Final Report

Group A5 — Supervisor: Ozgur Akgun

April 2021

Abstract — 180008859

In this project, we (as a super-group) developed a specification and protocol for a federation based social media application. Our team then implemented the protocol into a Java back-end and a web-based front-end. The Java back-end utilizes MongoDB and Spring to handle the database of posts, communities, and users. Spring is also used to handle the HTML based networking that connects the back-end to both the front-end and to the back-ends of other teams' applications. Our application also includes functionality to block users and to create many different permission levels in a community.

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 6,192 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 **4**
- 1.1 Terminology . . . 4
- 1.2 Problem . . . 4
- 1.3 Overview . . . 4
- 1.4 Usage . . . 5

2 Project Details — **6**
- 2.1 Functionality . . . 6
- 2.2 Beyond The Required Features . . . 7
- 2.3 Highlighted Features . . . 7
  - 2.3.1 Implementation Technology Choices . . . 7
  - 2.3.2 Organizational Technology Choices . . . 8
  - 2.3.3 Library Choices . . . 8
  - 2.3.4 Database Choices . . . 9
  - 2.3.5 Federation JSON Object Choices . . . 11
  - 2.3.6 Frontend Library Choices . . . 11
  - 2.3.7 Structural Choices . . . 11

3 Evaluation and Critical Appraisal - **12**

4 Conclusions **13**

5 Scrum / Agile Development - **14**

6 Plan Changes - **14**

7 Super-group Interaction - **15**

8 Testing Summary - **16**

9 Library acknowledgements **20**

1 Introduction — 

1.1 Terminology

Source: https://fmckeogh.github.io/cs3099a-specification/
Written By: Super-Group Collectively.

Instance a currently running server process, receiving HTTP requests / communicating with DB/etc.

Server the logical unit of the fed system, consisting of 1 or more instances and a single logical database.

Local Server the server a user is registered with, and accessing with that server's front-end.

Remote Server a server the user is not registered with, and interactions will be proxied through the local server using the protocol we are discussing.

Community a collection of posts related by a common topic on a single server (eg. a subreddit).

Post a basic unit of content in our protocol, can be nested and front-ends may choose to render top level posts as primary pieces of content, with nested posts rendered as comments.

User entity representing a student/member of staff registered with a server. Users cannot be transferred between servers, but can view and interact with posts and communities on remote servers through their local server.

1.2 Problem

The problem that the group set out to solve was to create a system for and implementation of a federated social network that could be used by a university. By creating a federated system, the university would have the ability to have a federated server for each department, class, school, etc. The federated system also allows for multiple universities or third parties to create instances and interfaces for those instances that could all connect, creating a system of connected servers, each of which being wholly controlled by the entity that it is representing in the federated system.

By creating a protocol as a super-group and implementing our own instance that is able to interact with other implementations of instances, we fully solved this problem.

1.3 Overview

Our instance is made up of four sections, that each connect to 1 or two other sections to comprise the full implementation.

Database: MongoDB is used to run a json-document-based NoSQL database. All the posts, communities, and users that are stored by our instance are stored in this database. The database also holds information on permissions in communities, user block-lists, and account information for the users hosted on our instance. The database is hosted on the A5 pseudo-user account.

Database Interaction: Spring's MongoDB integration is used to read and write from the database. Object templates are controlled by Java classes, with the methods for searching the database being controlled by the auto-wiring function of the Spring library, which allows for method signatures to be created in Java Interfaces which are then used to automatically create working database search methods without the need to create code manually. The auto-wired methods are then abstracted over with manually written methods that allow for extra checks on permissions, block-lists and complicated search parameters that would be very complex or impossible to create using Spring's auto-wiring. An example of a complex search parameter is that posts can be filtered by any combination of their community, post date, author, their author's host server, parent post, or content type.

Front-End Interaction: Spring is used to map HTML requests from our front-end and from other servers in the federated system to methods. The methods that handle the HTML requests utilize the a database interaction interface to write and read from the database without needing to deal with user permissions, block-lists, or how the database system is designed.

Front-End: We have a frontend that allows a user to easily make use of our program as well as to access other instances and interact with them in accordance to the protocol. The frontend has several convenience features such as as automatic reloading of the posts when new posts are made as well as error messages to convey to users when things have gone wrong.

