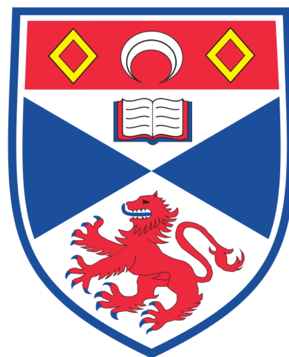
CS3099 Software Engineering Team Project

Submitted: 15/04/2021

Biotin: A federated social media platform



University of St Andrews

Group B7

Members:

Kazio Wilowski

Supervisors:

Angela Miguel + Ian Gent

Abstract

Our goal was to implement a federated social media platform for University use. We adopted the agile development methodology. As a supergroup we developed a common protocol to allow for sharing of content and authentication. The protocol acted as a minimum specification which groups could add additional features to. For example, we included a moderation system. This highlights the key feature of federation which is independence of instances.

This report outlines how we developed the project, what our solution looks like and why we made the choices we did, and includes a critical evaluation of the project as a whole.

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 13,915 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

Abstract 2

Declaration 3

Contents 4

Introduction 5

- Background 6
- Aims and Requirements 7
- Development Timeline 8

Project Details 10

- Overview 10
- Technology Stack 12
  - React 16
  - CSS 16
  - Loopback 4 17
  - Digital Certificates and Inter-Group Communication 17
  - User authentication using JWT 18
  - MongoDB 19
  - GitLab, nginx and running on the host servers 19
- Features 20
  - Supergroup features 20
  - Extended features 22
  - Extension features proposed but not implemented 22
- Changes in Plan 24

Testing Summary 26

- Backend 26
  - Methodology 26
  - Authentication, Navigation and Basic Use 27
  - Inter-Server Certification 27

Voting 27

Authorization 27

Frontend 27

Continuous Integration (CI) 28

Software Development Methodology 29

Tools 29

Use of Agile + Scrum 29

Supergroup Interaction 33

Evaluation 34

Comparison to real-world technologies 34

Backend 34

Frontend 36

Agile & Scrum 37

Supergroup Interaction 38

Conclusions 39

Acknowledgments 40

Appendices 41

Appendix 1: Example of inter-server communication header construction 41

Appendix 2: Running Postman tests 42

Bibliography 43

Introduction

(180012847)

Background

A social networking service describes an online platform in which people may interact with one another, primarily through the sharing of ideas and information. Currently, there exist numerous social networks including Facebook, Twitter and Reddit, and these are actively used by a large majority of the population around the world.

Federated social networks describe a particular decentralised implementation of the above service, where “the leading federated social networking software is open-source” (Esguerra). This allows for the creation of distinct social network ‘providers’ that follow the same protocols, so that users within the federated social network may interact with each other, regardless of which providers they use to access the network.

Advantages brought by this federated system include flexibility, transparency and a more distributed system of authority. Apart from the increased diversity of choices offered to users wanting to join the network, they address several concerns associated with centralised social networks, which typically use private proprietary software. For example, a company providing such a service may have incentives to handle a user’s information in an undesirable manner, while restrictions may be enforced by (or upon) the company that excessively limit the extent of interaction between users.

The aim of the Junior Honours project this year is to build a federated system in a similar manner as described above, targeted for use in a university environment. Each group in the supergroup would develop their own server hosting users and ‘sub-communities’ of their own, while still being able to access the ‘sub-communities’ and associated content in all other servers.

As a result of this, the structure of this federated system took a lot of inspiration from the structures of communities in this university. It was ultimately decided that each server would host a number of ‘forums’, each with a number of their own ‘sub-forums’, reflecting how communities in St. Andrews could be nested in overarching categories: Specific modules within a department, various teams of a sports society, or halls in the context of accommodation.

The social features of the system were then derived from currently existing social networks. While a federated system developed specifically for a university has not been made before, social networks with similar purposes already currently exist. Reddit, in particular, allows users to have extended discussions in ‘subreddit’ communities focused on certain subjects.

Common features of the Federated system were therefore based on the discussions in ‘subreddits’. Users could make posts in subforums, comment on posts and other comments, and ‘upvote/downvote’ each kind of submission - essentially contributing to a score which could decide a post’s priority. Individual users would also have information of their own for others to view, including a profile picture, description, and a list of all their posts or comments across servers.

Aims and Requirements

The aim for each group is to develop a system that allows users to interact with one another in a university-friendly environment, over a web-based interface. The system should have enough flexibility to facilitate discussions on various topics involved with university life, including its academic, social and community-oriented aspects.

Meanwhile, the general aim of the module itself is for members to work effectively as a team while developing software, and in particular, organise the development of the system through following the Scrum methodology and incorporating Agile practices.

Several initial requirements were set in the first hand-in, essentially noting the basic features of a decentralised online discussion forum: Registering users, posting/commenting on 'articles', and a protocol allowing content to be propagated between instances. Only vague guidance was provided beyond this point, and students within the supergroup needed to discuss how to develop the specification to satisfy the project aims.

As supergroup discussions progressed, it was agreed that an API protocol would need to be specified so that each group's system could communicate properly with one another. The endpoints proposed ultimately led to a common set of requirements between groups, including a backend REST API following HATEOAS principles, as well as a structure of 'forums' and 'subforums' to organise posts as described above.

Further supergroup requirements arose as endpoints in the protocol were developed. These included storing individual information associated with a user (e.g. profile images), advanced information in posts (such as time of creation/last edit), and an upvote/downvote system with both posts and comments. Moreover, discussions on security led to the use of digital signatures in authorising inter-server requests, adding further to the specifications.

Finally, within our own group, several additional requirements for our system were set in order to better tailor the user experience towards a university environment. User permissions and moderation were one important example; much discussion took place within the group on deciding how to avoid explicit content and ensure the system could not be abused.

Ultimately, we were able to develop a functioning system that met a large majority of these requirements while following software development methodologies. A scrum framework with meetings, sprints and continuous integration was initially adopted; as the academic year progressed, members of the group learned how to work more productively together, and this framework was further revised with respect to Agile practices.

Further elaboration on how requirements were met, the design choices involved, and the Scrum/Agile development methodologies involved will be given in later sections.

Development Timeline

This section focuses on the main events that occurred during the development of our system as well as the federated platform as a whole, and serves to provide context for the following sections that describe the implementation of the system and the group's use of Scrum/Agile methodologies in further detail.

A rough timeline was initially set by the module: The first semester would involve developing a 'Minimum Viable Product' of our system, focusing on the correctness of its core functionality. Following this, the second semester would involve working towards a more complete implementation, with additional features and a greater focus on user-friendliness. Further planning within the module was conducted by students themselves.

The summaries of technical discussions in weekly Scrum meetings can be found alongside the deliverable, as 'Meeting Minutes' documents. It should be noted that towards the end of the project, summaries stopped being taken as more work focussed on the report and not technical aspects.

Before the system began development, initial plans for developing the system were laid out. System features were explored, organised and prioritised by the group to decide the extent of functionality that could be included in both the MVP and full implementation. Time was also taken to consider potential options for the project's technology stack - tools for the front-end, back-end, database, and managing the project's development altogether.

Two supergroup meetings were also held at this stage, mainly to decide a structure for backend APIs that each group should follow. Very general 'supergroup requirements' were established at this point - primarily the nature of the API itself (RESTful, with data sent in JSON), as well as the concept of 'forums' and 'subforums' to organise content.

As development of the system began, members became increasingly familiar with developing the system, and additional tools that had been previously decided against were incorporated into the development of the system. Adaptations to the Scrum framework were made to account for differences when developing in a university environment, allowing members to work more efficiently. Meanwhile, testing and CI was also implemented to an extent by configuring the GitLab pipeline to automatically run backend tests.

Ultimately, the initial requirements of users, posts and comments were all implemented in time, though forums/subforums had to be excluded from the MVP. 'Inter-server communication' was instead achieved by showing multiple instances of our own server could communicate with each other.

Supergroup meetings were only held sporadically at this point, as teams had to finish implementing basic features of a discussion forum before they could focus on inter-server communication. Most changes in the specification were to remove ambiguity from descriptions in the original specification, such as the 'type' of ID fields.

Over the second semester, the supergroup protocol was rewritten to follow the Swagger specification. This allowed the protocol to be rendered as interactive documentation with Swagger UI, and therefore made it easier to understand. It was also moved from a GitHub gist to an actual repository, allowing all supergroups to directly propose changes.

Meetings to discuss any modifications or extensions to the protocol therefore began to be held regularly, leading to many developments in the supergroup specifications. Some of the additional supergroup requirements were beyond what had been initially discussed from the first semester - such as infinitely nested comments, a voting system, and inter-server authentication. Because of this, a number of planned features for the final product had to be removed from the product backlog, including notifications and a mobile site.

Several complications also arose from the introduction of inter-server communication, and various changes in the system had to be made. Endpoints on the backend were restructured several times to deal with requests from other servers, and the API itself was modified to adhere to HATEOAS principles. As a result, iterative development was taken more seriously, with backend tests changed to run through a proper API development platform.

However, the Scrum and Agile practices adopted by the group allowed the project's development to progress relatively smoothly even after these changes in plan, and all of the supergroup features were able to be properly implemented on time.

