
CS3099 WabberJockey

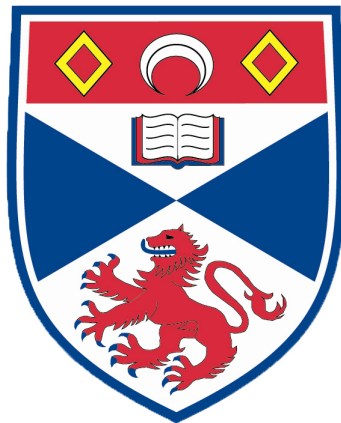
Authors:

Ben HAWKEN
Craig MCKENZIE
Charlie DON
Csoban BALOGH
Michael DOHERTY

Supervisors:

Edwin BRADY
Steve LINTON

15th April, 2021



University
of
St Andrews

Abstract

In this project we were required to implement and maintain a small-scale content platform that could be used for both private and educational uses. The system also had to be able to “federate” and share content with other instances within our supergroup of teams. Our solution is WabberJocky, a system that allows for posts, communities, and users to be created and changed by users of our platform; there is also many features that can be done across servers, such as creating posts and viewing content. All of this is accomplished using a mix of technologies, but primarily ReactJS for the front-end, Flask Python3 for the back-end, and MongoDB for the database. All of this has been put through proper end to end testing, both automated and manual. For next steps in the project we would like to build on the very solid body we have created and add more advanced features, such as live chats or video calling.

Declaration

We declare that the material submitted for assessment is our own work except where credit is explicitly given to others by citation or acknowledgement. This work was performed during the current academic year except where otherwise stated. The main text of this project report is 14,265 words long, including project specification and plan.

In submitting this project report to the University of St Andrews, we give permission for it to be made available for use in accordance with the regulations of the University Library. We also give permission for the report to be made available on the Web, for this work to be used in research within the University of St Andrews, and for any software to be released on an open source basis.

We retain the copyright in this work, and ownership of any resulting intellectual property.

Contents

1	Introduction - Charlie Don	2
2	Project details	2
2.1	Technologies	2
2.1.1	Front-end - Michael Doherty	2
2.1.2	Back-end - Craig Mckenzie	2
2.1.3	Database - Craig Mckenzie Ben Hawken	3
2.2	Navigation/Routing - Csoban Balogh	4
2.3	Main Page - Csoban Balogh	4
2.3.1	Filters	4
2.3.2	Scalability	4
2.3.3	Efficiency	5
2.4	Communities Page - Csoban Balogh	5
2.4.1	Requests	5
2.4.2	View	5
2.5	A Community Page - Csoban Balogh	5
2.6	Permissions - Csoban Balogh	6
2.7	Creating a Post - Csoban Balogh	6
2.8	Creating a Community - Csoban Balogh	7
2.9	Creating a Comment - Csoban Balogh	7
2.10	Posts - Csoban Balogh	7
2.10.1	Efficiency	8
2.10.2	Informative	8
2.10.3	Navigable	8
2.10.4	Multiple Types	8
2.10.5	Likes	9
2.11	Login and sign-in security - Michael Doherty	9
2.12	Account Settings - Michael Doherty	9
2.13	Searching - Csoban Balogh	10
2.14	Profile Page - Csoban Balogh	10
2.15	Users - Csoban Balogh	10
2.16	Front-end Code - Csoban Balogh	10
2.17	Front-end error handling - Csoban Balogh	11
2.18	Back-end - Craig Mckenzie	12
2.19	Deployment - Ben Hawken	13
2.20	Supergroup Communication - Craig Mckenzie	15
2.21	Deviation From Plan - Ben Hawken	16
2.22	Scrum and Agile Methodology - Ben Hawken	17
2.23	Exceptional Features	18
2.23.1	Caching - Csoban Balogh	18
2.23.2	Continuous Integration and Backend Test Suite - Ben Hawken	20
3	Evaluation and critical appraisal	22
3.1	Functionality - Craig Mckenzie Csoban Balogh	22
3.2	Supergroup protocol - Craig Mckenzie	23
4	Conclusions - Charlie Don	23

5	Appendices - Charlie Don	24
6	Testing Summary - Craig Mckenzie Ben Hawken	24
6.1	Back-end Testing - Ben Hawken	24
6.2	Benchmarking - Craig Mckenzie	25
6.3	Screenshots - Craig Mckenzie	25

1 Introduction - Charlie Don

WabberJocky is a federated web-application designed to be used in a university or academic setting for communication. Federation allows for communication such as posting or commenting on other servers within our super-group. The goal of this application was to create an environment where interaction with other servers is seamless and intuitive with the UI, displaying and organising information in a clear way.

2 Project details

2.1 Technologies

2.1.1 Front-end - Michael Doherty

For the front end of our project, the only option that our group discussed was React. The driving reason for this was because the majority of our group members were comfortable with React than using a new framework such as Vue. Given React has been around for some time now and is an incredibly popular framework the online resources for debugging and documentation are extensive and useful especially to some members just starting out in React. In fact, at the time of writing almost five million live websites are making use of React[3]. This information only further strengthened our belief that React was the best framework to use in the creation of WabberJocky. The structuring of React components is intuitive as well as powerful in creating parent child prop inheritance and creating complex designs. Throughout development, we relied heavily on React Hooks. Hooks allow for states to be shared between components in a much simpler fashion than the old method of requiring the definition of a local state. This allows for us to focus on other aspects of front-end development, as Hooks make light work of a previously big issue.

For the presentation side of WabberJocky, we relied heavily on the popular React component library Material-UI. It is well known that Material-UI offers thorough documentation on all its contents, thus allowing for our group to have a point of reference for any features that we were wanting to use the library for. Material-UI components are also capable of working in isolation. This means that we could use as many, or as few components as we liked without having to worry about any conflicts or the requirement for a global style-sheet. Making use of a large component library rather than multiple individual components also allowed for our presentation to maintain a consistent look as library components tend to follow the same design.

WabberJocky makes use of routing through separate URLs so that users can bookmark pages to allow for simple re-visitation of different areas of the web-app, such as a post or community to save for later. This also means that, once the username cookie is assigned and has not expired, users can go direct to these pages without having to log in, each and every time.

2.1.2 Back-end - Craig Mckenzie

The back-end technology is based around several key components: it has a Unicorn WSGI server that receives incoming requests running in front of a flask-restful framework, which uses a Flask-Pymongo extension to communicate with the database server. All of this is written in Python3.

We chose to use the Python3 programming language as we knew that it had many available libraries that we were already familiar with, and we were sure that we could discover more for any issues that may arise. We were also all familiar with it in some way, making development much quicker and simpler as we were not having to learn a new language and try to develop the system at the same time. Another benefit is that it allowed for us to solve things in a “pythonic” way rather than in a traditional HTTP sense when the situation called for it (for more detail on this functionality, please see the back-end functionality section)

Flask is the framework around which the entire back-end works, as it is a simple and lightweight routing tool which allows for a request to be read in, directed towards the correct method for said route and then processed. This is made simpler by the addition of the FlaskRESTful module, which allows for routes to be written within python class structures rather than having to rely on flask's default if statements, which can be difficult to interact with for a program of this scale. We also make use of flask-pymongo, which is a module that allows us to access a single database connection globally (flask does not natively support global variables asynchronously), preventing errors when too many connections are made to the mongodb database at once. All of this is supported by a selection of python modules, such as the cryptography library, which are used for specialised purposes such as encrypting headers for the security component of supergroup communication. We chose to use Flask as it is simple and lightweight, allowing for you to use Python for most of the work and only using special Flask methods and types when returning or receiving data. This also assists in efficiency, as none of the operations performed by this program are particularly complex, making a lightweight utility like Flask ideal.

Gunicorn is a simple utility which offers the ability to start a HTTP server that sits in front of our flask framework. This is important as flask natively only supports a development server, which does not have proper thread management or the ability to use multiple workers, making it inappropriate for systems that have many requests being processed at one time (such as this one). Gunicorn also offers the ability to use different types of worker; we have chosen to use the sync workers as wabberjockey does not contain many blocking calls, making the use of gevent (async) workers expensive and unnecessary.