1.4 Usage

Once the program is built to a jar perform the following configuration.
Add a directory `./keys` which contains `publickey.pem` and `privatekey.pem`.
Create an `application.properties` file to override the default configuration.

`spring.data.mongodb.uri` should be set to the connection string of the database.
`spring.data.mongodb.database` can be set to override the default database name.
`fedserver.host` must be set to the hostname that the sever will be run on.
`server.port` can be set to override the default port number (22879)

Then the server can be run by executing the `.jar` file.

2 Project Details [redacted]

2.1 Functionality

**This section was partially copied from the MVP re ort section written by [redacted] nd the Progress Report section written b [redacted]*

User Registration Users can be registered locally with a username and password. These credentials, and session cookies, can be used to log in to gain the ability to create posts and comments. Usernames uniqueness is enforced and passwords are stored in a hashed and salted form. As per the super-group protocol, users are local to a server, and cannot be logged into from another server.

Communities Users can create communities with descriptions and titles. Community pages have semi-random banner pictures based on their names. Each locally hosted community has a list of users that are members of that community. A user is considered to be a member of a community if they have viewed the community page.

Posts Logged in users with the permission to do so can create posts consisting of a title and body text on the community who's page they are viewing. Posts can be edited by users or deleted with the correct permissions. Users can also leave comments, which act as posts except their parent is a post and not a community. Posts support markdown, although our implementation does not permit the creation of markdown posts, and special Unicode characters, as well as plain-text.

Web Interface We have a web interface that allows for the registration and login of users, and the creation of posts and comments. It also uses cookies to store the users session so that they do not have to login every time. Once a post is created, it can be edited and deleted by any user with the correct permissions in the community the post was created in.

Protocol We have an internal API set up conforming to the Super-group protocol which allow us to support inter server communication. There is also an API set up for the web application which is used as a GUI for users.

Permissions Permission levels, stored as integers, are handled in the back-end and are visible to users with admin level or above permissions on the front-end. Any user with the correct permission level has the ability to change the permission levels of users by viewing the members of the relevant community and changing the permission sliders of any users that they are authorized to change. Permission level defaults are stored in a configuration file, and hard coded to make sure that in the absence of a valid permission configuration file required permissions are still initialized. The current permission types are viewing and making posts, editing and

deleting your own posts, commenting on posts, deleting and editing another user's post, and deleting a community. Creating a community can be done my anybody as permissions are handled by each community.

Back-End Only Features **These are features that are working on the back-end, but do not have any GUI elements in the front-end to support them.**

Users have the ability to block other users. This hides all the posts and comments by any user that is blocked. All comments can trace their root to a post or comment from a blocked user is hidden as well. Users are also able delete communities that they are owners of, delete and edit posts by other users if they have the correct permissions, and change their own password. The required permission level to perform any action (as listed in the above section) can also be changed by any user with the correct permission levels.

2.2 Beyond The Required Features

The projects goal was to front-end and back-end for an instance in a federated social-media system. That goal was achieved, and has more features than were required. In addition to all the required features, our system handles posts which contain not only text, but also images, markdown formatting and Unicode characters (including emojis and special characters). Users are not only able to be registered, but their passwords are handled securely, they are able to have different permission levels for each community they are a member of, and users are able to block other users. Blocking a user filters out all posts by, and replies to posts by, a blocked user.

2.3 Highlighted Features

2.3.1 Implementation Technology Choices

This section was partially copied from the Deliverable report section written by* [redacted]*

We made the decision to structure our product as a web application due to the advantages in portability and compatibility. Web applications are also the standard for social media applications (Facebook, Twitter, Reddit, 4chan, etc.). By committing to a web application format, we had to commit to using some combination of HTML, CSS & JavaScript, but we decided that that limitation was a non-issue as more than half of the team had some level of proficiency in these tools and it allowed us to not be limited in our selection of back-end languages/technologies.

To decide on our back-end language, we came up with the following requirements: the language needed to be one that most of the team are comfortable with and we didn't want to limit our library use with our language choice..

After some initial discussion we narrowed down our choices to three languages.