Project Details

(180014643)

Overview

Initial specifications distributed by the module coordinator indicated to us to consider a social media which might have applications in a university setting. The final system certainly fulfils this criterion, however is to a large extent flexible, offering the capabilities necessary to realise an online space for a vast array of online communities. The specifics of the system capabilities can be split into two parts. Firstly there is the super-group communication protocol which specifies what a server must be able to do in order to correctly interact with other servers in the federation. This communications protocol is comprehensive but intentionally minimal, specifying all the behavior one would expect from a functional social media but not constricting in any sense as to allow groups to implement the functionality they consider to be important. This specific functionality is the second, more concrete part of the system's capabilities. This obviously includes everything specified in the super-group communication protocol (channels, subchannels, posting, commenting, up/downvoting), but also features specific to our system, (banning users, admin users, profile pictures).

The platform is organised into forums, which in turn are organised into subforums. Both forums and subforums are categorised with titles however it is only within subforums that posts can be made.

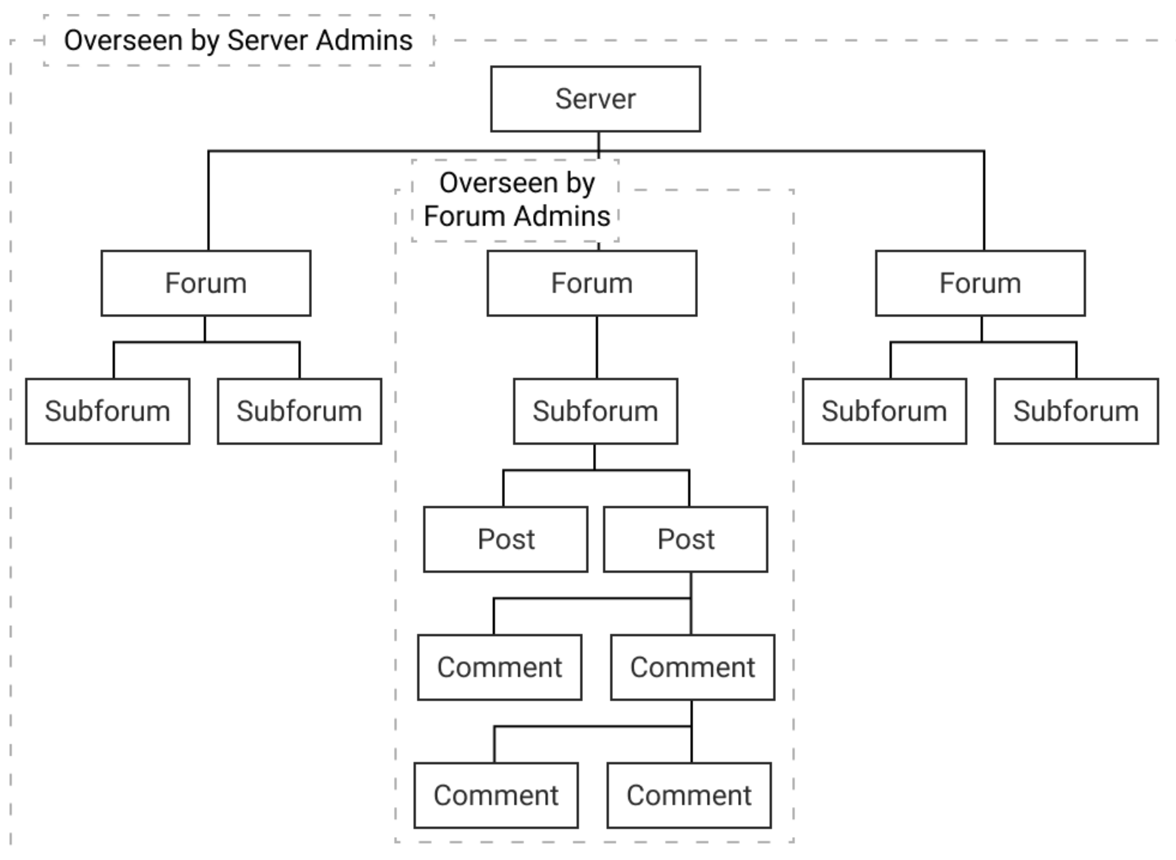


Figure 1 - Diagram showing an example of the structure of the system.

Looking at figure 1, forums contain some number of subforums, which contain some number of posts, which can have some number of comments. Each comment can be replied recursively, ad infinitum. This structure is universal between all servers in the federation. The hierarchy of admin overseeing is specific to our implementation.

Technology Stack

Multiple technology stacks were considered for the project, listed in the table below. Overall, it was decided to use the stack that would be the most optimal given the skills in the team, so that there would be no time wasted struggling with new languages.

The stack consists of:

- Frontend – React JS, CSS Modules and Sass.
- Backend – Loopback 4 with Typescript.
- Database – MongoDB.
- Management – Clubhouse and Microsoft Teams.
- User authentication – JWT tokens
- Server authentication – Digital certificates in the form of HTTP signatures

The system can be cleanly separated into two parts: a frontend built with React and a backend built with Loopback4.

React JS was chosen for the front end due to it being the only frontend framework that someone on the team has experience with, and for it's flexible and declarative style. The plan was to keep it simple and not use any of React's additional frameworks like Redux. React also has very nice syntax in the form of JSX which combines Javascript and HTML, and allows the rendering logic to be written alongside the UI logic since the two are tightly coupled.

For managing CSS, CSS Modules was used to stay in line with React's component-based architecture and reduce complexity. Additionally, SASS (CSS extension language) was used, mostly to keep things simple - SASS provides quality of life features like nested styles.

Loopback 4 was selected for the backend framework. The team had already agreed on using a Node JS based backend since everyone had some degree of experience with Javascript, where alternatives required experience with languages not everyone had used. Loopback 4 was chosen as it is designed for building REST driven APIs, and provided more structure than a bare bones express application. This emphasis on structure, however, would come back to bite us, as discussed later in the report. Loopback 4 also uses Typescript, which the team had assumed would be a benefit during development due to its type safety.

MongoDB was chosen for the database since it plays well with Javascript (and hence well with Loopback 4) – objects can be stored directly.

Clubhouse was chosen for tracking the project since it has a better feature set designed for larger projects, including managing sprints and milestones. There is more on Clubhouse later in the report.

Microsoft Teams was not considered during early development as a management tool, but it ended up being used anyway as a platform to communicate and hold meetings.

Technology.	Pros and Cons.
Front End.	
Bare bones HTML, CSS & JS	Pros. • Basic – everyone has used it before. Cons. • Too basic – A lot of functionality and quality of life features that are standard for web projects are missing.
React JS.	Pros. • It's a templating language, so it's great for writing efficiently, and allows JS to be written directly into the view. • Strong tooling support for JSX (React's hybrid JS with XML language) • Huge ecosystem for addons and libraries. • Reactive – Efficient when it comes to updating content with XHR operations etc using its Shadow DOM. Cons. • Only 1 person in the team has used React before, and not in a production setting, and there is a bit of learning curve. • View oriented – so data management can become an issue.
Vue JS.	Pros. • Easy to understand, templates are written in HTML. • No build chain like React. Cons. • No-one on the team has used Vue. • Too flexible for a large-scale project – different styles could creep in around different parts of the system.
Back End.	
Node JS.	Pros. • Language is JS – Majority of the team have experience in Javascript. • Fullstack JS. • Huge ecosystem of libraries, addons and frameworks (like express). Cons. • Dynamically typed – More prone to runtime errors.
Node JS with Typescript.	Pros. • Statically typed – More errors caught before runtime. • Lots of architecture in place. Cons.

	• Requires a pre-processing step to turn TS into JS.
Express (Node JS Framework).	Pros. • All pros of Node JS. • Widely used - lots of documentation and resources. • Minimal - Allows flexibility when developing, suitable for unusual requirements like federations. Cons. • All cons of Node JS. • Unopinionated - Extra effort will be required for consistency across the team.
Loopback 4 (Node JS Framework with Typescript).	Pros. • All pros of Node JS with Typescript. • Provides lots of structure - common things like CRON jobs and interceptors are available. Cons. • All cons of Node JS with Typescript. • Not widely used - less documentation and resources available.
ASP.NET.	Pros. • Very performant – 5x faster than Node JS. • Built in security. • Strongly typed. • Scales well. Cons. • Language is C# - Not part of the St Andrews course so few on the team know it. • Involves a build chain – more complicated to release.
Database.	
MongoDB.	Pros. • Majority of the team already knows it. • Flexible storage – MongoDB is schema-less since it's a document store. Cons. • Inflexible queries – Doesn't support JOINs and the like. • Not atomic – Doesn't support transactions.
Postgres.	Pros. • Allows ACID. • SQL based – majority of the team know SQL. Cons. • Uses a schema – which isn't suitable for a fast-moving project like this.
Management.	

Trello.	Pros. • Simple – Just consists of cards and places to put them. Cons. • Too simple – Missing a lot of features tailored to software engineering's requirements.
Clubhouse.	Pros. • Has a more serious feature set, including automatic burndown charts. • Has better organisational tools with different levels of abstraction - cards belong to epics, and epics belong to milestones. Cons. • It's a bit complicated at first due to how many features it has.
User authentication	
OAuth	Pros. • High level solution, uses already implemented login functionality of services like Google and Facebook, delegating actual authentication to these services • Allows for cross-functionality with server hosting user accounts (Google, Facebook, etc.), such as fetching that users photos or friends from that service for example. Cons. • Poor Loopback4 documentation making it difficult to implement • Reliance on external services like Google and Facebook does not align very closely with the federated system ideology which emphasises independence and autonomy as much as possible
JSON Web Tokens (JWT)	Pros. • Lightweight and minimalistic, requires no reliance on third party services. • Better Loopback4 documentation, including an example of a loopback application using JWT, usable as a template for our requirements. Cons. • No reliance on third party services means our system is responsible for all sensitive data and user information
Server authentication	
Central database of users	Pros. • Any user can log in from any instance of the system • Their login token can also be used as the server authentication signature, since a foreign server can consult with the central database to check if the token sent is valid.