2.1.3 Database - Craig McKenzie Ben Hawken

Our database runs off MongoDB, a NoSQL database solution that was available upon the school servers and could be run locally. We chose to use this database solution as there are many clear guides that we could use for setup, but more importantly we knew from the start that the data that we would be receiving across groups was never going to be 100 percent consistent, so we decided that the ability to use MongoDB's lack of fixed schemas to perform our own python validation and then simply use the database as a storage solution would be, in our opinion, the best solution to this issue. It is also extremely efficient in terms of retrieving data from our database, as all our HTTP pass in and responses are done in JSON, and MongoDB exclusively stores data in JSON document format, allowing for us to simply pull data and send it, rather than performing complex string building operations. Whilst MongoDB would usually benefit from being run on a cloud database (called MongoDB Atlas), we decided that for the purposes of this project the simplicity and control offered by a local MongoDB instance would be the best choice.

2.2 Navigation/Routing - Csoban Balogh

The idea behind the URLs and navigation was to have each and every page individualised. What this means is that navigation to any one page is not dependent on previous navigation of pages. This allows crucially for sharing of URLs among classmates who can copy and paste into their search-bar and get to the content they need without needing to step through a community or main page to arrive at a comment or post.

This navigation is also intuitive in the sense that navigating from for example Community “cats”, it gives the server the user will be posting to, and the server id in the URL for clarity. As a user this gives you context of where you are in the site, and hints at what you are doing among the other site content obviously provided.

The actual navigation is handled via our built in App-Bar with the WabberJocky logo, a search-bar, communities, create post, and profile button. Given this is an App-Bar, a user can interact with these buttons wherever they are on our website, making it easy to navigate back and forth between home page, communities page, profile page etc. Further, these buttons are extraordinarily clear in functionality that it is intuitive as a user who has never used our website in how to create a post, go to communities, go back to home page etc. The overall goal was to create a website which could be navigated around without prior knowledge of layout, something we have achieved.

2.3 Main Page - Csoban Balogh

The Main Page is where all posts are displayed from all servers in a nice feed style. The choice of displaying all servers at once rather than separate servers was because this gives the most federated feel. Scrolling through you have posts from all different servers, communities, and users who you can interact with. This feed style was chosen so a user can continue scrolling through posts and see information at a glance all at once, does not need to step into the post itself. To curtail the feed and display the most important information on the main page the title and only part of the description is rendered, meaning posts can be displayed in a feed style without one post taking up half your screen due to a long description.

2.3.1 Filters

To filter your view the server the post was posted from can be narrowed down using a drop down menu. This makes it easy to see the posts from a server you might be interested in finding for example, or what's rather nice is you can select multiple servers, not just one, giving flexibility in choice.

You may note there is no way to sort by likes or time or by some other metric because of the super-group. Some groups have not implemented all of the super-group functionality which makes sorting very difficult in filtering and getting from servers without having to do all of the work on the client side from pulling all posts over. For example, our group implemented our own version of likes given we believed the protocol version was over-complicated for our usage, so we share some of the blame. Of course this is not ideal but given the other provided functionality this can be overlooked.

2.3.2 Scalability

In order to be scalable for future expansion if our project ever goes huge, to begin with the posts are fetched incrementally with increasing limits, a form of pagination to ease

load on the server and the client. As information is loaded in the user is fully aware of when loading is done through a simple indicator of a loading icon. This icon used to be animated but found some older computers could not handle this rendering very well, so we went with a simpler loading icon that will display with almost any computer.

These posts fetched from several servers come in incrementally as requests are being waited on from other servers, meaning we always have something to serve even if we have yet to retrieve all of the information as of yet. This ensures faster rendering times for the user in displaying parts of the main page only that a user may be viewing at a time, getting content to the user as fast as possible.

2.3.3 Efficiency

Alongside scalability is efficiency of the requests alongside filtering. If a user decides to filter the servers they see on the main page and load in more information, only the servers selected for viewing will be loaded in, not the one's filtered out. This of course makes sense, do not do more work and put more strain upon the network or client than is necessary. The main page tracks the pagination (limit) of the requests per server so if a user does want a certain server back, the request for only the server selected who is behind in pagination is made, preventing unnecessary requests from being made.

2.4 Communities Page - Csoban Balogh

2.4.1 Requests

The communities page is a simple view of all available communities across all servers, following the style of federation. Due to the protocol if we wanted to display more information about the server a separate request must be made on-top of the community request already sent out. For the most part this request is redundant, it only provided a description of the community which a user may or may not want depending on if they regularly use the site and know this information already, or can guess from the community name.

As a result, to reduce requests the community description is not immediately fetched and instead can be requested by the user in showing more. Once a user decides to show more of a community, the description of the community is saved into the state so if the description is hidden and shown again, a repeat request does not need to be made.

2.4.2 View

The filtering of the communities follows the same as the main page where a user can filter out numerous or a single server in their view. For convenience the communities are grouped by server to see clustering.

2.5 A Community Page - Csoban Balogh

A community page is basically the main page but with posts only from that community, displaying these posts in a feed style as well as some information about the community page, such as title and description. A user has the ability to select a button and create a post, and as this is federated, this community does not need to belong to our server. If a user is an admin of a community they have greater control such as being able to modify

the community description, as well as certain power over posts in the community such as deleting posts that do not meet a community's standards.

Note the community title cannot be changed as this is tied to the community ID. Much like the main page a communities posts is loaded in as it is requested to show something to the user, and will display loading to indicate to the user the request is still being processed, ensuring transparency.

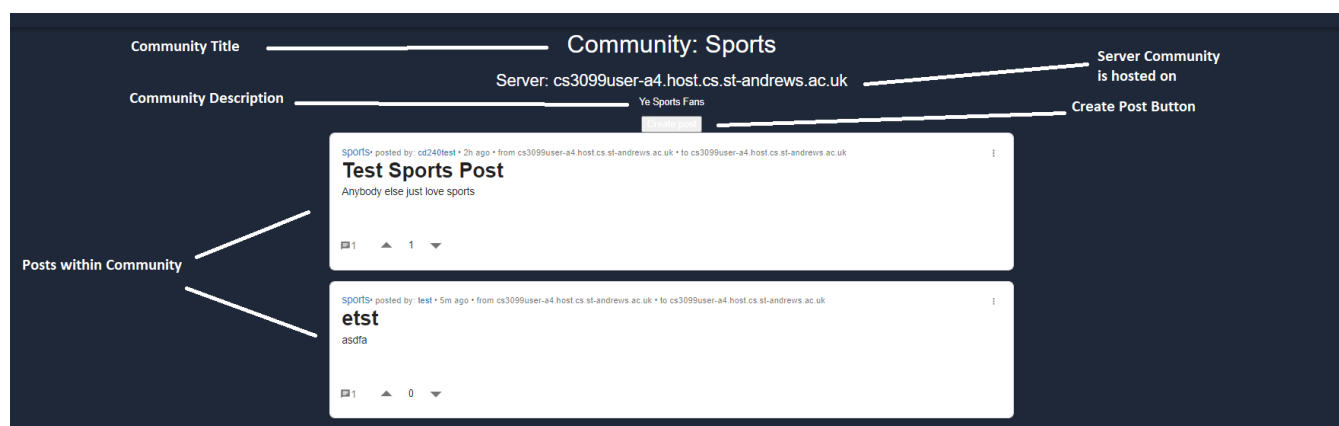


Figure 1: A Community

2.6 Permissions - Csoban Balogh

With creating any form of platform, there are certain actions users should not be allowed to do. For example, a user should only be able to edit/delete a post, community, or comment they have created. A simple way to restrict this is to hide the buttons of these actions from the user in the first place, preventing them from doing these actions. If a user somehow bypasses this, like by editing the front-end code, the back-end will respond with an error code and prevent the action from going through. Certain users have more permissions such as admins of communities, allowing them to moderate the community posts and comments, removing any which they deem necessary.