JavaScript

JavaScript running on *Node.js* is a common base for web applications and is purpose built for this kind of work. There are also a large number of frameworks such as *React.js*[1] (briefly introduced in CS2003) that complement this style of project. However as a team we decided that as only half of the team has experience in JavaScript, and those skills will already be put to use on the front-end, it would make sense to use one of the other options.

Python

Python was also considered as it was a language that most of the team had experience in from CS2006 and is reasonably straightforward to learn (in comparison to JavaScript). Python has a plethora of libraries available and uses a directory based package structure which helps organise the code. There were no obvious downsides to Python besides not everyone on the team having experience with it.

Java

In the end we opted to use Java as it best fulfilled our main requirements. It is the language that all of us are most familiar in, has a large variety of libraries available and also has a package system to allow modular development. We also make use of Gradle[2], one of the two most commonly used Java build systems.

2.3.2 Organizational Technology Choices

We originally planned to use mostly excel as a user-story management system and as a way to organize tasks, but we found that augmenting excel with Trello as our week to week sprint work-log made organization and planning easier and quicker.

2.3.3 Library Choices

We initially chose to utilize the Spring framework simply for its hosting of static web content along side its support for a REST JSON API. Later in the project, we decided to also utilize Spring's database integration with MongoDB and it's templating features.

Despite being Java, Spring follows a different process than we were used to for Java development. Rather than doing composition by explicit instantiation components are linked using the `@Autowired` annotation which instantiates the components and attaches them as required, somewhat similar to an automated version of the singleton pattern. This was confusing at first as we were trying to use the processes we were more familiar with which led to some things not working as expected. However, once it was working, it worked quite well.

[1] https://reactjs.org/
[2] https://gradle.org/

As Java's built-in encryption library does not handle the key format required by the federation protocol (PEM), we needed to pick a third party library to use for the authentication section of the federation API specification, which was added relatively late in the development process. BouncyCastle[3] was the obvious choice as there were code segments available online[4] that contained the basics of what needed to be done for the authentication of other federated servers. BouncyCastle also has built in support for PEM, which is required by the super-group federation specification. In addition to making it fairly simple to create a super-group specification compliant authentication system, BouncyCastle is also lightweight and distributed using Maven, which allows Gradle to access it too.

2.3.4 Database Choices

We made the decision to use MongoDB as our database as it would provide the greatest benefit to our project, due to having a multitude of easy-to-use tools, a well-documented API, and the compatibility with JSON files. Using a NoSQL database allowed us more easily handle the lists that were built into the federation API, and the compatibility with JSON was originally meant to allow the JSON based federation protocol to integrate directly into the database. However, we found that it was actually better to convert from JSON objects to Java objects and back again, instead of keeping objects in JSON form. Therefore the choice of MongoDB didn't have a large effect on the project, as any other NoSQL database would have worked equally well, at least in terms of the file storage type.

Having a NoSQL database was also advantageous for agile development as it meant that small changes could be made to object types just by modifying the interfaces, which is discussed later.

Within the MongoDB database there are collections (equivalent to tables in SQL) for each of the following types; posts, communities, local users, user block lists, user permission levels, community permission settings, and foreign servers. Due to wanting a more scaleable application, when making all decisions to do with database storage verses database speed we chose to optimize speed as in general. This is because it is generally much easier to add more storage to a server than to change the way a database's architecture is optimized internally without risking outages or loss of data.

A single post document in the database contains a UUID (the primary key), the post's community, a title, a parent (either a post UUID or community ID), the author, the times that post was created and modified, a list UUIDs containing the post's children, and a list of content contained within the post's body. The body of a post can be any combination of markdown and/or text. Post content is stored as a sub-object in the post object as it can be multiple types, which need to be handled differently when storing and rendering them.

[3] https://bouncycastle.org/

[4] https://artofcode.wordpress.com/2017/05/26/rsa-signatures-in-java-with-bouncy-castle/

Foreign servers are stored in the database to allow for the `/fed/discovery` endpoint specified in the federation API to retrieve a list of known foreign servers. This is done so that other servers can more easily discover the whole federated network. By using recursion on the discovery endpoint, a foreign server could map all servers that are known to any other server. The read impact that this feature reduces is quite large, as needing to search every post for its author's host would be very computationally intensive if there were a significant number of posts.