	Cons. • Incompatible with federated ideology, as it leads to a reliance on a central resource. • Would require unnecessary supergroup overhead work
Digital certificates	Pros. • Aligns with federated approach since no centralised infrastructure required • Same system as that used by Mastodon, the most popular and successful federated social media in the public domain. • Allows more granular control for individual servers in the federation, e.g. a server can choose to not respond to any requests except those on a whitelist of servers within the federation. Cons. • Non-trivial implementation.

Table 1 - A table of the pros and cons of different technologies that could have formed the tech stack.

React

The react web app is served from its own frontend server which runs independently of the backend. Decoupling the frontend and backend meant that the overall system was more robust to failure, as the frontend could handle errors independently to the backend.

React has a modern component based approach, which makes it easy and fast to develop UI in small chunks. Many reusable components were written during development, and this led to shorter and shorter development times as the library grew. The components written are a mix of class and functional components, as at the start of development the benefit of functional components wasn't realised - that state can be refactored away from the component, which can't be done with class components.

While the initial goal of using React was to keep it simple, as the team became more familiar with it, more complex features were used. For instance the user context provider, which gives access to the currently logged in user anywhere in the component tree via wrapping a component with a React context provider.

CSS

CSS modules are used in conjunction with React components, meaning each component has its own module, and local styles. Each component is written to be reusable, meaning that any case specific styling has to be local to that case and not embedded in the used component. Heavily used components have several styles however, like size and width, in order to allow the widest use cases. For instance, the same component is used for *all* the buttons in the system.

A UI kit called Reakit was used for the frontend, but in contrast to normal UI kits, Reakit comes completely unstyled. It purely provides accessibility options for interactable elements like buttons and dropdown. All styling on the frontend is written by the team.

Loopback 4

The LoopBack 4 application runs on node and is written in TypeScript. LoopBack 4 is designed to make it easy to create CRUD REST APIs. It largely abstracts away the database it is running on top of, providing only a generic API. The wrapped database is referred to as a "datasource" in LoopBack terminology.

The backend is divided into two halves: "internal" endpoints which are only used by our frontend, and "external" endpoints which are those endpoints accessible to other servers in the supergroup. Internal endpoints which are only accessible to logged in users (e.g. POST endpoints) use JWT authentication. Some internal endpoints are accessible without logging in so that users can still view site content without an account.

All external endpoints use asymmetric key cryptography to verify that the server making the request is legitimate. The PATCH endpoints also check that the editing user is the same as the original poster, by checking the user-id in the header. It is understood that a malicious backend could put a fake user-id in the header to illegitimately modify any user's posts. However, we trust that all of the servers in our whitelist will not do this. If they did then we would remove them from the whitelist so that they don't have access to our server.

Digital Certificates and Inter-Group Communication

Part way through this project, it was realised that secure communication between servers within the federation would require an agreed standard method of self-verification which servers could invoke when sending requests in order to guarantee their identity. The reason for this was that a HTTP request in itself has no immutable characteristic which can be guaranteed to uniquely identify its origin. A naïve approach might be to use the ORIGIN header of the request to determine where a request has come from, however this can be straightforwardly spoofed, rendering it lackluster evidence of a server's true identity. The agreed upon solution within the supergroup was to create a digital certificate protocol which would be sent with every request made by a server. The goal of this protocol was to provide all servers in the federation with a guarantee that the server a request was being received from was in fact the server it purported to be, this worked in conjunction with the standard use of HTTPS, which guarantees that the content of a request has not been tampered with between being sent and received, to provide a robust and secure basis for inter-server communication within the federated system of the supergroup.

For one server *(server A)* to verify the identity of a server *(server B)* it is receiving a request from, it is necessary the sent HTTP request contains some sort of immutable “proof” of server B’s identity. In our system this proof takes the form of a header containing two critical pieces of information as well as some metadata. The critical pieces of information are the components of a message string (which can be reconstructed to form this signed message string), and the message string’s signature. The message’s signature is generated using a standard, agreed upon algorithm used by all members of the supergroup

(RSASSA-PSS-SHA512). The algorithm takes as input a string and private key and generates a signature which can then be verified by a third party possessing the corresponding public key.

A Loopback component called an *Interceptor* is used to handle foreign server certification. As the name suggests, interceptors can be used to "intercept" data being passed from one part of the system to the other. They can then perform any checking or modifying of the data required before passing the data to its intended destination. An example of how a HTTP signature certificate is created can be found in *Appendix 1*. Below is a diagram representing the process of server certification.

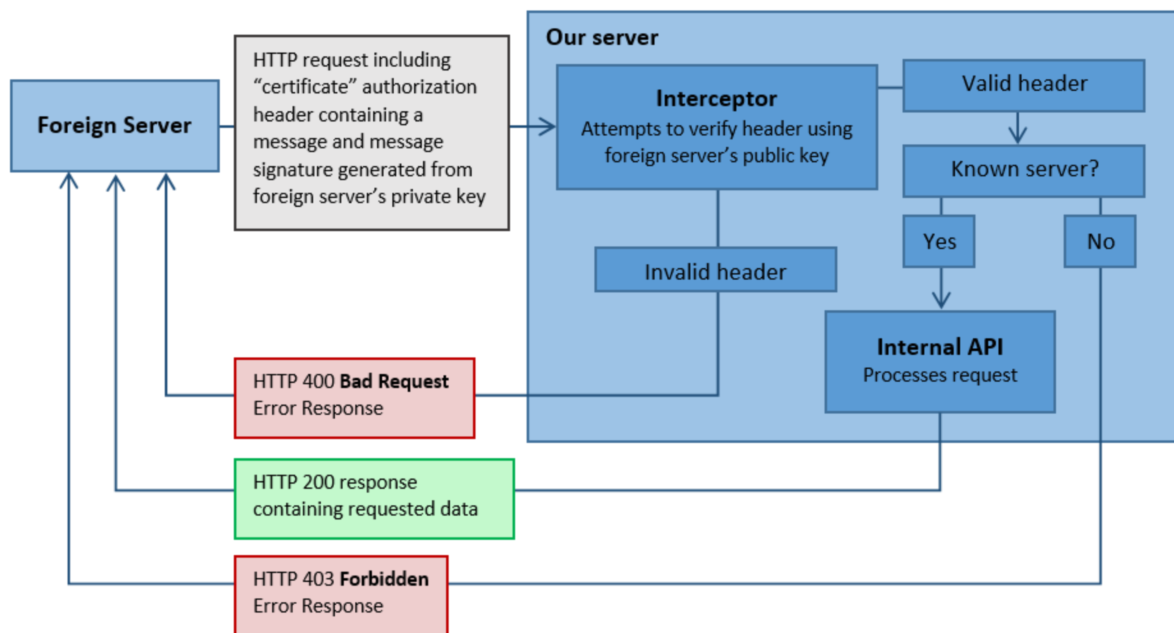


Figure 2 - Diagram showing the execution flow of the server authentication interceptor.

User authentication using JWT

Implementation of user instances within the system proved initially to be a surprisingly challenging aspect to realise. By the time it came to implementing users it had already been established that the documentation provided for *Loopback 4* could be oftentimes unwieldy, verbose and unhelpfully put together. This, combined with the fact that documentation and forum posts regarding the depreciated *Loopback 2* and *3* frameworks often appeared haphazardly muddled in with search results for queries regarding *Loopback 4* established a dichotomy in implementing system features whereby either the feature's implementation would be straightforward and work roughly as expected within a couple of attempts or it would become the bane of several members of the group for a prolonged period, often stretching over the course of several sprints. Such was the case with the implementation of users in the system. A (perhaps naive) commitment to doing things "the *Loopback* way" meant it wasn't until a somewhat obscure example of a *Loopback 4* application with basic user functionality implemented using JWTs was found that real progress on this aspect of the system could be made. The source code from this example, combined with the array of

disparate knowledge on *Loopback 4's* authorisation system gathered previously, meant it finally became possible to achieve a working user authentication system.

Implementing JWT authentication proved to be a surprisingly non-trivial feat. The various difficulties encountered in its implementation showcased a microcosm of issues with *Loopback 4* which are discussed in more depth in the *Evaluation* section of this report.

MongoDB

MongoDB is a NoSQL document oriented DBMS which stores data as BSON or "binary JSON". This is similar to ordinary JSON apart from the fact that boolean and numeric types are represented in binary instead of a text representation in order to save space. Our use of LoopBack meant we did not make API calls directly to the MongoDB driver within our application. We used mongodump and mongorestore to share data with each and easily upload it to the host server. This was useful as our schema was frequently changing so we needed to continuously update our local databases in order for the backend to work.

An alternative database system - MariaDB - was considered halfway through development, due to the heavy reliance on relations the current system had developed. MariaDB is a MySQL relational database system which stores data according to a strict schema. Eventually the decision was made *not* to use MariaDB, as all database API calls were done through the Loopback 4 interface and this would not change no matter what database system we used.

