community2, community2, community 1 is the best] (34 ms)

 PASS  src/tests/LoggedOutRouting.test.js
  ✓ Routes to default page (164 ms)
  ✓ Routes to login page (25 ms)
  ✓ Routes to sign-up page (21 ms)
  ✓ Home reroutes to login page (28 ms)
  ✓ Moosfeed reroutes to login page (32 ms)
  ✓ Create-Post reroutes to login page (34 ms)
  ✓ Comments page reroutes to login page (23 ms)
  ✓ User-Profile page reroutes to login page (26 ms)
  ✓ User-Settings page reroutes to login page (20 ms)

 PASS  src/tests/LoggedInRouting.test.js
  ✓ Routes to default page (124 ms)
  ✓ Login page reroutes to welcome page (51 ms)
  ✓ Sign-up page reroutes to welcome page (50 ms)
  ✓ Routes to welcome page (42 ms)
  ✓ Routes to Moosfeed page (176 ms)
  ✓ Routes to Create-Post page (244 ms)
  ✓ Routes to Comments page for parent post id (71 ms)
  ✓ Routes to User-Profile page (69 ms)
  ✓ Routes to User-Settings page (65 ms)

 PASS  src/tests/CommunityManagerSetDefaultRole.test.js
  CommunityManager default role form is valid
    ✓ for input [admin] (180 ms)
    ✓ for input [contributor] (135 ms)
    ✓ for input [member] (149 ms)
    ✓ for input [guest] (106 ms)
    ✓ for input [prohibited] (133 ms)
  CommunityManager default role form is invalid (blank)
    ✓ for input [] (142 ms)

 PASS  src/App.test.js
  ✓ App renders without crashing (5 ms)
  ✓ App renders without crashing (1 ms)
  ✓ Welcome page renders without crashing (1 ms)
  ✓ SubscribedFeed renders without crashing (4 ms)
  ✓ CommunityExplorer renders without crashing (4 ms)
  ✓ HeaderBar renders without crashing (1 ms)
  ✓ PostCreator renders without crashing (8 ms)
  ✓ Login renders without crashing (1 ms)
  ✓ SignUp renders without crashing (3 ms)
  ✓ User Settings renders without crashing (1 ms)
```



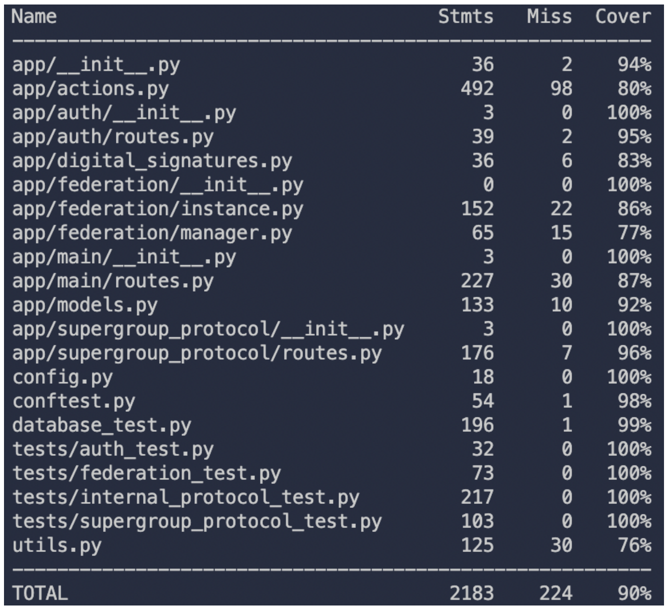
```
 PASS  src/App.test.js
  ✓ App renders without crashing (5 ms)
  ✓ App renders without crashing (1 ms)
  ✓ Welcome page renders without crashing (1 ms)
  ✓ SubscribedFeed renders without crashing (4 ms)
  ✓ CommunityExplorer renders without crashing (4 ms)
  ✓ HeaderBar renders without crashing (1 ms)
  ✓ PostCreator renders without crashing (8 ms)
  ✓ Login renders without crashing (1 ms)
  ✓ SignUp renders without crashing (3 ms)
  ✓ User Settings renders without crashing (1 ms)
  ✓ PageNotFound Settings renders without crashing (1 ms)
    ✓ Post accepts data props (80 ms)

 PASS  src/tests/AdminAuth.test.js
  ✓ Routes to Community Manager page (269 ms)

Test Suites: 12 passed, 12 total
Tests:       103 passed, 103 total
Snapshots:   0 total
Time:        28.101 s
Ran all test suites.

Watch Usage: Press w to show more.
```

Appendix E

Backend Test Summary



```
================================================================ test session starts ================================================================
platform darwin -- Python 3.9.2, pytest-6.1.2, py-1.9.0, pluggy-0.13.1
rootdir: /Users/moxis/Documents/Programming/Github/project-code/server
plugins: requests-mock-1.8.0
collected 58 items

database_test.py ...............                                                                                                              [ 25%]
tests/auth_test.py .....                                                                                                                      [ 34%]
tests/federation_test.py .........                                                                                                            [ 50%]
tests/internal_protocol_test.py ................                                                                                              [ 77%]
tests/supergroup_protocol_test.py .............                                                                                               [100%]
```



```
Name                                    Stmts   Miss  Cover
-----------------------------------------------------------
app/__init__.py                            36      2    94%
app/actions.py                            492     98    80%
app/auth/__init__.py                        3      0   100%
app/auth/routes.py                         39      2    95%
app/digital_signatures.py                  36      6    83%
app/federation/__init__.py                  0      0   100%
app/federation/instance.py                152     22    86%
app/federation/manager.py                  65     15    77%
app/main/__init__.py                        3      0   100%
app/main/routes.py                        227     30    87%
app/models.py                             133     10    92%
app/supergroup_protocol/__init__.py         3      0   100%
app/supergroup_protocol/routes.py         176      7    96%
config.py                                  18      0   100%
conftest.py                                54      1    98%
database_test.py                          196      1    99%
tests/auth_test.py                         32      0   100%
tests/federation_test.py                   73      0   100%
tests/internal_protocol_test.py           217      0   100%
tests/supergroup_protocol_test.py         103      0   100%
utils.py                                  125     30    76%
-----------------------------------------------------------
TOTAL                                    2183    224    90%
```