The list of all the users on a community can be accessed by any user of that community. Instead of needing to check through every post in a community to get all the members of a community, the members of a community are stored in their own collection. This list of users is then easily accessible to show all other users of a community who else is in their community. It is also used to allow for a user with the correct permissions to change the permission of any other user of a community without needing to remember their name and instead being able to find it in a list. Every user that views a community is given a default permission level in that community (defaulting to 0), so as having a permission in a community and being a member of that community are inherently linked, both community membership and a user's permission level in that community are stored in the same collection in the database.

To handle permission storage for each community, there is a collection that stores permission handlers for each community in the community collection. Communities and permissions are stored separately because they are often not needed by the same source. As permission containers have the potential to become quite large objects if the permission system is expanded, storing a large permission object in a relatively small community object would make queries over many community much more read intensive on the database. Therefore, the systems are separate as communities are almost always queried to create a list of names, and permissions are only required one at a time, increasing speed at the small cost of some duplicate data in the form of the community id attached to each permission container.

For similar reasons, unique user permission levels are not stored attached to community permission handlers. They are often not required by the same queries and detaching them from each other allows for more efficient access of the database at the cost a slight increase of storage requirements.

User block lists are stored separate of users for the same reasons as permission levels not being attached to a user. They are not often not required by the same API requests, so it is more efficient to detach them and have them be accessed separately.

Due to users only being unique on to local servers, users are handled using both their username and host address. This is done wherever users are stored in the database, so users are stored using this method for permissions and for handling blocked users.

2.3.5 Federation JSON Object Choices

Each object defined in the federation API is also defined as a Java Object in our code as a schema. These objects are used to convert the JSON in the API to java objects which could be handled more cleanly by Java typing requirements. It also allowed for easy constraint checking, as each schema object's constructor has all the type and formatting constraints required built in, which means that no constraint needs to be explicitly handled by the REST API handling code.

All the schemas are also tested by unit tests, which check constrains and consistency within each schema.

2.3.6 Frontend Library Choices

By choosing Java as our back-end we ruled out using a full stack JavaScript framework for our project. This left us with a selection of front end frameworks to choose from both for functionality and aesthetics. When prototyping we decided to use *Ractive.js*[5] as this was a technology that one member was already comfortable in. *Racitve* is a live templating framework that supports dynamic content. We had initially decided to consider other options after the initial prototype however during the process we developed our skills with the library and discovered that it actually contained several features that would prove useful while also remaining modular. The key features that we liked were the components feature that allowed us to break up the front-end code into a smaller number of specialised components such as a new comment dialogue and a component for post listings. It also has a nice feature that lets it work with asynchronous requests (the spinner when waiting for posts). Finally although the similarly named and aforementioned *React.js* supports similar features it requires a preprocessing step converting its native `.jsx` files to pure JavaScript which we felt didn't fit with our java based deployment pipeline.

For a CSS framework we chose *Materialize.js*[6]. Although the standard for a new web application is usually *bootstrap* we decided that we liked the slightly more defined style of *Materialize*. We were able to pair *Ractive.js* components with the library effectively especially with components such as cards which require a small but often repeated amount of HTML tags.

2.3.7 Structural Choices

When starting to create the database handler and the API handler parts of the code, we chose to develop them separately from each other, mainly as we wanted to separate the work out and it made sense for different group members to work on them individually. To facilitate this, Java interfaces were used to define how the API handling would interact with the database handler before it was fully implemented. This method was also used for the API handling, allowing a defined API handler to be worked towards before it was fully complete.

[5] ractive.js.org
[6] https://materializecss.com/

The interfaces allowed changes to be made, mostly as the federation API was changing, to the required methods without needing to interfere with in-progress or messy code.

Java Enum types are used to define the allowed permission levels currently handled by the server to reduce the amount of incorrect permission names that are used in the code. By doing this, we are able to retain more internal consistency with our permission system and to allow for changing the names of existing permissions or add new ones with little to no hassle or need to look through the code for all the uses of an arbitrary String with the permissions name.

3 Evaluation and Critical Appraisal - [redacted]

Evaluation

Our team has created a federated social media targeted at a university environment which was our original goal and contains all of the basic features specified. It functions correctly as part of a federation of similar platforms and can interact with them to both receive and request data. While it meets these goals our platform would be well served by some amelioration of the front-end as some parts of features were left disconnected to the front-end.