GitLab, nginx and running on the host servers

We used gitlab as required as the central repository for all our code. We also used gitlab runners which allowed us to automatically run integrated tests on pushed change. Debugging issues with the runner was difficult because it is somewhat of a black box. Its environment only exists temporarily, and all it can do is run the CI script, it can't be debugged from the command line. Every time a change was made to the CI script, testing it required waiting for the container to spin up, which took tens of seconds. We opted not to use CD because we found that deploying on the host server often required tweaking the code and so it was best to deploy changes manually and then thoroughly check that everything was working. The code may pass the runner tests and yet still not work on the host machines due to issues with things like .env files, port numbers, and nginx configurations. Furthermore, we were not regularly testing against other implementations until the end of the year and so it was not necessary to always have the most up to date version of our system running on the host servers

We used nginx to serve our frontend and backend on the same port. There was no choice but to use nginx since that is what the school host servers run on. nginx did not cause us too much trouble because some group members were already familiar with writing nginx.conf files.

Initially we didn't know how to get MongoDB to run on the host server because it is not installed on school systems and the usual way to install it requires root permissions. After asking fixit for help, Stuart Norcross sent us a helpful email guiding us through the process,

which we forwarded to another group that we knew were using MongoDB as well. He also explained how each group has a sudo user, which is something that we were not informed about during lectures.

Features

A social network by definition has to have users, user authored content and means to allow users to express opinions about said content. The following sections look at the features implemented as part of our system. Both the features implemented on a supergroup-wide level (i.e. those supported by all servers in the federation) and features implemented independently on our system instance are discussed. A brief discussion of features which were originally considered for implementation but were not included in the final system follows this.

Supergroup features

These features were implemented by every group within the federation. A systematic list of these features was standardised as an API protocol. For the most part, these features coincide closely with the system functionality outlined in the original specifications from the project.

Aspect of system	**Feature**	**Explanation**
Posting	*Create*, *edit*, and *delete* posts to any subforum in the federation	The most basic feature for a social network is being able to author and view content. Editing and deleting provide a quality of life feature that improves user control, by allowing them to correct mistakes. Several systems in the federation (including ours) support markdown in the body of posts.
	Upvote and downvote posts	A common part of any social media is an abstract way to indicate approval/disapproval of something. The supergroup uses upvoting and downvoting *a la* reddit. Allows servers to implement post aggregation systems if they wish to which will be able to aggregate posts made to and from any part of the federation.
General	Server certification	Servers must be able to recognise requests from other servers in the specification. This is achieved using digital certification and asymmetric key cryptography. Security and privacy for a social network is important for users to feel comfortable using the site. Having digital signatures for inter-server requests allow federations to restrict what can access their users information. Servers can control access to

		their system by only considering requests from a whitelisted set of servers.
	Fetching content from other federated instances.	The core mechanism for a federated content system. Due to the nature of federated content, all data must not assume locality of reference. That means that each piece of data is accompanied by an array of full URLs to navigate request related data (HATEOAS). The frontend uses these links via a relay in the backend to securely make requests to other servers. A relay was required as the frontend is incapable of creating digital signatures.
Commenting	*Create*, *edit*, and *delete* comments on any post made by any user within the federation	Another basic feature of most social media, facilitates conversations and discussion of ideas.
	Commenting recursively on posts	Most social media will allow some level of nestedness within comments. Within the supergroup, it was agreed to support a *reddit-style* system where comments can be infinitely nested.
	Upvote and downvote comments	Similar to upvoting and downvoting posts.
Users	A signup and login system to populate systems with users	In our system, JWT (JSON Web Tokens) are used. Hashes of passwords are stored in the system's backend to keep user credentials secure.
	Get information about an arbitrary user	Another common aspect of most social media is some sort of profile page for users. To allow instances from any part of the federation to display informative user pages, it was necessary to have a set way of fetching user information. This included a user's username and a list of all posts they had made on their native instance within the federation. Some instances (such as ours) also have support for profile pictures in the forms of URLs.
Hierarchy	Structure of system into forums and subforums	Each instance within the federation consists of some number of forums, each of which contains some number of subforums. It is within these subforums (and not within forums) that posts can be created.

Table 2 - List of features that were required by the supergroup.

Extended features

On top of the functionality outlined above, our system also supported several extended pieces of functionality. These are outlined below.

Aspect of system	**Feature**	**Explanation**
Posting/commenting	Embedding content within posts and comments	Posts can include more than just text. Images can be attached via URL and videos can be attached via Youtube URL.
Authorization	System of authorization to ensure that certain actions can only be made by privileged users or owners of content	There are two types of privileged user: server admin and forum admin. Forum admins have the role of moderating content and they can delete posts or ban users from a forum if they deem content to be inappropriates. Server admins have forum admin privileges across all forums and they are also able to ban users entirely from logging in to the server. Posts and comments can only be edited and deleted by either the original creator of the post/comment or an admin with the relevant scope.
Users	Editable profile pictures, descriptions and *"cake days"*	Part of the supergroup protocol but not required to be supported by all groups, our system supports users setting profile pictures, descriptions as part of their profile, and by recording the time at which their account was created, a "cake day" (*a la* reddit) a cake day was generated for users within our instance.
Hierarchy	Dynamically create new forums and subforums	Users are able to create new forums on the server and (within forums on the local instance) create new subforums too

Table 3 - List of features that were implemented in addition to the supergroup features.

Extension features proposed but not implemented

Over the course of the two semesters spent working on the project there were several features considered for implementation but which in the end were not implemented. Some of these features were decided to simply be beyond the scope of the project (instant messaging) while others would have been good inclusions to the project however their implementation either wasn't feasible due to time constraints or would have required a large amount of time to be set aside for only a small gain (e.g. uploading profile images would provide advantages however simply using URLs was far more straightforward to implement and offered almost exactly the same functionality).

Feature	Reason for not implementing
Not implemented	
A direct messaging system using websockets.	Having an instant messenger feature is common on social media networks (Facebook, Instagram and LinkedIn for instance). However, since this requires two way communication between the frontend and backend, websockets would be required - and the project has no foundation for implementing such a feature in the time provided.
The ability to upload photos.	Having a mechanism to upload resources like photos would have multiple benefits across the system, from uploading photos for posts to upload custom profile pictures. It was decided not to implement this feature, as the existing technique of just providing a URL to an image already on the internet was deemed sufficient for use for photos in posts and comments, as well as profile pictures.
Pinning posts.	Being able to pin a post to the top of a subforum is a useful feature seen in a lot of social media networks like Facebook. It allows users to draw attention to subforum rules or FAQs. It was decided not to implement this feature as it is deceptively complex. The backend would have to still serve the regular subforum to foreign servers, but a special subforum to the frontend, and the effort required to implement it was too much compared to the gain the feature would bring.
Subscribing to forums / subforums and aggregating that into a central feed (like reddit).	Allowing a user to create their own aggregate feed by subscribing to forums / subforums is a feature seen in social media networks like Facebook and Reddit. It allows user control in what they see. The reason this feature wasn't implemented was simply time. However, it was popular among the team and would likely be the first to be implemented given more time.
Push notifications	As with instant messaging, this would require looking into an additional way for frontend and backend to communicate (specifically using the javascript Push API). This feature was considered to provide minimal functionality, and it was decided time would be better used implementing and improving more important features.
Mobile site	Most social media platforms also provide a mobile site for viewing on non-desktop devices. This would be a good feature to have implemented. However as with subscriptions, time simply didn't permit its inclusion.

Implemented but later removed	
Post & comment pagination.	Only loading a certain amount of data (posts and comments) at a time reduces the load on the frontend and backend, and would be a required feature in order to allow the network to scale. This feature was actually implemented for posts, however due to decisions in the supergroup that impaired its functionality, the feature was reduced back to a “load more” button at the bottom of the post feed. An outdated version of the codebase which supports pagination can be found in the “updated-interceptors” branch.

Table 4 - List of features considered, but not implemented, or removed.

Changes in Plan

One change from previous deliverables was going back on the plan to switch from MongoDB to MariaDB. One reason for this is that the way we store the newly implemented ‘likes’ system is non-relational, as the users who have ‘liked’ a post are stored as an embedded array. The advantages of switching to MariaDB were also diminished by LoopBack's limitations. For example, in LoopBack it is impossible to have a compound primary key. Using a table with a primary key that is a compound of two foreign keys is the relational way to model a many-to-many relationship.

Another mechanism that evolved constantly over the lifetime of the project was how the server would communicate with foreign servers, as can be seen in figure 3. As the implementations of inter-server communication continued to develop, various complications arose with currently existing systems and the mechanism was ultimately rewritten several times.

Initially, there was the dispatch controller, which the frontend queried, providing the server ID and other relevant IDs in the URL, and the backend would trigger a redirect to the appropriate server. This approach meant that each piece of data on the frontend had to be tagged with the server ID it belonged to. Since the type of query was hard coded into the URL (`/servers/{:id}/posts/{:id}` etc), every external endpoint (the ones part of the protocol) had to have an equivalent internal one. This leads to many issues, like server authentication headers later on, and a lot of redundant code. The next evolution was the local-foreign controller, which instead of triggering a redirect, made the request directly. The local-foreign controller still relied on server IDs and hard coded internal endpoints. As the project developed, the huge flaw was discovered in using server IDs (discussed earlier). This prompted the final iteration - the relay controller, which behaves similar to the local-foreign controller by making a request for the frontend, except this time it uses the link provided by the frontend as the target URL, completely removing the need for server IDs. It only exposes 4 generic relay endpoints, for get, post, patch and delete - while it could all be done with one, it was split into request types for readability on the frontend.

The new design not only cut down on the amount of code but also better conforms to the HATEOAS principles the supergroup API is built on. This means that if the URL structure were to change in future we could adapt our code (backend and frontend alike) much more easily.