2.7 Creating a Post - Csoban Balogh

Creating a post was designed to be as flexible as possible, allowing for different content types, manipulating the text display, displaying code and so forth. The most pragmatic and simplest way to approach this type of problem was to have posts be markdown, meaning users can insert in most content types they want with the formatting they like, offering great flexibility to the user.

Given the complexity of storage management, as well as the fact other solutions exist in much better API we decided not to store content other than text/markdown in our server. So if a user wishes to include an image in their post, they must first upload the image to an external site to obtain a URL which can then be inserted. For users who may be unfamiliar with markdown we provide a helpful guide on the create post page detailing how to put an image into markdown.

2.8 Creating a Community - Csoban Balogh

Communities as per the protocol can only be created on our own server as this functionality was not specified by the protocol, handled server by server basis. A user can set a community's title and description upon creation, taking the user to the community page once it has been made. Note the community title which is tied to the community ID must be unique to the server, so there cannot be two "test" communities on our server, otherwise this clashes in the back-end database. This also makes sense given if a community name already exists, we should not be making a duplicate copy essentially. This clash error is handled gracefully and reported to the user without losing information put in for the description or title. When a successful creation is made, the user automatically becomes an admin and gains special admin privileges over the community posts, description etc.

2.9 Creating a Comment - Csoban Balogh

Comments are designed to be easily readable through a tab based layout so replies of comments are grouped together to follow the conversation easier. A comment is practically a post with a few extra add-ons; especially if the comment is in reply to a another comment, the top of the comment displays who you are replying to, as well as a piece of the text you are replying to.

Like posts the author of the comment is embedded in a link to follow and navigate to for ease, as well as giving a description of which server the post is coming from. As a bonus comments can be liked as well similarly to posts, but only if they are posted to our server (i.e. if we support it).

2.10 Posts - Csoban Balogh

The layout of posts was designed to be easy to read, efficient in requests, and intuitive in actions.



Figure 2: An Annotated Post

2.10.1 Efficiency

Not only can a user delete their own posts, but if they are an admin of the community of a post they can delete that as well. To determine this a separate request must be made to get the community admins. This is not ideal if we have to make multiple requests upon first loading into the page to determine which posts a user can delete. To reduce these requests, only once a user clicks on the “more options” button does the request for the community admins get made, so when the user wishes to delete/edit the post respectively. This result of if a user is an admin is saved into the state of the post so the request does not need to be repeated if the drop down menu is opened once more.

2.10.2 Informative

The post displays all of the relevant content a user would want to know from a post; the community it was posted into, the user id of the author, how long ago it was posted, which server it was posted from, and the server it was posted to. How long ago the post was created is printed in a nice format depending on the time it was posted relative to you, giving it in minutes, hours, days, or a date respectively.

The title of the post is displayed in big bold letters for readability, and depending on which page you are on either part or all of the post content is displayed. Part of the content may be displayed if the page is only showing a preview that the user can then expand by visiting the comment page in order to truncate and save space for other posts on the page.

If the post is from our server then the likes of that post is displayed as well as an indicator of whether you’ve liked the post or not. The number of post comments are displayed as well. Note this is only the number of top-level comments as in order to fetch the number of all comments, this needs to be done recursively and can be quite expensive. This is especially true because some groups have not implemented the protocol feature to do this work for you on the back-end, thus this is all done on the front-end to ensure maximum functionality with a price to performance in requests.

2.10.3 Navigable

For convenience a user can navigate from a post to a user’s profile page, the post community page, or to the post’s comment page. This is done by embedding links into the post for each respective element, making it easy and intuitive for the user to navigate from the post to where they want to go.

2.10.4 Multiple Types

A post can support multiple different content types thanks to the use of markdown. This is displayed within the context of the post, meaning images and so forth do not extend beyond the boundaries of the post, keeping everything nice and contained, especially for the main page. The use of multiple content types allows for great flexibility to the user in expressing a need or conveying an idea across to other users. If you remove the barriers to communication, ideas and the exchange of knowledge flows much faster and clearer.

2.10.5 Likes

To indicate to users which posts are the most helpful, or in some cases which one's are the least helpful we implemented likes to show how well a post is doing. Note these likes only work on our server given the lack of participation in protocol likes, so do carry that small disadvantage there.

2.11 Login and sign-in security - Michael Doherty

The focus for all pages associated with the log in process was to keep things as simple as possible by avoiding unnecessary clutter. The login page consists of two forms – 'Login' and 'Sign Up', whilst the 'ForgottenPassword' and 'ResetPassword' pages consist of only a single form. All pages also contain the WabberJocky logo.

All forms undergo presence checks upon submission to ensure that no fields are left empty. For the 'Sign Up' form on the 'Login' page, a check is made to see that both password fields are matching along with a regex check on one of the password fields, ensuring that a user's password meets the general standard of being at least eight characters long and a mixture of alphanumeric, special characters and capitalisation.

Through the 'I've Forgotten My Password' button on the 'Login' page, users are sent to the 'ForgottenPassword' page where they are asked to enter their username. If their username exists within the database, then their security question is retrieved from the back-end and they are sent to the 'ResetPassword' page. Here the user is to answer said security question (which is passed via cookies) as well as enter their proposed new password twice. This password must follow the previously mentioned regex.

Via the use of cookies, if the user attempts to gain access to a page beyond the login before signing in or after their username cookie's expiration, then they will be sent back to the 'Login' page automatically. This only differs on the 'ResetPassword' page, which uses a different, temporary username cookie. If this cookie has not been set, then the user is sent back to the 'ForgottenPassword' page. Upon the completion of the reset password process, the temporary username and question cookies are removed. Cookies were included as they allowed for a persistent log in, which both aids in usability as well as testing for Wabberjocky. When a user successfully logs in, the username and host cookies are set. As mentioned before, these cookies have an expiration time of a day from when they are set, but they can also be cleared by the user logging out. To further aid in security, all cookies are set with the 'sameSite' tag. This means that the cookies are not sent in cross-site requests, reducing the chance of an information leakage.

2.12 Account Settings - Michael Doherty

The profile settings page allows for the user to change three aspects of their account – their about, password and security question. The user's about can be changed without requiring anything, the user's security question can be changed with their password and the user's password can be changed with either their current password or their security question. This means that there are four buttons to open the forms for each process.

When one form is opened, any other form that is currently open is automatically closed to aid in presentation. For both password change options, they undergo the same regex and matching password checks seen in the 'Login' page.

For the form that changes the user's 'about', a textarea is used rather than an input to help show the user more of what they are typing, as a standard input reaches its display

cap rather quickly. A length cap of 280 characters is applied to the ‘about’, to match the same restriction set up in the back-end.

Upon the validation checks passing for any submission, the JSON for each change request is created from the username cookies and form inputs and is sent to the back-end. Upon success, the form is then closed. However, if there is an error (wrong password or answer to security question) then the form remains open, and the error message sent from the back-end is displayed to the user.

2.13 Searching - Csoban Balogh

All the posts in one feed can become a bit overwhelming at times, so to make it easier in narrowing down an issue or problem you might be facing in class, a persistent search-bar is implemented so users can navigate and search with ease. Sadly the protocol did not support searching by title of posts so this was implemented locally only, but the search does return closest matching prefixes of users, as well as prefixes of communities. All of these results are presented cleanly in a tab each to their own, alongside with the functionality of loading in more posts (pagination) with a loading indicator so the user understands the progress of the search. The pagination helps a user’s browser not be overwhelmed with 1000 results being returned, and can incrementally increase as necessary.

Note at one point we did support searching all federated servers and all posts by pulling everything client-side and sifting through it ourselves. However, this process took about 30-40 seconds and that was with the limited number of posts each group is currently putting out, so this idea was quickly scrapped and instead moved to supporting title searching locally only.

2.14 Profile Page - Csoban Balogh

The profile page displays the a user’s about, as well as all of their posts so any user can browse through to see the history of the user. This is a simple way to show to a user what type of posts you create all in one place, plus grouping together posts is ideal as a form of simple filtering. A user on their own profile page can take this moment to edit or delete any of their posts, all localised for a quick view.