Critical Appraisal

The concept of a federated architecture was first proposed in the early 1980's by Denis Heimbigner and since then this architecture has been applied to many example systems. With one such system being our project, we will take time here to compare and contrast our project with others of its kind. One such example is **Fediverse** which follows a similar structure and goal to this project, a group of connected independent social network nodes with a commonly agreed upon protocol for communication. They have the additional advantage of a single website that collects all of their servers and provides easy interface between them which is a great feature and a potential future step for our super-group. Much like ours all of the individual group involved have their on implemented social medias that function in many programming languages and with many unique components. Fediverse opens an additional level of freedom by utilising 5 protocols in total, all for different purposes in contrast to our single protocol. Our projects also predictably differ in completeness as Fediverse has been in development for around 12 years. One key difference in the projects is their goal, Fediverse is not aimed at any particular environment or purpose unlike ours; so to supplement this evaluation we will also compare it with **GARR**. In their own words *"The GARR Cloud Platform offers cloud services to the Italian academic and research community. GARR coordinates a federation of clouds, located in national datacenters owned by members of the GARR community, which participate to the federation by sharing resources and services."*. In addition

to being a more recent example GARR is aimed specifically at an academic community and works upon the same principles as our own. It has its on custom designed protocols for inter-server communication and shares key ideas such as user/admin privilege system. A paper discussing GARR in more detail and which was of inspiration to us can be found in the bibliography[1].

4 Conclusions - [redacted]

UniRoam is a university social network that is a member of a federation of similar platforms. Together we provide a decentralised university social network that returns agency to its users. Each server provides a node in the network and UniRoam is one of the platforms that these nodes can run. Anyone can run their own node, giving control back to users over their data, censorship and experience unlike traditional social media. On UniRoam users can create their own communities, share posts and comments and interact with other nodes as part of the network. This project was largely unlike anything any of us had created before, so we are quite proud to say that we accomplished the majority of what was outlined in the practical even while being short a group member. We are also quite proud of our involvement in creating the super-group protocol, as it is never easy to coordinate between so many groups. Each us learned quite a bit throughout the course of this project. Each of us learned more about some aspect of coding, whether that was working with databases, networking, or working on front-end. In addition, we all became quite familiar with working within an Agile environment. Despite all that we accomplished, our project is unfortunately not perfect. One of our biggest issues was with debugging our inter-group communication, as we can never be quite sure whether the issue was caused by our server or the server we are connecting to. Our product also suffered from a week-long forced-hiatus, where we were unable to seriously develop code because we had a rather severe issue with our continuous integration and integration testing. We were able to solve the issue, but it did eliminate a rather significant chunk of time from our development schedule. As proud of our program as are, each of us acknowledges that there is certainly more work that can be done. If we had the time, the first direction we would go to improve our product is to implement more features. Some of the 'stretch features' we wanted to include were: an up/down-vote points system, private chat-rooms, the ability to react to posts/comments with emojis, and setting nicknames for users. After implementing these features we could look towards hosting our server on a more permanent basis, and even expanding our product to include students at other universities.

5 Scrum / Agile Development - [redacted]

Throughout this project our teamed followed an agile development model, which entailed a more fluid development process that was new to us. We quickly settled in to this new workflow and structured our work according to its principles. Our work focused around short periods of focused work called sprints, the work to be completed in these sprints was decided in scrum meetings which marked the beginning and end of the sprint. To supplement scrum meetings we had shorter stand-ups in which we discussed our progress and challenges and altered our workload if necessary. The content of these scrums was kept track-of through a Trello board. One of the strengths of this methodology is its flexibility, which we would come to employ in the second semester as we moved from 2 week sprints to 1 week sprints. Our move to shorter sprints, enabled by agile, allowed us to accommodate for challenges the team was faced with one of which was timezone conflicts. In the second semester our team was spread across 3 time-zones which complicated meeting times and so part of our move to shorter sprints was to make these meeting of more significance and reduce their frequency. While agile did provide a partial remedy to his problem, it in part caused it; Agile encourages more frequent shorter meetings to keep the team in-step with group progress and to help catch problems sooner. It provided our team a healthy freedom of choice in what tasks we undertook and (mostly) in what order we completed then which undoubtedly had a positive impact on our creativity and productivity. However it also have us the tools to deliberate over difficult problems or potentially focus on simply completing lots of tasks rather than achieving a practical goal. Overall Agile, while effective in creating a flexible and nonrestrictive workflow, must eventually bend to real deadlines and whether the correct tasks have been completed by then was more a matter of work ethic than principle. Our group was left with a positive impression of Agile but noted that it works best when short regular meetings are possible, unlike in our current situation. We would be likely to use it again in the future given we could make tweaks such as breaking up long sets of sprints with smaller feature based deadlines.