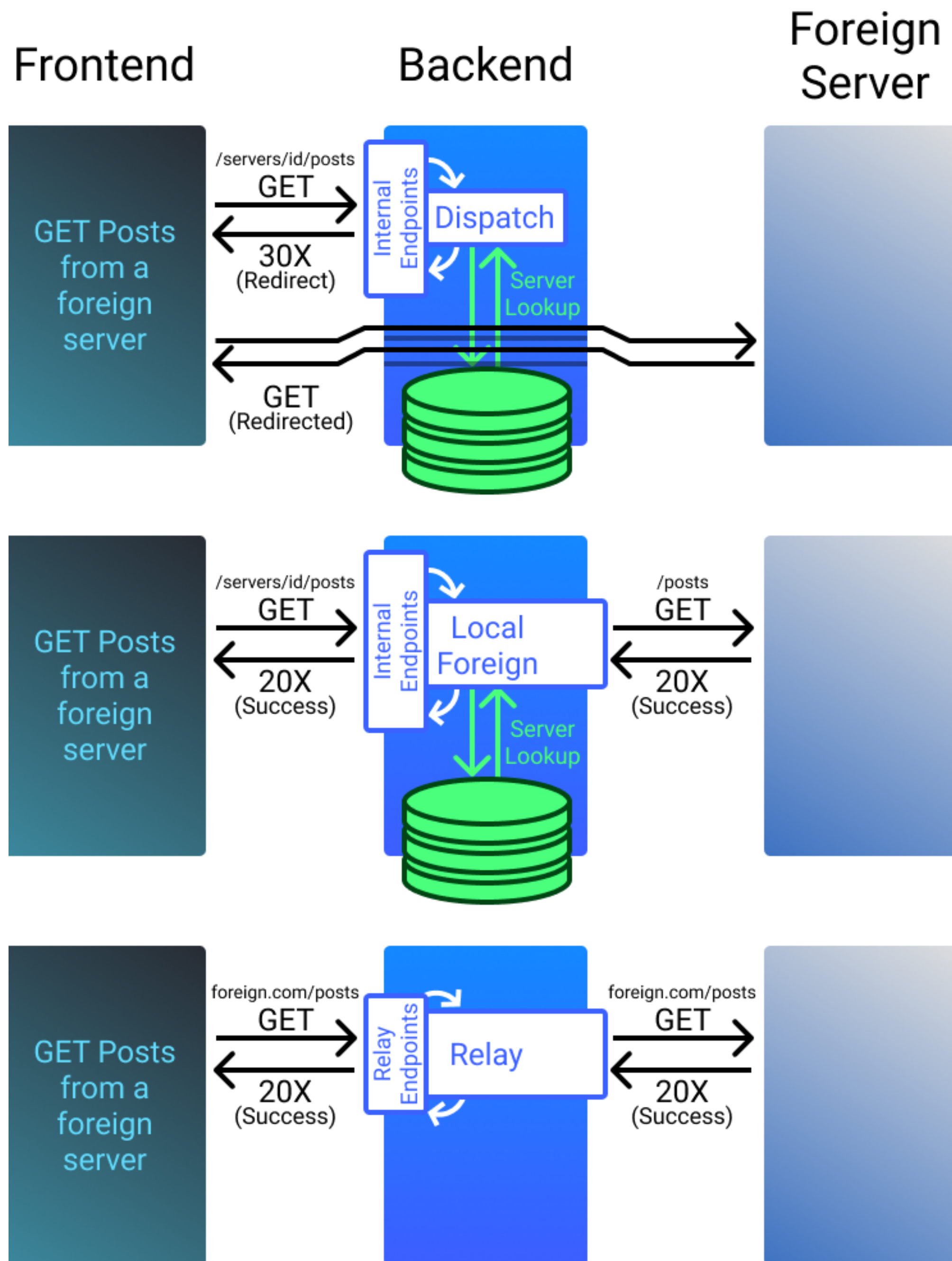


Figure 3 - Diagrams showing the dispatch, local-foreign and relay controllers as they deal with requests from the frontend.

Testing Summary

(180002209)

Testing is an essential part of developing any robust system. We made use of a variety of methodologies and techniques to rigorously test our system. This helped us to more easily manage the development and management of features during development and also gave us confidence in the robustness of our system after completion.

Backend

Methodology

The system's backend was comprehensively tested using an *"end-to-end"* approach. This is a testing approach which abstracts over the entity being tested, treating it as a *"black box"* similar to how a user generally treats a system. "End-to-end" testing involves testing the system comprehensively, as a whole, rather than as individual modular parts, this provides several advantages to other methods of testing.

- Component integration: For the system to comprehensively behave correctly as a unified whole, each part must also work as expected individually, therefore multiple aspects of the system can be tested simultaneously, and the same test can also test that these aspects not only behave in isolation but also integrate with each other correctly.
- Simulate user interaction: End-to-end tests are generally based around simulating user input (or in our case, simulating front end requests to the backend), this minimises the risk of a disconnect between what a developer may consider it important to test and how users may actually interact with the system, since in this scenario, they are ideally as close to one and the same as the developer can achieve.
- Easy to modify: Aided by the supergroup system protocol, tests could be written agnostic of implementation, meaning internal changes to the system "black box" require only minimal (if any) modification to the relevant tests.

To systematize this methodology, the testing application Postman was used. Postman is an API testing platform which facilitates the efficient creation of such tests by providing an easy to use and efficient UI and set of tools (environment variables, exporting of tests as JSON, team collaboration tools, etc.). These tests could then be run as part of the *Gitlab* pipeline using an *npm* package called *Newman*.

Postman tests are based around sending requests. The aspects of a request (the method, body, header, and target) can all be set and tweaked and the response can be analysed using concise Javascript scripts which can evaluate whether the response meets certain requirements (e.g. response had code 200, contained a field called "postTitle" equal to "new post", etc.). Postman also supports environment variables which were used to build more complicated chains of requests (e.g. one request can send a GET request to fetch a post and then the response's "postId" field can be saved as an environment variable for use performing a POST comment request to comment on that same post).

To more easily manage these tests, they were split into several groups, each more or less focusing on a specific aspect of the system. Below a brief description of what was tested within each of these groups. A comprehensive and detailed explanation of every backend test written can be found in the spreadsheet attached as part of this deliverable (*Testing spreadsheet.xlsx*)

Authentication, Navigation and Basic Use

These tests concerned the basic use of the system from the perspective of a user wishing to log in and interact with the system. Functionality such as signing up, signing in, making posts, and getting posts, (and doing the same for comments) is all considered in this section, as well as things like editing and deleting posts.

Inter-Server Certification

These tests were concerned with the protocol used between servers to verify one and other via digital certificates. This involved checking that the backend exclusively accepted valid certificates but no others. These tests check that invalid signatures are appropriately rejected and valid ones allowed through to the rest of the system.

Voting

These tests verified that the upvoting behaviour worked as it should. There were two aspects to this; a “micro” level, consisting of checking that vote requests from a single user worked as they should (e.g. user can’t upvote twice, a user downvoting a post they have already upvoted decrements the total by 2 not just 1), and a “macro” level, testing that votes from a multitude of users were correctly recorded. These tests required some slightly awkward Postman maneuvers, for example creating the set of users for the “macro” tests required building a *for-loop* by continuously setting a request to fire again, decrementing an environment variable each time until said variable hit a certain threshold. Although it was inconvenient such features weren’t more readily accessible through Postman, the workarounds found were still rigorous and reliable in their behaviour.

Authorization

These tests verified that authorization mechanisms worked correctly at both the server and forum levels. It tested that a privileged user is able to ban people at either the forum level or the server level, and that these bans have the expected effects with the appropriate scope.

Frontend

The original goal for testing React components was to verify that their internal state was behaving as expected. However, the mainstream testing methodology (and as a result testing tools) are geared towards testing from a user perspective. Testing from a user perspective means it’s difficult to test internal state since the user doesn’t have access to said state.

After debating the effectiveness of this approach of testing, it was decided that the team wouldn't use it and little frontend (automated) testing would be done as it is in general not appropriate for the functionality expected of the frontend. Instead, lots of manual testing from the devs and other users was carried out continuously during development.

That being said there are still automated tests for the plain Javascript end of the frontend, just not for components and their trees.

Continuous Integration (CI)

Tests were run upon commit to the Gitlab repo via a straightforward continuous integration pipeline. As mentioned earlier, the *npm* module *newman* was used to run the written Postman tests within the CI docker.The CI pipeline effectively performed the same steps as a member of the group would do when running the Postman tests locally by setting the *.env* variable “TESTING” to “TRUE” (More information regarding this can be found in *Appendix 2*). Since the pipeline would instantiate a new Docker instance whenever it was run and build the program from scratch, it removed any chance of strange behaviour which sometimes occurred when recompiling the system on top of an already compiled codebase. It also eschewed issues resulting from leftover database content, such as more than one server being in the Server collection, which can cause the tests to fail.

Although testing APIs, or web applications in general was not something members of the group had prior experience with, the testing approach taken proved to be a powerful, comprehensive, and intuitive one.

Software Development Methodology

(180012847)

Tools

Work had to be done remotely due to the current pandemic situation, so all tools had to have asynchronous distributed interfaces, since in most cases team members would be working in different locations at different times. Working remotely places additional restrictions on how the team collaborates, as being physically present allows spontaneous building of rapport (Shneiderman et al. p. 392).