2.15 Users - Csoban Balogh

Users are simple yet have lots of functionality at their fingers. A username and password are unique across all servers, meaning each user has a combination allowing them to delete, edit, create etc only under their name. If a user attempts to circumnavigate this, the back-end will stop any further progress. Users additionally have the ability to become admins of communities, giving them power over deleting posts as a form of moderation.

2.16 Front-end Code - Csoban Balogh

The front-end code is split into multiple files as a way of compartmentalising the different React components, and for method reuse crucially. For example, as you can imagine the Communities.js file is for communities, Profile.js is for the profile page and so on. So much of this project is about displaying information, and so an entire file is dedicated to displaying different kinds of information such as communities, posts, etc. These methods

are reused several times in displaying the posts and communities across separate pages, with nuances in functions called.

By compartmentalising functions, a large amount of abstraction was done to reuse code across components. This was done because halfway through this project we realised how volatile the Supergroup protocol was, and how quickly it could change at a moments notice. After several refactors it was decided to abstract out methods, essentially writing a Java program with getters and setters so if at any point the code-base in the future changes, the only lines that need to be changed are those within the getter and setter functions.

As much of the networking code is almost identical, this was abstracted out as well in an attempt of method reuse. Now the networks gets, puts, deletes all point to one function which can be changed as need be if there are changes to the networking requirements. This was all done partially in mind of making the code maintainable for the future and easy to extend by simply calling these functions.

So much of the code is dedicated to server-client efficiency, reducing the number of requests one needs to make in order to have a functional front-end. This of course is thanks to React states and exploiting the virtual DOM in being able to easily save and update information as it comes in to then reuse at a separate time if the need arises.

2.17 Front-end error handling - Csoban Balogh

The front-end handles errors in a simplistic manner through the use of alerts. Personally we did not want to go down this road of using alerts but each group returns errors and especially error messages in different ways, making it hard to insert into the HTML in a nice way and be informative in the messages. As a result, for simplicity and for the fact certain groups in the SuperGroup do not return well formed messages this forced our hand to simply print out whatever text we do receive into an alert.

A majority of the error handling is based around the network and dealing with errors there. If one server fails to return us information but the rest do, we don't crash or go to a 404, the front-end continues and serves what it can to the user at the time. Note some errors are suppressed and grouped in an effort to streamline the user experience and not be constantly bombarded by multiple different error messages. This came about because some groups routinely would either have their servers down or would be passing back malformed data, so rather than informing the user every time and exhausting them, these are mitigated to streamline the experience.

Other forms of error handling are if the other servers send us bad data, or data that we do not understand. The front-end is a best effort service and does not fail fast, if there is incorrect data or data it does not know it will display what it can to the user and inform about the nature of the other error. For the most part the front-end can run smoothly without incident but at times there are certain failures which it cannot recover from, particularly in getting a post or community to display it's sub-posts. If such a case arises, once more the user is notified with the message from the server and the front-end keeps going with whatever it did manage to get. Otherwise this is a good prompt for the users to try a different page at least.

For all requests to the back-end related to a user's account, whether it's the creation of an account or changing an account's settings, error handling exists so that if a request fails due to the user's input (for example attempting to register with a username that already exists), the user is notified with a simple message sent from the back-end.

2.18 Back-end - Craig Mckenzie

In this section we will be describing the functionality available to the back-end and how it accomplishes this functionality. For in depth descriptions of each piece of functionality, please refer to the front-end section, as the focus of this section will instead be how the back-end interacts with both the front-end, database, and other users to provide a robust and efficient experience. For a description of the technologies used to accomplish this, please refer to the “Back-end Technologies” section above.

For usage details of the back-end please see the README.md contained within the “Back-end Code” folder

The back-end operates on a restful basis, in which it is first loaded with all its possible routes upon start-up. Once someone tries to access one of these routes with a http request, it is then scanned for its request Method (PUT, GET, POST, etc) and directed towards the correct function for this method. This is then handled in a “pythonic” way, as one of the best features offered by the flask framework is that it allows for us to process things using known and well-defined python terminology, such as using python dicts and recursion, whilst still giving us helper methods that let us do things like access the Mongodb database and return in the correct format.

The error philosophy of the program follows the system that was developed by Netflix for methods that were going to experience possible failure [4]; fail properly, fail silently, or fail quickly. If someone fails properly, then it returns a proper error code as designated by us within the flask application and returned in the same way we would return a successful request (only with a failing error code). If this is not possible, we fail without letting the user know anything has gone wrong by catching common errors within try catch blocks and assigning an approximate code to them (this only ever happens due to issues with other groups, as we are able to predict with high accuracy anything else that could go wrong. If both protocols fail, we rely on the pythonic solution, which is to use a generic try except excerpt on every method that will catch any crashing errors (mostly network related issues) and provide a generic error message to the user immediately, stopping the issue from propagating and affecting the rest of the network. This allows for solid uptime and quick requests whilst only using 2 workers on our Gunicorn instance.

Whilst local requests (the front-end accessing data that we host on our Mongodb instance) are relatively simple, supergroup requests are more complex. These require us to effectively build our own request within the instance and then send these to another group’s back-end instance, having us operate as a “middleman” to allow for the front-end to receive data from/send data to other groups whilst still being insulated to requests within our environment. This is also necessary as the protocol requires for us to build up a series of headers in line with those required, a digest header which is a hashed version of the body of our request and a signature header, which we encrypt with our own private key. Other servers then receive these headers and can verify them using our public key available at the endpoint /fed/key to ensure the request came from our backend. It is important to note we were only able to implement the security headers part of this feature, and our implementation does not have the verification. Unfortunately, these hash and encryption requirements do slow down requests to other servers somewhat, but our error checking solution discussed above does prevent any request that has taken too long from affecting the network.

In terms of features of the protocol that we have been able to implement, we have all of the functionality described apart from the previously mentioned security verification, the ability to post messages (we saw this as an unnecessary addition) and the ability to

use reacts and likes across instances, as we felt that the solution that was developed in the end is too inefficient to scale well across the network of groups, so we instead chose to use our own local only version of the likes system. All of these compose very minor parts of the protocol, and every other feature works across other groups, whether it is sending or receiving data.

In terms of the functionality specifically contained within the protocol, the server allows for posts to be propagated across instances, meaning that as a4 we could post on any other group that has given us their server details and then retrieve said posts using a selection of methods, such as getting a list of posts conforming to several conditions or getting a specific post. This is the only creation method we support on the protocol front (the only other one possible is the messaging function). This is because we felt that we wanted there to be absolutely no stored information across the servers (beyond individual caching), as the entire point of the federation, in our opinion, was the ability to send/customise data across servers whilst still only keeping the data stored on one server; for example, 6 users from 6 different groups could all be interacting on one post, and whilst it would appear consistent across all of these users, all of the data relating to this post is in fact only stored on the server where the post was originally created. This is all supported by the fact that the back-end and database both structure all our data to conform with what we must send according to the protocol. This means that rather than having to perform complex string creation to get posts from our database, it is very easy to simply pull the JSON document from our MongoDB, convert it to a JSON string and then send it to another server. This makes the process of retrieving and sending data very efficient, with the only possible delays being those when other servers are accessed.

2.19 Deployment - Ben Hawken

Once we had minimum viable local implementations of our front-end and back-end, we then needed to tie them together and serve them to users. We initially thought about using an external deployment service such as Heroku. Heroku would have dealt with a number of deployment details automatically so would have been a good option, however, for safety of our project it made most sense to deploy to the school servers. If Heroku went down or had an issue during our demonstration then this would be unfixable.

Having chosen to deploy to the school servers we would have to manually create a number of deployment features that would have been handled automatically in Heroku or equivalent. It required us to find a way to maintain continuous up time of the servers, link the servers together so that interacted with each other, access these servers publicly through a custom URL and allow for quick and easy changing of the code to ensure as little down time for users when making feature changes or bug fixes.