6 Plan Changes - [redacted]

When we started the project we were unfamiliar with the spring framework and we were using it solely for its ability to host static web content along side supporting a REST JSON API. Over the course of development we started to explore the wide array of functionality that spring provides including templating and database support.

One of the main drivers of this exploration was that we wanted a secure login system for our system. It is often said that you should not write your own security where other pre-built and well tested options are available so we cast aside our primitive login cookies and investigated spring security. It seemed to be a good and well featured system which would allow for user sessions as well as

support for salted passwords which is good for security however, to use it most simply it relied on having a database accessible though spring. We decided that on balance it would be worth rewriting some of the code we already had as the abstractions and query support provided by the system were much easier to use and extend than our previous system.

As a result of the move to using springs inbuilt data framework we had to get rid of the manual connection to the database. The major downside this brought is that it meant that we could no longer run database functionality outside the spring environment. This meant that the initial integration tests for the database we had which relied on this no longer worked. Although it would definitely have been possible to rework the tests to work within spring the process did not appear to be simple and required a time investment that we didn't deem necessary as lots of the functionality we had been testing was now the responsibility of the data framework.

We also decided to try to implement continuous deployment. We ran into issues when we realised could not get an SSH key for the pseudo user we wanted to deploy to. Instead we made a script on the pseudo user that could be run to fetch the most recent build of master and extract and run it. Although not quite as elegant as the standard process this still substantially eased deploying a live version we could test against.

7 Super-group Interaction - [redacted]

This section was partially copied from the Progress & Plan Report section written by [redacted], the MVP report section written by [redacted], and the Progress Report section written by [redacted]

We assigned a representative that would be the bridge between our group and the super-group. Each week, the super-group had up to an hour long meeting that our representative attended. During these meetings the representatives of each group talked about the protocol, any concerns about implementations, and all other questions or comments on how the groups were doing. At the end of the first semester, we switched representatives as the first semester rep was unable to attend the second semester meeting time.

For the first several meetings, the super-group had fairly disorganized meetings where the reps discussed anything they wanted to about the practical. After the first few meetings, the super-group decided to move towards a more organized approach with a one of the representatives acting as a moderator for each meeting. The moderator would decided when to talk about each topics brought forward by the representatives, and who would be able to talk on these topics. Topics were usually posted in the Super-group A Microsoft Teams group, and anyone who wanted to talk on a topic would "raise their hand" via the Teams video call interface. These organizational measures made communication and discussion much smoother during the meetings. When not in the weekly meeting, discussion via text chat in Teams was also a frequent method for discussion,

but most final choices where made with all representatives in the weekly video meeting via a majority vote.

The protocol was finalised and informally frozen on the week of February 17th 2021. One of the main principles behind the super-group protocol was allowing for people with varying levels of completeness to communicate, as such although additional features were able to be supported by custom content types other features that would have required additional endpoints and functionality within the protocol such as live streaming were excluded. This approach to system design was beneficial to us as it meant we could still maintain compatibility despite not having time to implement some of the more advanced features that other groups managed to implement.

As we started to implement the authentication, we started testing with another group (A8) to confirm that our authentication was compatible with theirs. Their help was an incredible boon for the success of finding some more obscure and fiddly bugs in our authentication handling. With their help, we were able to quickly bring our authentication online once it had been attached to our JSON REST API handling code. We were unable to implement the verification of incoming requests due to time constraints however this does not impact our ability to communicate with other groups, just leaves open a potential gateway for attack.

The protocol specification can be viewed here [7].