For planning and directing sprints, an online development-tracking and management tool called Clubhouse was used. It provides a system for managing project tasks and gauging development progress through organising and tracking the progress on stories. Apart from assigning stories priority levels and ‘story points’, they can additionally be arranged into different levels of abstractions, with a group of stories making an epic, and a group of epics making a milestone. However, as discussed in the evaluation, Clubhouse ended up being primarily used as a product backlog.

Meanwhile, Microsoft Teams was used by both the group and supergroup to host meetings and facilitate communication. Teams was also used to store some important documents during development as it came with a built-in Microsoft Word processor, though alternatives such as Google Docs were eventually used due to versioning and collaboration issues that were found with Teams.

Finally, Git and GitLab were used for source control, as required by the project specification. Unfortunately, no-one on the team had experience with GitLab, so some additional time was required to get used to its features. Additionally, several issues were encountered with GitLab - a SOCKS4 proxy was required to access it outside the university network, and the university’s restrictions placed on the group repository also made it difficult to set up any CI/CD pipelines, as discussed later in the evaluation.

Use of Agile + Scrum

Throughout the development of the project, the Scrum framework was used to prioritise the requirements of the system and establish regular communication between group members. This was to ensure that everyone was informed about the state of the system and had clear, focused tasks to perform. As the project progressed and sprint retrospectives were performed, various changes were made to this framework with respect to the aims of Agile practices, ultimately allowing team members to work more productively together.

The first Scrum meeting was used to plan out various aspects of the methodology that members would follow. It was initially decided to hold scrum meetings (and therefore begin a

new sprint cycle) once every several weeks with regular, scheduled stand-up meetings held in between, so that members could raise any complications during sprints and receive help.

Moreover, tools were decided upon in order to help with outline planning. One important decision was to use the **Clubhouse** app to build up and maintain a product backlog; as more scenarios and user stories were collected, further meetings then elaborated on the specifics of how this app would be used. This included organising user stories into backend/frontend categories, assigning them one of 4 priority levels based on necessity and relevance to the MVP, as well as the use of a 'difficulty index' to indicate our estimated time for their implementation.

Sprint retrospectives at this time mainly noted difficulties with finding a meeting time that everyone was capable of attending, and so it was suggested to assign someone as Scrum Master to organise meeting times, while holding members accountable to important responsibilities.

After a majority of the outline planning had been done and development began, the following scrum meetings were used to plan sprints for building an MVP, based on the user stories organised in the product backlog. The organisation of tasks and assignment of priority levels were extremely helpful in directing each sprint; as the system expanded in complexity, members were able to focus entirely on developing the front-end or back-end separately, while still having a good grasp on the overall state of the project.

Smaller stand-up meetings held between sprints were very helpful in clearing up confusions about interactions between the front and back-end, such as the existing API endpoints or property names of JSON objects returned by the API. The meetings were also used as an opportunity to properly discuss how to merge different branches of the GitLab repository, ensuring that new feature implementations did not conflict with each other.

The stand-up meetings also helped facilitate regular communication between the group. This was particularly important when a major issue was encountered during the implementation of user authentication, where the relevant Loopback documentation was very limited. Multiple members of the team worked together to research potential solutions, and while the product backlog had to be changed to exclude forums and subforums, the issue was resolved over the following sprints without greatly affecting any other aspect of the project's development.

Continuous integration was additionally implemented at this point, with a GitLab runner configured to run end-to-end acceptance tests for the backend after each commit. This ensured the product was always in a stable, working state throughout development, and was very useful in helping members quickly solve bugs in current features, so that they could stay focused on implementing new features in future sprints.

During the development of the MVP, these development processes continued to be reflected upon, and the group decided to make several changes to the previously designed Scrum framework. With reference to the Agile practice of prioritising 'individuals + interactions' over rigid processes, it was thought that rather than following the framework exactly as planned, a

better option would be to adapt our approach and account for the differences with developing in a university environment.

One major change was shortening sprint times to a single week due to the limited time available to develop the project. Longer sprint times carried a greater risk of falling behind schedule, as there would be fewer Scrum meetings; members would therefore have fewer opportunities to reflect on the amount of progress achieved with respect to the deadline, and make adjustments to future sprints as necessary.

Another change was the development of the product backlog itself. While the current system of prioritisation and organisation were helpful, the ‘difficulty index’ on tasks was excluded as they were often irrelevant; team members could have other responsibilities during sprints, meaning the time they could dedicate to the project was unpredictable. More focus was directed in upholding regular communication instead, so that members struggling with sprint tasks could receive help from others.

Finally, the designation of a Scrum Master was found to be unnecessary due to the small group size. It was found that the responsibilities of organising a scrum were simple enough to be covered by any member of the team whenever convenient, and enough trust and understanding had been established between members for tasks to be completed without the explicit need for accountability.

Over the second semester, scrum meetings continued to be held at a fixed time every week. Having finished a majority of the outline planning, these events began to follow a much more regular structure aimed at upholding an Agile workflow: Facilitating communication to resolve issues, revisiting or clarifying requirements, and assigning responsibilities for the next sprint.

The product backlog was updated throughout to reflect the app’s development as well. Responsibilities for new features were discussed during scrums, and members were able to choose which new features they wanted to implement over the next sprint, with proper consideration of their priority and difficulty. This was very helpful in keeping members focused on their part in developing the app.

On the other hand, stand-ups between scrums began to be held irregularly, rather than following a strict schedule; emphasis was placed on the Agile practice of working with our individual circumstances over establishing a rigid meeting time for all discussions. This gave members a convenient way of receiving assistance and suggestions from other members when stuck on a particular task, such as deciding how user roles and permissions should be stored in the database. The development process was therefore very open to change, and difficulties with feature implementations were resolved much quicker than otherwise.

Furthermore, since most groups had finished basic forum implementations and could focus more on inter-server features, developments in supergroup discussions began to occur more often. As a result, the flexibility of these meetings were also critical in responding to the constant changes in the supergroup’s API protocol. On several occasions, members of the group met immediately after supergroup meetings to redefine the requirements for our backend when the protocol was updated - often regarding inter-server authentication

methods and the common API endpoints. This followed the Agile practices of evolving quickly to meet new standards, and overall greatly reduced the time spent developing towards an outdated specification, allowing the team to work more productively.

One such instance where this occurred was the removal of pagination-related endpoints from the supergroup protocol. Discussions regarding how this feature could be adapted were able to take place soon afterward, and an alternative client-side ‘load more’ button was implemented over the same sprint, with no further disruption caused from this event.

Finally, pair programming began to be utilised by members of the group - this process was particularly efficient in the process of fixing bugs, as multiple people could suggest ideas and error-check any written code, while gaining a better overall understanding of the system. In fact, a new ‘*Code With Me*’ feature of the WebStorm IDE was also found to be very helpful in solving larger issues. This tool allowed multiple people to access a person’s computer and make changes to the files together; members would get instant feedback on whether their changes conflicted with each other, and the code could also be run in the middle, allowing bugs to be found and resolved together.

Overall, the overarching ideas of Agile and Scrum were used to significant effect. Throughout development, members of the team had clearly assigned responsibilities every sprint to focus on, and Scrum meetings were good opportunities for everyone to catch up on the current state of the project and the responsibilities of other members.

Frequent communication during sprints and the iterative approach to development reduced the impact of unstable requirements or unexpected difficulties on workflows, especially during the second semester, as members were able to find solutions to their issues faster. By following Scrum and Agile methodologies with an emphasis on organisation, coordination and frequent communication, the team was able to work much more productively together.

Supergroup Interaction

(180014477)

Supergroup communication was not great during the first semester. Some groups did not attend the majority of meetings during the first semester which made it hard to agree on anything. The initial version of the API was almost entirely designed and written by Samuel Wykes alone. All edits had to be made by him because he had sole ownership of the API specification document, which was stored in a GitHub gist.

In the second semester the supergroup came up with a better solution to make it easier for others to contribute to the protocol. We set up a GitHub where each group could have one or two members who were added to the repo and could therefore propose changes via pull requests. The permissions on the repository were set up so that 5 members had to approve a pull request for it to be merged, and only one person per group was to approve each request. This way we ensured that changes and additions had a good consensus behind them before they were made official. It also meant that multiple people had to check over changes before they were approved which helped iron out silly mistakes. The API spec, which had been written in a JSON-like format, was translated to an open-api YAML document. This made it easier to put more specific information for each request type, although it also made the document a bit more verbose. It was frustrating that we had to use the third party service of GitHub rather than being able to set up a GitLab for the supergroup, since our main codebases had to be on GitLab anyway. We were forced to do it this way because the school does not give students the permission to create their own GitLab repositories.

By the end of the year there was a healthy amount of continuous discussion between each group. A spreadsheet was made in the supergroup Teams to keep track of the level of functionality that each group had implemented. After digital signatures were implemented, groups began testing against each other regularly. Group members would message each other on Teams to notify them of compatibility issues and we would work together sharing logs to aid in debugging. Overall there was a good level of collaboration between groups which allowed everyone to iron out issues well in advance of the deadline.

Evaluation

(180012847)

During the two semesters spent working on this project, a system was developed to comprehensively meet the criteria set out through the different deliverables distributed throughout the year. Part of this process involved communication with the supergroup to develop a systematised version of this functionality in the form of the supergroup protocol. The system also incorporated local extended functionality, such locally tailored functionality being a characteristic aspect of federated systems. Said functionality was implemented to a high degree of polish and tested as robustly as the fundamental aspects of the system.