Firstly, in order to ensure that our servers were running with a constant up time even when we were logged out of the school system, we used tmux. Our front-end server, back-end server and database server were all kept in separate tmux sessions to allow for modular changes of the code to each. These three tmux sessions were set up and each connected to a respective 'production' GitLab branch. It was then simply a case of calling 'git pull' in whichever tmux session you were in and then using the command to start the server. These tmux sessions were a good solution that only had a few faults. One session reached its limit of lines and shut down which caused us a great deal of confusion for a while but once we found this issue it was easy to relaunch the tmux session and remove the hard limit of lines. When the university servers were taken down for maintenance this

also removed the tmux sessions, however again, they were incredibly quick and efficient to create again. A further positive we found with the tmux sessions was in the back-end where we required environment variables to be set up in linux to be used in our python code to distinguish between production and test databases. These environment variables only last for as long as they are in the session for, however, with the tmux session open consistently, the environment variables only needed to be set once and they then persisted for as long as the tmux did.

Linking the servers together and then being able to access them through a custom URL was the actual ‘deployment’. For this we used the universities installation of nginx with our own custom configuration file. The main nginx config file already specified that our URL would be ‘https://cs3099user-a4.host.cs.st-andrews.ac.uk/’ so visiting this website when nginx had been launched on default gave an nginx welcome page. It was then a case of writing our own separate config file that would redirect any traffic to that URL through to our local servers. This way we could have our servers running locally as if we were going to access them with localhost, but then point them at this new URL. In order to distinguish between the backend and frontend requests, we used two separate port numbers. Each server was started on a local port and then there were two sections to the nginx file. If a URL contained a request that would effect the frontend, nginx directed the request to the frontend server. If a request was made that didn’t need to change anything in the UI but needed to update the backend then it was directed to the backend server. The backend server in turn updated the database server so no separate nginx configuration was needed for the database server.

At this point we were now able to access our application from our custom URL and perform actions that updated either the frontend or backend. In order to make this deployment even better for users, however, we needed to find a way to decrease the amount of time it took to update code and push that update to the URL. We found that we were updating the backend functionality a lot and we didn’t want to have an application that showed the frontend but then wasn’t responsive when used because the backend had been taken down for maintenance. To fix this issue, we used a deployment server in front of our aforementioned Flask server called Gunicorn. To set this up, all that was needed was one extra file that called main python flask file that would be in turn called from a Gunicorn command. Gunicorn was useful for two major purposes, the first of which is allowing Gunicorn to run the Flask application outside of development mode. This has multiple benefits, such as the ability to print without accessing the python error library, but most importantly it allows for us to set up a group of workers. As the mongodb allows for multiple connections to be processed at the same time through native thread management, these workers can work separately from each other. This does present some limitations, as long blocking calls can result in both of the workers being made busy, but the system is designed within the backend to prevent any method from operating for too long, and there is no blocking that could occur within the Wabberjockey instance (this clearly cannot be guaranteed within other servers, so requests to them are assigned a relatively short timeout). All of this together allows for the Gunicorn utility to handle hundreds of request per second, well beyond the scope of this project.

Secondly, when we were using just the Flask server, in order to update the code we would need to kill the Flask server, pull the new code from the Gitlab and then restart the Flask server. With Gunicorn, we were able to employ the `- - reload` and `- - daemon` flags to eliminate all of these extra steps. The `- -daemon` flag allowed us to run the server as a background process so that we could still enter commands into the command line in the

same tmux session that it was running in. The `--reload` flag allows the server to ‘reload’ rather than ‘restart’, meaning that the server doesn’t need to be taken down in order to incorporate the change to the code. When it detects a change in the code, it reloads those changes and incorporates them without downtime. Using these flags together, this allowed us to enter the backend tmux session and simply write ‘git pull’ to add the new changes to the deployed backend.

The discussed decisions we have made have allowed us to create a robust, manageable and flexible deployment of our application. It is separated out into 3 different tmux sessions which means that if one goes down for whatever reason, it doesn’t affect the others allowing for a quicker fix. We have also taken steps to ensure that the deployment itself from our end is as efficient and non-intrusive to user experience as possible with our use of Unicorn’s features.

2.20 Supergroup Communication - Craig Mckenzie

In this section we shall outline both the structure, process, and results of the interaction that occurred amongst the teams within the supergroup. This structure has remained mostly consistent throughout the project, but at times where it differed (during times of “protocol freeze” or in the final testing stages) this will be highlighted below.

The supergroup originally met on 30/9/2020 at 12pm and throughout term time from that point on we would meet at this time each Wednesday. The attendees of these meetings would be the chosen rep for each group within the supergroup, who would also oversee relaying info back to their team, but other group members were welcome (though they would usually be requested to ask questions through their rep to keep the meetings on track). These meetings originally took place in the style of structured meetings, where-in a chair would be nominated to manage each meeting to ensure everyone got an even amount of time to discuss the issues at hand. Gradually this structure would become more free form as the representatives of each group become more familiar with each other. At these meetings we would go through a list of issues, originally mainly being comprised of the formation of the protocol in terms of big picture (no shared resources, request based only etc.) but as time went on, we would instead focus more on individual issues (such as the structure of a particular request or reviewing a pull request together). It was agreed very early on that we would prefer to be slightly slower in having a full protocol together, as we thought this would allow us to have better forward vision rather than trying to rush something together which would not hold up as the project expanded.

The only times where the above structure was deviated from was where we would “freeze” the protocol in the run up to deadlines, with this effectively being periods where meetings would not occur, and no changes could be made to the protocol (unless said change was due to a critical flaw or a missing piece of information). The meetings also deviated in the period before the final submission, as these were held as “testing” meetings where groups could test the protocol live between them, making sure that the protocol would work in a production environment rather than just during each group’s individual testing.

The results of these meetings can be seen in the protocol, which is hosted at:

<https://fmckeogh.github.io/cs3099a-specification/section/Description>

If there is an issue with the above hosted link, then we have also attached the .yaml file within the appendices folder, which can either be read free hand or installed into any popular browser viewer (we would recommend the site SwaggerHub as this is what we

used in development, but it could be used in any viewer of your choice.)

The protocol is based around HTTP requests, as we decided early on that while a redirect approach (in which we knew of other groups and could redirect you to their frontend using a shared login feature) may have been easier to implement, we felt that this would not provide a seamless user experience. The other major point that we emphasised throughout the development of the protocol was that we wanted groups to implement their own version of the protocol rather than having to strictly follow the protocol. To do this, we decided that the protocol would only treat some things as absolute (mainly the json that had to be passed between groups) and beyond that it was entirely up to individual groups how they wished to interact with said json. Using this philosophy, we structured our development from the start to follow the structure that was prescribed by the protocol within our mongodb database (as mongodb stores natively in json and that was what we would be passing between groups), meaning that we could quickly serve both requests from our own frontend and other groups without having to adjust the data in any way. This meant that when it came time to add the supergroup functionality (described in the Functionality section of this report) we simply had to serve requests in the exact same way and redirect our frontend requests to the correct server when they wanted to retrieve nonlocal data. This made the change from local only to supergroup very fast and allowed us to spend time implementing other extra features which would only be served locally.

2.21 Deviation From Plan - Ben Hawken

Over the course of this year our main direction and goals for the project have stayed consistent, however, our extensions and extra features have changed drastically. This has been facilitated by our adherence to an Agile Methodology which has allowed us to create flexible short term plans that can adapt and change. Looking back to our first report, we have fulfilled our main goals. We stated that we would attempt to implement a system according to the “who, what where” criteria where a user could see who posted something, what they posted and where it had been posted. We further planned to be able to display comments and replies in a tree like structure as well as have the ability to create custom ‘communities’ to post in. We have fulfilled all of this criteria and our plans have stayed relatively consistent regarding them throughout. The only major changes to these plans that we had to make was to migrate our local application to work with the federated Supergroup protocol. This required changing URLs, functions and even switching to different technologies in some places for every member of the group. This was time consuming and required a great deal of communication to ensure that compatible changes were made in both the frontend and backend.