8 Testing Summary - [redacted]

Throughout our development process we ensured to test our code for robustness and correctness at each step. We have a suite of 77 unit tests that test most local schemas with current coverage of 91% however much of the database abstraction and session code is untested for the time being. Our group employed Continuous Integration for the majority of the time that we were working on the codebase, which ensured the health of our code through-out its development by ensuring that code passed all tests, built and deployed before it could be merged. Integration testing was handled manually as we were strained for time and because the matter itself proved rather difficult. The primary difficulty being with which server to test, other servers have differing levels of implementation, go offline, have strict authentication or could also have issues themselves that would impact the consistency of our results. To test out our ability to request data we sent our requests to a variety of foreign servers within our super-group and monitored their responses. To test our ability to respond to requests we used an instance of our own server to send requests to which will at least test our consistency with ourselves. A list of the integration test completed and their status can be found following this:

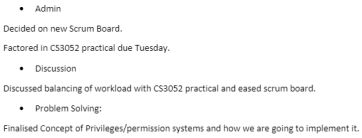
[7] URL: https://fmckeogh.github.io/cs3099a-specification/

Feature	**Test**	**Result**	**correctness**
Foreign communities visibility	visit a foreign community	Their content can be viewed like ours	✓
Foreign community Interaction	Post on a foreign community	A new post request is successfully made	✓
	Delete a post made on a foreign community	A delete post request is successfully made	✓
	Edit a post made on a foreign community	An edit post request is successfully made	✓
	Comment on a post on a foreign community	A comment request is successfully made	✓
	See other users post containing markdown	Markdown content is displayed correctly	✓
Responding to foreign server requests	Receive a new post request	A post is created on the community from the foreign user	✓
	Receive a delete post request	A post is deleted on the community provided they have permission	✓
	Receive an edit post request	The targeted post is edited	✓
	Receive a comment request	A comment is made on the targeted post	✓

End of Table

Appendices - [redacted]

The interim progress report our group wrote in February can be found in the folder with this report and is titled *ProgressReport*. A sample of our weekly scrum meetings notes can be found below and can also be found as word documents in the accompanying folder. A copy of the **Product Backlog** can be found at the end of this document.



- Admin

Decided on new Scrum Board.

Factored in CS3052 practical due Tuesday.

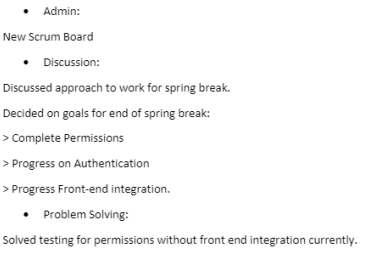
- Discussion

Discussed balancing of workload with CS3052 practical and eased scrum board.

- Problem Solving:

Finalised Concept of Privileges/permission systems and how we are going to implement it.

Meeting Notes 8th of March.



- Admin:

New Scrum Board

- Discussion:

Discussed approach to work for spring break.

Decided on goals for end of spring break:

> Complete Permissions

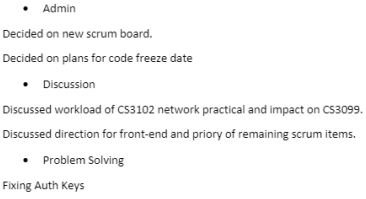
> Progress on Authentication

> Progress Front-end integration.

- Problem Solving:

Solved testing for permissions without front end integration currently.

Meeting Notes 22nd of March.



- Admin

Decided on new scrum board.

Decided on plans for code freeze date

- Discussion

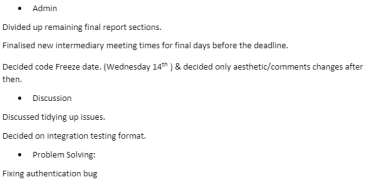
Discussed workload of CS3102 network practical and impact on CS3099.

Discussed direction for front-end and priory of remaining scrum items.

- Problem Solving

Fixing Auth Keys

Meeting Notes 5th of April.



- Admin

Divided up remaining final report sections.

Finalised new intermediary meeting times for final days before the deadline.

Decided code Freeze date. (Wednesday 14th) & decided only aesthetic/comments changes after then.

- Discussion

Discussed tidying up issues.

Decided on integration testing format.

- Problem Solving:

Fixing authentication bug

Meeting Notes 12th of April.