Comparison to real-world technologies

Our system's implementation drew inspiration most heavily from two real-world social media platforms: Reddit and Mastodon. Mentioned explicitly by Edwin in the first deliverable specification for this module, Reddit provided the abstract notion for what our system should look like in the broad strokes.The obvious parallels being between subforums and subreddits, upvoting and downvoting is also inspired by reddit and the system's deeply nested comments are also a staple of reddit threads. Other less major aspects, such as the minimal user profiles and "cake days" were also inspired by Reddit.

While reddit provided much inspiration for the structure and functionality of our system, with regards to how federation could be achieved, Mastodon provided an illuminating case study. The most valuable thing Mastodon provided was a template for implementing secure supergroup communication via the use of digital certification. The method of certification used by Mastodon itself seems to be based heavily on an approach outlined by the Internet Engineering Task force (IETF) for HTTP signatures (M. Cavage et al) This approach became the backbone for our chosen method for secure inter-server communication within the project.

Backend

With hindsight LoopBack 4 was not a good choice for the backend. It had a number of downsides which were detrimental to development.

One of the biggest issues with LoopBack was its poor documentation. Example code was often incomplete or outdated, while several documentation pages led straight to GitHub issues. LoopBack 4 is relatively new and not backwards compatible with LoopBack 3, so it doesn't have some of LoopBack 3's features. It was sometimes hard to find out how to idiomatically implement a certain feature in LoopBack 4, whereas searching brought up documentation for how it would have been done in LoopBack 3. Additionally, a lot of the documentation and example code was about how to migrate from LoopBack 3 to LoopBack 4, which was not helpful or easy to understand for a team that never used either before. Searching for how to implement a feature often lead to GitHub issues describing proposed additions to LoopBack 4 rather than documentation or Stack Overflow answers describing how to solve the problem, as you would find for a more popular framework like Express.

One way we mitigated the lack of documentation is that one member of the group joined the LoopBack 4 official slack group to ask for help with a specific problem, which one of the slack members gave a helpful answer for. This was a good resource to have when we were completely stuck on a backend issue.

LoopBack recommends that you use their command line interface to auto-generate code. This perhaps indicates that their framework is bloated with boilerplate. It worsened the learning curve at the start of development because we generated the barebones of the application through the CLI, and then none of the team members understood the code, because none of us had written it ourselves.

We used TypeScript, which LoopBack 4 has first class support for. This meant that the application had a long compile stage every time the code was changed. The advantage of TypeScript over JavaScript is that it is supposed to catch silly errors at compile time (such as type errors, misnaming variables, objects being undefined). However, LoopBack has a very soft-coded system for gluing together different parts of the application. Classes are referenced by string literals, which aren't statically checked, and a slow, complicated dependency injection process happens at runtime. The error messages generated when dependency injection goes wrong are long and obtuse, and normally describe an error happening within code generated by LoopBack rather than source code.

The disadvantage of TypeScript (a long build time) was retained, while its advantage (screening out silly mistakes) was not. Additionally, TypeScript is only transpiled to equivalent Javascript, and does not provide any performance benefit, whereas a truly compiled language can perform many compile time optimisations and have a smaller runtime overhead (e.g. no runtime type checks).

LoopBack abstracts over the particular database used, but by doing this it also hides important functionality specific to each DBMS, providing only a generic CRUD API. At the same time, it doesn't fully abstract away the details of which database is chosen, because each datasource has certain features it supports and doesn't support, and there are some odd quirks to this which require digging through documentation to identify.

One of the main advantages of Node over other web servers is that because the backend and frontend use the same language, code can be shared between them, and it is less mentally taxing to alternate between developing each part of the system. This proved not to be too relevant to our project due to a strong separation of concerns between the frontend and the backend. One our project usually different people would be working on the frontend and backend, and features would first be added to the backend and then later supported by the frontend, so one person was not usually developing both at the exact same time. No code was shared between frontend and backend. The backend used TypeScript and the frontend used JSX. Both these languages are based on Javascript but they are somewhat different to each other. With this in mind, we could have used C# (one of the initially proposed languages) instead of Node and reaped the benefits of a more performant compiled language.

Alternatively if we were to stick with Node then we would use Express rather than LoopBack 4 as the amount of documentation and tutorials available for Express is much greater.

The REST API developed by the supergroup is quite long and repetitive. If we were to redo the project we would advocate for the supergroup to use a GraphQL API rather than a REST API. This would allow for more work to be delegated to libraries/drivers and reduce the amount of code in our applications, as with a REST API you need some code for every endpoint. A GraphQL API is really a query language that allows the requester to ask for data in the shape *it* expects, which would allow for more flexible federations. When we were adding query parameters to certain endpoints we were uncertain of what syntax to use. LoopBack itself supports two different syntaxes for query parameters, one of which is embedded JSON is the URL. In the end, for simplicity's sake, we did not suggest using the LoopBack syntaxes to the supergroup because they were too complicated. In GraphQL there would only be one way to write queries with filters, which would simplify the process of suggesting and agreeing on protocol with the rest of the supergroup.

MongoDB was also a poor choice with hindsight. Our data ended up conforming to a relational model more than initially expected, which is a common issue when choosing to use a NoSQL database for a project of this kind (Mei).

Frontend

React proved to be a good choice for the frontend. Some of the group members already had experience with it and those who didn't were able to get to grips with it much more easily than with LoopBack due to the abundance of documentation and online resources. As the project matured, the collection of bespoke reusable components grew, making it easier to quickly compose new components from old ones. React’s own JSX language made UI design and logic very easy to write as they were alongside each other instead of seperated. It’s quick compile time and development mode (recompiles on changes to the codebase) made iteration fast and efficient.

React isn’t without faults however. One of the main issues encountered when using React is how it handles data. Data only moves down the component tree, not up. So communicating information from one component to another could at times become an arduous design task, involving restructuring and rewriting many components in between. An option to deal with this kind of state management would be to use a library like Redux or MobX, which provide ways to manage all kinds of state, like global and nested. However, these libraries are large and difficult to use properly. The team would have to invest time in taking some training beforehand to gain the benefits of using a library like these.

CSS Modules also turned out to be a good pairing for React. It’s concept of writing CSS in modules meant that each component could get it’s own module, and hence locally scoped styles that would never clash. Using Sass with CSS Modules also proved to be a good choice, as one risk with using CSS modules is that there is a lot of repetition - but Sass introduces a system for inheritance of CSS styles and avoids this issue.

The frontend interface was evaluated against Nielsen's 10 Heuristics (Nielsen).

The frontend developed for the project is sleek, consistent and transparent - which are all tenets to good Human Computer Interaction heuristics. The interface is sleek as it only displays the information currently relevant to the user, and does so with a minimalistic

aesthetic. It uses bold blue colours to indicate a significant element to the user, and a vibrant red to indicate an element which deletes something. The interface is consistent thanks to Reacts component architecture. A post looks and behaves the same on the feed page as it does in the profile page. Loading behaviour is consistent across the whole app. The interface is transparent, as it keeps the user informed to its internal state due to heavy use of reactive information elements. Whether it's loading content, or if it's hit an error, it will let the user know.

What the interface fails at is error prevention, flexibility and error recovery. It fails at error prevention because interaction with destructive consequences has to confirm dialogue. For instance, deleting a server in the admin page has no confirmation before the server is removed. This means the user feels less confident in using the system. Sites like Instagram have confirmations before every destructive action, which results in less user error. The interface fails at flexibility because there is none - no preferences, no keyboard shortcuts and no experience tailoring. Social media sites like Facebook have keyboard shortcuts (ref) which allow experienced users to use the system more freely, and preferences like dark mode. Additionally, there is no mobile site version of the site, meaning users can't access the app on the go. This severely restricts the usability of a social network. The interface fails at error recovery, as even though errors are shown to the user, they are rarely in a user-friendly form. They tend to be error codes that only make sense to a developer and not a user. This means cannot correct any underlying error (if it were even possible for them to do).

Agile & Scrum

Various Agile practices were incorporated with our adapted Scrum framework. As the system expanded in complexity, communication, organisation and flexibility became increasingly important, and our use of these software development methodologies contributed significantly to our success, detailed below.

Increasing the regularity of scrum meetings was likely an important contributing factor. These meetings allowed us to further adapt our use of Scrum and evaluate our rate of progress more often, ensuring that we did not fall behind schedule. It also instilled in group members the value of frequent communication early on, and made everyone familiar with using Microsoft Teams to collaborate. Pair programming also happened as a result on several occasions, leading to some very productive debugging sessions.

During the second semester, our decision to increase the flexibility in meetings between sprints was another useful change. This took advantage of our small group size and familiarity with one another from the previous semester, allowing us to quickly adapt to changing requirements within the supergroup; even larger changes such as the removal of pagination in the backend could be solved within the same sprint.

Incorporating Agile practices related to incremental development were also very helpful during development. The implementation of automated tests ensured that the product backend was always in a functioning state, and greatly reduced the amount of time taken to find bugs in the system whenever changes were made; any errors in the system would

almost always be relatively new and therefore quick to fix, and so members of the team could therefore focus more on writing new code rather than fixing old code.

However, there were also several areas of Scrum and Agile that could have been reconsidered.

One aspect was our choice of Clubhouse for managing all aspects of development (deciding sprints, timeline estimations and holding feature discussions). Clubhouse was aimed towards larger projects with complex stories and multiple teams, and ultimately many of its features were inappropriate for a team of our size. As development progressed, members quickly found that keeping every story up-to-date with meetings felt more like duplication than organisation, and ultimately it was used as a simple repository for the product backlog. Sprint backlogs were instead managed through verbal discussions - and with our small group size, this was already sufficient to give members a clear objective over each sprint.