While our core plans and design has stayed consistent throughout, our plans for extension features and what we consider impressive extension features have changed significantly over the project. Due to this being our first project of this scale, it was clear that we underestimated how much time certain features would take and how many hidden features there are under the hood of an application that allow it to work effectively. Initially, for one of our extensions, we wanted to create a series of bots that would make their way through the posts in different communities and perform certain actions, much like a Reddit or Discord bot. With no experience of software engineering pertaining to web applications, at the time this seemed like an impressive extension that would make the application more exiting to use. As the project progressed, however, it became apparent

that a more important extension would be to implement caching. This is not as exciting for the user or as ‘flashy’ as bots, however, to create a web application that functions in a professional and scalable way, caching is essential. Over the project we have learnt that there are a number of processes, like caching, that are far more difficult to implement than a Poll or a Bot but are essential for a smooth running application. Our focus certainly shifted more towards scalable functionality and creating a robust application as the project continued and this is reflected in our final extensions section of this report in comparison to our initial ‘potential extensions’ section in our first report.

2.22 Scrum and Agile Methodology - Ben Hawken

The Agile Methodology and Scrum Framework were born out of the need for a more efficient process than the outdated Waterfall Software development methodology. If we were to have used the Waterfall method in this project, we would not have been able to implement anywhere near what we have been able to following a Scrum framework. An essential part of our process has been flexibility and the ability to change any part of the code at any time. With the Waterfall Methodology, every part of the code has to be finished completely before the next one is started (database, then backend, then frontend then deployment etc) and this takes away any kind of flexibility in the development process. Further, the actual application is not seen or used until the very end of the project which is detrimental to testing usability and adapting to the results of those tests along the way. If we needed to add certain functionality to the backend because of a feature we wanted to add in the frontend, the Waterfall Methodology’s rigid specification would make this very difficult. Instead, using Agile development processes within the Scrum framework has allowed us to make all of those necessary changes alongside each other.

As mentioned, a key part of this project has been flexibility and ability to change our goals and aims to adapt to what is most important at the time. If we set a goal at the beginning of the project for what our application would be like at the end of the project, there would have been little chance that we would have succeeded in that goal. With this being our first software engineering project, we have all had steep learning curves to overcome. Learning new technologies at the same time as using them to build something lends itself brilliantly to an Agile Methodology. One week we could be aiming to build some feature but then the next week we realise that the technology we have chosen to use doesn’t allow us to complete it in the exact way we envisioned. With the Scrum framework we are able to adapt or change that goal in our next sprint meeting, allowing the process to continue at the same pace rather than having to stop and re-plan large chunks of the project that relied on that exact feature a long time into the development lifecycle.

Our group has fallen into a regular pattern of Scrum meetings and sprints as was outlined in our last report. We have a weekly team meeting where we discuss what each of us has achieved that week, what functionality we can tick off verses what is still left to implement and finally, what we will each aim to finish before next week’s meeting. We ensure that we are realistic with what we each choose as our sprint goal for the coming week, ensuring that we meet our goals more often than not. This allows us to plan effectively and map out our progress so as to fit in our development with our other modules. While we have our main weekly Scrum meeting to track our progress, if needed we also set up smaller more informal meetings when a member of the group needs help

completing a certain task so we can re-calibrate the scope of the sprint for that week. A directory containing all of our notes from these weekly Scrum meetings is included in our submission.

At the end of each weekly sprint we would demo the current state of the application to our supervisor, as is in line with the Scrum Framework. This enabled us to have a useful fresh perspective on our application in the same way a user would think differently about the application than the developers. It is often the case that users will use specific features in a way the developers hadn't intended or not even use other features entirely. These sessions with our supervisor emulated this user testing and allowed us to make changes based on the feedback. If we waited until the end of the yearlong development process to ask for feedback on the usage of the application, as is more common of a Waterfall Methodology, we would have certainly not been able to adapt our features to a more ergonomic and exciting user experience as we feel we have been able to. Instead, obtaining feedback each week allowed us to adapt our individual goals and tasks for next week's sprint and use our time as profitably as possible.

The incremental development of the minimum viable product of our application also proved useful. We found that having some kind of working implementation, even as basic as it was at the beginning, seriously helps with the direction and vision of the project. Being able to see how a new feature actually interacts with others every time one is added rather than just theoretically thinking about how they will interact allowed us to make far more informed decisions about which user stories to focus on. The initial list of user stories we came up with was overwhelming at the beginning of the project because it was difficult to know where to start with them. If we had chosen a pathway through them at the beginning of the project and stuck to that without inspecting how each one performed in the application itself we certainly would have encountered problems when we put them all together at the end.

A useful analogy would be to think of a jigsaw puzzle. If we had all of the pieces of the jigsaw puzzle (the user stories), inspected them and then tried to theoretically come up with a plan about how to assemble the jigsaw without ever touching the pieces (Waterfall Methodology), the end result would certainly not fit together smoothly. On the other hand, if we started with a few smaller pieces and put them together incrementally (Agile Methodology), as the puzzle building process goes on we would be able to fit the pieces in with each other much more accurately and see the bigger picture. Further, if we put the puzzle pieces in the wrong place, we can adapt and move the pieces around. For us, this analogy perfectly describes the benefits we have seen with working in an Agile Methodology.

2.23 Exceptional Features

2.23.1 Caching - Csoban Balogh

To further reduce the number of requests to the user there is a client side cache in local storage, storing the requests and results of queries to the server and other servers. To be functional, each request has a TTL attached, storing this information in cache for TTL and fetches this instead of from server if TTL has not expired. Perhaps the hardest part about the cache was implementing local changes made by the client while the TTL had yet to expire. This is because any changes made by the client before TTL must be reflected in the cache, otherwise there would be unexpected data when re-fetching the cache. This could have been easily circumvented by re-requesting every time a user makes a change

locally, but this would result in unnecessary requests for information that we may expect to stay “good” for a bit longer.

To do so, we basically rebuilt our own little database client side. Consider going to the home page and seeing a post about cats. If we also navigated to the community cats and saw the same post, the client is doing redundant fetches for the same data. To avoid such an issue, post data is stored separately under their unique IDs, using these as keys. The request for the home page and the request for the community cats store the IDs of the posts they expect to show, referencing both to this one stored value of the same post to avoid redundancy in storage.

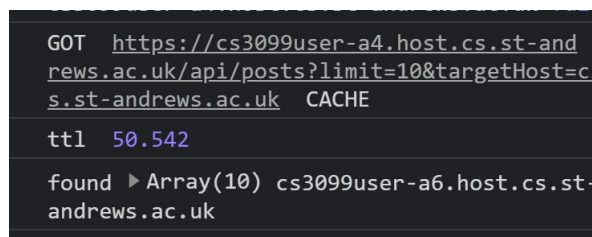
This makes it far simpler to manage the cache by deleting only a single reference, but makes it a bit difficult to update the cache by inserting the ID of the post in all expected places such as home page, the community, your profile page etc. As requests do take time in fetching data, not only is this beneficial for our server, this is also beneficial for the user in speeding up the time it takes to retrieve data, an overall win-win for everyone.

There is the disadvantage of content being slightly stale by about a minute or however long a server decides to cache a piece of information, however the trade-offs of performance far out-weigh the slightly stale content the user may have.

Security was a consideration when implementing the cache, specifically given the cache is stored in localstorage and is not difficult to read if one is inclined. The idea was to use the combination of the IP address, username, password, and host-name to generate a unique hash to encrypt the cache. Upon further consideration however, the security issues of local storage comes down to if another user can access your computer and sort through your caches, which at that point we believe WabberJocky posts are most likely the last thing they are interested in stealing. Not only this, WabberJocky is intended to be used in an academic setting for communicating between groups of students and teachers, it’s really not like any truly sensitive information will be thrown around that a user may wish to protect seriously given it is public and free to sign up, although it is unfair for us to assume, some security is better than no security.

Certain information is not cached in order to stay up-to-date as much as possible. For example, searches are not cached as the results must be fresh for the user who is looking up the information. Another is discovering other groups, given all federated requests are based on who is online. There is no point caching this information given if a server goes offline we would be constantly pinging them requests for no reason, it’s best to have fresh information in this regard.