Priority	As a ...	I want to be able to ...	so that ...	Rough Components	Refined components	Implemented
0	Everyone	Get posts from a database	to retrieve data			Y
0	Everyone	Put posts into the database	to store data			Y
0	Everyone	Make an account	To participate in the community			Y
0	Everyone	Have a user interface	To interact with program			Y
0	User	Comment on a post	share their thoughts			Y
Priority Key						
0: Required for MVP	Everyone	Verify login attempts	Keep my account secure			Y
1. Suggested for Final Product	User	Comment on a comment	share their thoughts			Y
2-3. Additional features	User	edit/delete their post	fix a typo or error	post edit/delete requests	Maybe and 'edited' tag?	Y
1	User	edit their comment	fix a typo or error	comment edit requests	Maybe and 'edited' tag?	Y
1	Admin	Lock posts on a sub-community	Prevent any further user interaction with the post			S
Implementation Key	Admin	Delete another users Post	I can maintain subcommunity integrity	Admin operations/privivilages		F
Y: Fully Implemented	User	Host a Sub-Community on my home	Other Users can view posts on in my Sub-Community			Y
F: Needs Front-end integration	User	View posts on other Sub-Communities	I can view and interact with the posts			Y
S: Scrapped Feature	User	Create links to share a page	I can share a page with others	unique post/community links (use IDs?)		Y
1	User	post on other Sub-Communities	other users can view and interact with my post			Y
1	User	change my password	To keep my account secure			F
1	User	Interact with posts using a voting/po	To curate user content	Need password change request	Atomicity in changes	S
1	Admin	Ban/unban a user	I can moderate the subcommunity	Ban System	Limit to Admin	F
1	Admin	delete an inactive sub-community	no space is wasted on inactive sub -communities	Ability to delete community	Limit to Admin	F
1	Admin	Delete user comments	To moderate user interactions			F
1	Admin	Give/remove another users admin pri	To control admin privilage access			Y
1	Everyone	interact with the program intuitivly	I can easily understand my interaction with the program.	Better UI		Y
2	User	Have an Account Page	other users can view information about me			S
2	User	Change nickname	Be more easily recognized by fellow users			S
2	User	Message another user	talk privately			S
2	Owner	Remove an Admin's privileges	To prevent admin privilege misuse	Sub-Community Hosting/ Posting system		F
2	User	sort posts	easier to find active posts; sorting by comment count and rel	Post comment system		S
2	User	Edit my post	To fix typos or make corrections	User account System	Registration/Log-in	Y
2	Owner	Transfer ownership	To change who own the subcommunity			F
2	User	Block/unblock another user	To allow users moderation of user interaction	User Block list on server/client; filter posts out by blocked users		F
2	User	See Markdown	See decorated posts			Y
2	User	Send Markdown	Decorate posts			F
3	User	attach files to my posts	I can share relevant files with other members	Graphical User Interface		
3	User	Set a different nickname for each com	be known as different name in different groups	Server data sharing through communication protocol		
3	Admin	Identify a user regardless of nicknam	Perform Admin operations unhindered by the nickname system			
3	Admin	Talk in an admin chatroom	Communicate with other subcommunity admins			
3	User	save posts	I can easily find posts that I've liked			
3	User	Have unique subcommunity nicknam	To be able to have a unique identity within a subcommunity			
3	User	Message other users privately	Communicate with other users privately			
3	User	Open an admin support ticket	To request support from admins			
3	User	Subscribe/unsubscribe to a subcomm	To get notifications on new posts from that subcommunity			
3	User	Have an account nickname	I have a unique identity			

Product Backlog

A native intractable version of this can be found in the accompanying folder.

9 Library acknowledgements

Name	Link	Usage
Spring	spring.io	Backend framework
Ractive.js	ractive.js.org	Frontend JS Library
Materialize	materializecss.com	CSS Framework
Marked	github.com/markedjs/marked	Markdown Processor
GSON	github.com/google/gson	Java to JSON converter
BouncyCastle	bouncycastle.org	Cryptography

References

[1] Attardi G. and Di Martino B. (2018) 'Using Federated Cloud Platform to Implement Academia Services for Research and Administration', 2018 32nd International Conference on Advanced Information Networking and Applications Workshops (WAINA), Doi: 10.1109/WAINA.2018.00121