To have dealt with this better, the group could have assigned an actual Scrum Master for meetings - this would give them a responsibility of keeping the Clubhouse app up-to-date with each meeting, and they could further use it to direct scrum discussions. Another option, perhaps more suitable for a group of this size, would have been to choose a different tool altogether - such as Trello for its ease of use and simpler structure, better reflecting the nature of this project.

Another aspect that could have been improved upon was the use of continuous integration - particularly regarding testing on the front-end of the system. While back-end tests were very useful as 'white box' tests which considered the internal operation of the system, the team took too long to develop 'black box' tests that could be done on the front-end, meant to test the overall behaviour of the system itself.

This was particularly troubling when the API endpoints themselves, or their expected behaviour were being modified, as the current back-end tests would be invalidated. Moreover, outstanding issues from previous changes to the front-end would sometimes be missed. If the team had spent time developing front-end tests earlier, the errors caused by both of these issues could have been mitigated and resolved faster.

Supergroup Interaction

Supergroup interaction was fairly limited in the first semester, but this was understandable as many of the groups had to focus on developing implementations of core features first, so that there was meaningful data to be sent across servers. Being able to establish the general method of communication (REST) and a structure of endpoints (particularly subforums) before the MVP was already very helpful, as it set up a system that could be easily built upon during the second semester. The oversights in the protocol could then be fixed by relatively minor changes, and this kept development smooth for all groups.

Following this, supergroup interaction in the second semester provided a sort of implicit user testing environment, and continued to support development for all groups. Often one group would notify another if a feature of their system wasn't working as expected. This provided another layer of assurance that systems were working well.

However, inter-group functionality was a main issue that arose with testing, particularly near the end of submission. Local systems generally could not test inter-server features, and as a result it was difficult to identify the source behind an error from actions involving multiple servers; information could have been processed or displayed incorrectly on either side.

With hindsight, it would have been good to adopt a standardised set of tests across the supergroup earlier, so that we could automatically check our implementations against known common requirements. This would be possible through the end to end testing provided by Postman.

Conclusions

In conclusion we developed a system that met the outline of requirements given to us at the beginning of the year. The supergroup designed an API that was relatively effective. The digital signature system that we implemented was effective in verifying the entity of servers, protecting us from attacks from malicious users pretending to be a trusted server. We also protect ourselves from malicious users who log in to our site by implementing authorization mechanisms on our own instance to facilitate banning users and deleting malicious content.

We adopted the agile development methodology which allowed use to rapidly develop features and change requirements as needed. This was especially pertinent in the second semester when the supergroup agreed minimum requirements were changing and so we had to change priorities in order to develop these features.

The level of intergroup communication increased steadily over the course of the year and by the end of it there was a good amount of intergroup discussion and testing.

Our system has a clean and intuitive interface which makes the site functionality clear to new users. We have a strong separation of concerns between frontend and backend. This assisted us to design for graceful failure so that if some of the site content is missing or invalid then the site will still function for the most part.

There were a number of features left on the cutting room floor which could have been implemented, including the cut pagination system and a direct messaging system that was proposed but we didn’t have time to implement at all.

Acknowledgments

We would like to thank our supervisors Angela Miguel and Ian Gent, and our module coordinator Edwin Brady. We would also like to thank Stuart Norcross for his assistance in setting up our system on the school servers. Finally we thank every other group in supergroup B for their various contributions to the project as a whole.

Appendices

Appendix 1: Example of inter-server communication header construction

To illustrate exactly how inter server certification works, we look at an example. To construct a valid certification header, we construct a signature input string (split into several lines below for readability).

Signed string:

```
*request-target: POST https://cs3099user-b7.host.cs.st-andrews.ac.uk/api/subforums/1\n
current-date: 2021-04-07T23:02:24.204Z\n
user-id: 1\n
signature-input: sig1=(*request-target, date, user-id)
```

We then sign this string using our private key and *RSASSA-PSS-SHA512* with a salt length of 20.

Signature

```
gBxyJIB1jPfo/pNKG0YDA6kry4xmHJ3fNiGF7n/XSjUAfu9FIqwN/VQ80RDk442PmMP09OKyAl36wO3uvKelo66
LRiLx5ICJWCbefiO1eVAqSPPUN2KqicaSrueJXsv9OM/CGTl+NGTqODqs7j3SF9tF6OGCZPvAOflqrtae18itcL
b5Lmi9EzWEgP4oWme/vYatxJOYwgrZRkeP4LR5u8GJ24XnPFeLhqnBqqmR73HVm55uXGj2neaB+GFoSBFXOZ6uU
bc/lnGEEiG2wobo/VU1TDOLZgf5cDpOg0kmclc3jxxI1/5idgzdzdOIVbwBZuqdz6j/X+4xI0+/Y1lnB7SiMa4M
M60xII0FKYzSk81eWla2yw8yauDLyCAhP7pl0ItuDcQmQMzg0XdyonaMOaL2Yuuck1TegkOsCKhWmZhbDJT3nvl
//pdNMF/HDOa4XFr/NaO584ukukw0BpJCVwSBDhcakfbt99FcnSaOHx9ia3OaVDwbD65byJGgLH43DK+G5nMbMJ
wRecEQxs4UeL2h5A5lJOCY/kJhgO7rcXKhsmuDYtkfUyn2Su3jPi1D4xdxwj5gDX7gzVaAOU/G5QxMZ2SaYPn+e
Kc8isxYFUGzMpC57NjpusfmuLVP/Innj2p5+Gi3abQVn26BNFIZAP/Y9ETUx/oauPubEqdacIg=
```

Finally we construct our headers to send with our request to another server. This must include all the information above (except the **request-target* which can be inferred from the base HTTP request) as well as the location of our public key which the other server can use to verify our signature. Using this information, a foreign server can reconstruct our signed string and verify it matches the signature attached. At which point it can choose to grant us API access if it approves of our server.

Headers attached with request to foreign server

```
current-date: 2021-04-07T23:02:24.204Z
user-id: -1
Signature:
sig1=:gBxyJIB1jPfo/pNKG0YDA6kry4xmHJ3fNiGF7n/XSjUAfu9FIqwN/VQ80RDk442PmMP09OKyAl36wO3uv
Kelo66LRiLx5ICJWCbefiO1eVAqSPPUN2KqicaSrueJXsv9OM/CGTl+NGTqODqs7j3SF9tF6OGCZPvAOflqrtae
18itcLb5Lmi9EzWEgP4oWme/vYatxJOYwgrZRkeP4LR5u8GJ24XnPFeLhqnBqqmR73HVm55uXGj2neaB+GFoSBF
XOZ6uUbc/lnGEEiG2wobo/VU1TDOLZgf5cDpOg0kmclc3jxxI1/5idgzdzdOIVbwBZuqdz6j/X+4xI0+/Y1lnB7
SiMa4MM60xII0FKYzSk81eWla2yw8yauDLyCAhP7pl0ItuDcQmQMzg0XdyonaMOaL2Yuuck1TegkOsCKhWmZhbD
JT3nvl//pdNMF/HDOa4XFr/NaO584ukukw0BpJCVwSBDhcakfbt99FcnSaOHx9ia3OaVDwbD65byJGgLH43DK+G
5nMbMJwRecEQxs4UeL2h5A5lJOCY/kJhgO7rcXKhsmuDYtkfUyn2Su3jPi1D4xdxwj5gDX7gzVaAOU/G5QxMZ2S
aYPn+eKc8isxYFUGzMpC57NjpusfmuLVP/Innj2p5+Gi3abQVn26BNFIZAP/Y9ETUx/oauPubEqdacIg=:
signature-input: signature-input: sig1=(*request-target, date, user-id);
keyId=https://cs3099user-b7.host.cs.st-andrews.ac.uk/api/key; alg=RSASSA-PSS-SHA512
```

Appendix 2: Running *Postman* tests

The JSON containing the Postman tests can be found in the following directories:

Tests: *backend/cs3099-loopback/postman_tests/CS3099.postman_collection.json*
Environment: *backend/cs3099-loopback/postman_tests/CS3099_Testing.postman_environment.json*

After following the system repo's *readme* file to set up the system, an additional environment variable should be added to the backend *.env* file:

```
TESTING=”TRUE”
```

After which the backend can be run in the usual way using:

```
npm run start
```

The system will initialise itself before running all postman tests and logging a summary to the console before exiting.

Note: The database should be empty prior to running the system if it is being run in “testing mode”. The system will initialise a test database when it runs in testing mode and if there is already data present, the tests may fail due to unexpected things being in the database.

Bibliography

Mei, Sarah. “Why You Should Never Use MongoDB” 11 November 2013, http://www.sarahmei.com/blog/2013/11/11/why-you-should-never-use-mongodb/ Accessed 14 April 2021.

Nielsen, Jakob. “10 Usability Heuristics for User Interface Design.” Nielsen Norman Group, 24 April 1994, https://www.nngroup.com/articles/ten-usability-heuristics/. Accessed 11 April 2021.

Esguerra, Richard. "An Introduction To The Federated Social Network". Electronic Frontier Foundation, 2011, https://www.eff.org/deeplinks/2011/03/introduction-distributed-social-network Accessed 8 Apr 2021.

Shneiderman, Ben, et al. *Designing the User Interface*. 6th ed., Pearson, 2016.

M. Cavage, Joyent, M. Sporny, Digital Bazaar. “HTTP Signatures” February 1, 2014 ,https://tools.ietf.org/id/draft-cavage-http-signatures-01.html Accessed 15/04/2021