As a performance metric, the main-page takes on average over three runs 3.4 seconds as timed to load in initial posts and fetch the data without the cache. With the cache the page took only 0.6 seconds to discover then fetch data from the cache, a significant decrease in time taken to display data to the user, adding up over time.



```
GOT https://cs3099user-a4.host.cs.st-andrews.ac.uk/api/posts?limit=10&targetHost=cs3099user-a6.host.cs.st-andrews.ac.uk CACHE
ttl 50.542
found ► Array(10) cs3099user-a6.host.cs.st-andrews.ac.uk
```

Figure 3: example of cache being fetched

2.23.2 Continuous Integration and Backend Test Suite - Ben Hawken

To ensure that every new piece of backend code added to the repository was robust itself as well as not breaking any previous features, we created a complete end-to-end set of tests that mocked user experience. Initially we attempted to set up individual unit tests for each method in the backend, however, it quickly became apparent that it would not be possible due to how linked the results of the backend were with the database. If we wanted to test 'editUser' we would first have to ensure that there was a user to edit in the database, requiring a 'createUser' method to be called first in the unit test. Having seen how linked the methods needed to be to obtain results from many of them, we instead opted to commit to linking them and create a User Interaction end-to-end test suite [2].

We used pytest for this which allowed us to create global pytest variables. These variables were mutable and could be edited, updated or deleted as the tests went on, allowing us to test that a password had changed in the database after was called and tested.

We further used a python module called Faker which generated random fake data within the boundaries that we set to be passed into the test methods. This ensured that we had a diverse set of test data that ensured we would not fall into the trap of testing every iteration of code on the same test data. Our worry here was that if one of the functions worked with every very specific set of test data but only that data, we were not testing rigorously enough and that would turn into an issue for us trying to debug later down the line, or an issue for our users if we did not catch it. Generating the random fake data allowed for a variety of usernames, passwords and community names, in turn affording us more confidence that if the tests passed, the code was ready for production.

The way the tests themselves actually function is by sending a request to whatever specific URL is associated with the method we are testing and then asserting that the return status code is equal to what it should be, which in almost all cases is 200. If the request fails for whatever reason, this will return a 400/500 error code and this will cause the test to fail. If a particular method, such as createPost, needs JSON passed in then the JSON is mocked up within the test and given as an argument in the request. This simple style of testing allowed for efficient writing of the code as well as making it easy to understand, while at the same time giving us confidence that the back-end was working correctly if all tests passed. Due to the fact that all of the tests were linked in some way, often if one failed then this caused others to fail as well. For example, if createPost failed then deletePost would also fail because there was no post in the database to delete. This did mean that sometimes it was hard to pinpoint the exact issue, however, as the project progressed, we became more tuned into what the test results meant. Often the earliest test to fail in the suite was the actual issue and if that problem was solved it usually solved the rest of the problems after it.

Initially when we were testing the application, we required the Flask server to be running so that the tests could send requests to it and inspect the status codes. In the early stages of production and testing this fulfilled our requirements, however, later on in the project when we needed to be able to test in different environments, most importantly the Gitlab CI Docker container, it became apparent that we were not able to run the server and then also initiate testing on it within the same container. To fix this, we added a file to the testing directory that created a pytest fixture called client which configures the app for testing. This fixture is called for every test and gives a simple, streamlined interface to the application allowing us to trigger test requests. This meant that the server did not have to be running, instead, all of the tests would send their request to the application

through the client that we had created. This made testing much more system agnostic and allowed for the testing to be used in our Continuous Integration pipeline.

In order to take the usefulness of the testing one step further we decided to set up a Continuous Integration (CI) pipeline in our GitLab for the back-end. This way, whenever new code was pushed to the back-end production branch, it would immediately have the whole test suite run on it to ensure that in changing, adding or deleting code from the previous version, the functionality was still completely robust and ready to be pulled for deployment. If the tests pass then GitLab marks that push with a green tick signalling that it is ready for deployment. If the tests fail, it is marked with a red cross and an email is sent to one of our team members. We could then click on that specific iteration of the pipeline to see exactly where the tests went wrong so that we could fix whatever had changed. This form of testing is in line with Agile software practices [1] and proved incredibly helpful in catching bugs as early on as possible so that they did not pile up and become more difficult to differentiate and fix.

The pipeline itself is setup using a `gitlab-ci.yml` configuration file. This file is essentially a GitLab proprietary version of a Docker config file that allows the user to create a repeatable and isolated environment within which they can use a command line interface to perform tasks. We used a recent python image as the base for our Docker Container. Before any tests were run we used a file called `requirements.txt` and the python command ‘`pip install`’ to quickly install every dependency we needed for the back-end. Due to the fact that the docker container only persists for as long as the commands in the yml file are run, we needed to install these dependencies new every time. Once they were installed it was simply as case of calling ‘`pytest TestCases`’ to start the aforementioned tests on the back-end code. As previously stated, changing the testing code so that it did not require actually running the server meant that we could easily run these tests inside the Docker container.

The only issue that we then had was we needed some kind of database connection for the back-end to update and fetch data from. Initially we tried to use an ssh into the school systems from the Docker container to access the production database and run the tests as if they were happening locally. This method provided a host of problems so we quickly abandoned it. Our better solution was to use a Docker container ‘variable’ which allowed us to have a brand new instance of Mongo DB inside our container every time the tests were run. This presented its own problem, however, because the address to access this container was not the same as the address to access the production database. This meant that we would have to have two versions of our back-end code, one with the correct code to access the test database and one with the correct code to access the production database. We briefly considered using two separate instances of the code. We would test one and if it passed then we would use the production version that was identical to it. This clearly was not an efficient or safe solution and we knew that this would eventually cause inconsistencies in the code between them. After researching we decided to use environment variables instead which completely solved our problem and presented an elegant solution. In the back-end python code we set all of the addresses for databases to environment variables using the python `os` module. These variables are global variables that are stored in whatever computer system you are using and can then be accessed from the command line or, in this case, programs. It was then simply a case of setting the variables in each desired environment to the different database addresses.

In our `gitlab-ci.yml` file which configures the CI pipeline, before the command to run the tests, we used the ‘`export`’ command to set up various environment variables that

would then become global variables in the Docker container ‘environment’. Every time the tests were run, these variables were defined and the back-end code would know it was in the testing environment, attaching itself to the Docker container instance of MongoDB. When the same code was inside the deployment directory in the back-end tmux session on the school systems, different environment variables we had set, which persisted because of the tmux session, pointed the back-end to the actual production database address. This meant we could use the same code in different places and achieve different results by priming those places for the code. We believe this to be an elegant solution that allows for our CI pipeline deliver the exact code that is ready for production with no tweaks or changes. This also allows us to have a completely isolated testing environment that does not affect the production database at all which is a serious benefit. When we were testing in the early stages, the production database would fill up with random test data that clogged up the actual application with unnecessary content, requiring frequent wipes and edits to the database. It also meant that certain tests would not pass because our random data generator would generate the same username as an entry that was already in the database from previous testing, causing the tests to fail (our application does not allow duplicate usernames). By having this isolated testing environment, we know that the tests will never be effected by data that is already in the database and we can work tabula rasa every time.

Overall, the combination of the back-end testing suite, the CI pipeline and the different environments we set up have allowed us to code with confidence in the back-end. Every push is evaluated and this allows us to fix one bug at a time as they come up, not wait until we decide to run tests manually ourselves at a later point and find too many bugs to be manageable. Our system is automatic and flexible because of the CI pipeline and the different environments and we believe this to be an exceptional software engineering feature.

3 Evaluation and critical appraisal

3.1 Functionality - Craig Mckenzie Csoban Balogh

Overall, we believe that the end user functionality that we have accomplished is of a very high standard. We allow users to smoothly use the system, with clear paths to different features whilst keeping a clean layout.

We feel that one of the best things about our functionality is how we blend in our server with other group’s servers. It was a goal of ours from early on that we wanted to offer a seamless experience whether they were viewing a post on WabberJocky or any other instance, and we feel that we have accomplished this. We are also very proud of the caching, as it enables far faster load times than simply having to make requests upon every load, with our solution working even across different URLs.

In terms of what did not go as well, we found that some of the advanced functionality we wanted to support (searching, fuzzy string matching) ended up not being feasible within the scope of the protocol, as well as many of these features slowing down our system when we attempted to implement them. We solved this by passing more things off to the client and operating them in the background, meaning that more resources are used, but we feel this is a worthy trade off to improve the user experience).

If given more time we would have liked to increase the efficiency even more, as at times the system can still feel slow on computers that have low specs, so to solve this we

would likely try to upgrade our server implementation so that more is managed by the back-end, allowing for the client to operate more smoothly.

Not only this, a way of testing the front-end to streamline the process even further would have been ideal. The front-end could use some heavy optimisations in the form of React-Memoisation and saving past renders to then reuse, as well as further increasing scalability by using react-window or react-virtualiser to allow for partial renders of huge lists. This was attempted but the results were poor visually and so was left out commented at the bottom of Display.js.

3.2 Supergroup protocol - Craig Mckenzie

Overall, we believe the supergroup protocol to be a truly exceptional piece of work; it successfully allows for all the groups to communicate quickly and without ambiguity, and the methods we used to develop it as described above helped immensely in ensuring that everything worked properly and smoothly.

We feel that the process was one of the things that went best about the supergroup protocol, as the decision to take our time and produce a robust protocol at the first time rather than rushing and having to amend basic things later meant that implementation was not disturbed by having to go back and respond to changes in core features of the protocol. It was also very good that we set up weekly meetings where we could have free and open discussions about issues we were having with the protocol, allowing us to catch any issues far quicker than if we were simply messaging each other.

As for things that did not go as well, we likely took a far too wide view of things from the beginning of the protocol, making some groups feel overwhelmed. This has meant that in the end many of the groups are still missing what could be considered basic features (post filtering is missing from several groups), and we handled this by adding better error checking to assume that any additional features like filters may not work properly, so we must manage that properly on our front-end.

If given more time we would like to have made it possible to do some sort of live chat across servers, as whilst there is a basic messaging endpoint it is quite weak and could be improved a lot by simply having the ability to connect with sockets across two users across servers, making it feel like a more complete system for users to use.

4 Conclusions - Charlie Don

In conclusion, we feel that as a group we have successfully come together to create an incredibly resilient, usable, and interesting implementation of the required specification. We felt that we have communicated very well throughout, following proper Agile/Scrum methodology to ensure that we were following good practice at all points. We have produced properly commented and error-checked code for both the front-end and back-end and have managed to successfully deploy both as well as the database upon the school servers. All of this can be accessed through the front-end at:

<https://cs3099user-a4.host.cs.st-andrews.ac.uk>

We are particularly proud of both the front-end caching and the CI end to end testing/general deployment, both of which we believe to be exceptional features. We struggled at points with things such as bad data from other groups and successfully finding where errors were occurring, but we were able to develop the solutions described in the sections above to these problems. If given more time we would like to have been able to introduce

more advanced features which the end user would expect, such as the ability to live stream or chat directly with other users across servers.

5 Appendices - Charlie Don

The Appendices of this project can be found within the “Appendices” folder of our submission, which contains the following:

- A full list of all our meeting reports from the second half of Academic Year 2020-2021, the entire period of development after the MVP. This can be found within the “Meeting Reports” folder.
- A group of testing screenshots proving the validity and robustness of our program, showing the program successfully completing normal usage, such as making posts (both locally and federated) and user functions. This can be found within the “Testing Screenshots” folder.
- The protocol which we built as a supergroup for use within the project. The easiest way to use this is to load the “specification.yaml” file into any capable API editor (we recommend SwaggerHub). This can be found within the “Supergroup Protocol” folder.
- All of our previous reports (from submissions 1, 2 and 3) that we have given throughout the course of the project. These can be found within the “Previous Reports” folder.

You can also find our “Source Code” folder within the submission, which contains within it all the code which we use to both run and test the back-end and the front-end. This is structured as:

- A folder for the back-end code, which contains not only the files necessary to run the back-end but also the end-to-end tests that are used to test the implementation. This can be found within the “Backend” folder.
- A folder for the front-end code, which contains all the files necessary to run and use the front-end. This can be found within the “Frontend” folder.

6 Testing Summary - Craig Mckenzie Ben Hawken

6.1 Back-end Testing - Ben Hawken

The back-end tests that we use in our CI pipeline have been very useful. There have been numerous times where small typos have been found immediately due to these tests and this ensures that our code quality has stayed high throughout the development process. The tests were designed in such a way that they emulated a user interacting with the website which gave us confidence in the robustness of the back-end compared to if we had just used isolated unit testing.

Integrating these tests into our CI pipeline proved challenging. If they had just been unit tests, it would have been a simple case of executing the tests in our gitlab-ci.yml file, however, due to the end-to-end nature of our tests and their need for database connection,

this made it more challenging to set up. As mentioned before, this required a number of manual solutions such as creating a pytest fixture, creating a new database in our Docker Container and using environment variables. While this was challenging to solve, it was rewarding to have created an end result that worked so well and fulfilled its exact goal of maintaining a high quality back-end.

One aspect of testing that we were not happy with was front-end testing. The original plan was to have an equivalent front-end test suite using Jest and React Testing Library(RTL) to create mocks of the Reacts Components in our App. RTL would have allowed us to render the DOM inside of the testing suite without having to run the actual application. We could then have used RTL to simulate a series of user interactions with the rendered React Component such as button clicks, data input and screen refreshes. While trying to implement these tests, however, we ran into technical difficulties with the testing technologies not being compatible with our application. There was JSX syntax used as well as plain JS and this then required a compiler called Babel to make both work in unison. The compiler threw errors every time we tried to use even the simplest functions from RTL and none of the fixes or workarounds we found in our research solved the problem. We decided that rather than waste more time trying to fix this issue we should focus on other more important issues and stick with manual testing for the front-end. If we had more time we would have liked to have implemented these front-end tests to ensure we had the same amount of confidence in our front-end code quality as our back-end.

6.2 Benchmarking - Craig Mckenzie

In our benchmarking of the back-end, we found that during times of low load, simple GET request to our server would take around 0.004s on average, with this number increasing to around 0.007s during times of high load (in the context, take high load to be around 40 requests per second). For more complex requests, such as adding or deleting a post, we found that the average time at low load was around 0.008s, and at high load it could go as high as 0.012s, but it would generally stay around 0.011s. All of this can be largely dependant on the speed of the network at the time of the tests/how busy the network is, and these tests were done at around 10:00 (Military time) as we felt that this would be a time of approximate normal usage. All the above benchmarking was done using a simple python program that would send the requests, on however many threads was necessary, and then average out the returned total time of the requests. Sadly, the above cannot be repeated with any consistency for operations involving other groups as there can be wild fluctuations depending on what group the tests are performed against.

6.3 Screenshots - Craig Mckenzie

For in depth screenshots of our functionality, please refer to the TestingScreenshots folder within our Appendices, that will clearly walk you through all of the functionality (the file/directory names are descriptive and detail what the purpose of each test is)

References

- [1] Nico Krüger. *Agile Testing Methodology — 5 Examples for the Agile Tester*. URL: <https://www.perforce.com/blog/alm/what-agile-testing-5-examples>. (accessed: 13/04/2020).
- [2] Camilla Lassen. *Automated End-to-End Testing: What Are The Benefits?* URL: <https://www.leapwork.com/blog/why-you-should-conduct-and-automate-end-to-end-tests>. (accessed: 13/04/2020).
- [3] N/A. *React Usage Statistics*. URL: <https://trends.builtwith.com/javascript/React>. (accessed: 11/04/2020).
- [4] Ben Schmaus. *Making the Netflix API More Resilient*. URL: <https://netflixtechblog.com/making-the-netflix-api-more-resilient-a8ec62159c2d>. (accessed: 12/04/2020).